Machine learning system for data science

23 settembre 2024

Indice

Capitolo 1

Introduction

Remark 1 (Exam). multiple choice and translation of an algorithm into a python program

per l'orale possibili domande: what doest this code do, or does this code actually implement this algorithm

1.1 Sets

Definition 1.1.1 (set)**.** A set is a collection into a whole of

- *distinct*: it must be always possible to tell whether an object x is an element of a given set or not
- *definite*: any two elements must be different

objects called *elements* (members) of the set.

Important remark 1 (Membership). If Y is a set and X one of its elements, then we can say 4 equivalent things all written $X \in Y$:

- X is an element of Y
- X is a member of Y
- X belongs to Y
- Y contains X

Definition 1.1.2 (Set equality)**.** $X = Y$ if and only if X and Y contain exactly the same elements.

Important remark 2 (Empty set). There exists no more than one set containing no members, the empty set, written $\emptyset$

Remark 2 (Set denotation). A set can be written/denoted either by:

- *enumeration* of its elements within curly brackets $\{X, Y, Z\}$

- *property*: if X is a set (called *universe*) and P is a property which is either true or false for any member of X, one may write a set containing exactly the elements of X that make P true; this set is a subset of X and is written as

$$\{Y \in X : \text{P is true for Y}\}$$

or concisely as

$$\{Y \in X : P(Y)\}$$

where ":" may be replaced by the vertical bar "|".

Definition 1.1.3 (Subset)**.** If any element of X is also an element of Y, then we can say 3 equivalent thing, all written $X \subseteq Y$

- X is a subset of Y
- X is contained in Y
- Y contains X

Proposition 1.1.1 ($\emptyset$ and subset)**.** $\emptyset$ *is a subset of* any *set*

Proposition 1.1.2 (Set equality and subsets)**.** *If* $X \subseteq Y$ *and* $Y \subseteq X$*, then* $X = Y$

Definition 1.1.4 (Proper subset)**.** X is a *proper subset* of Y, written $X \subset Y$, if and only if $X \subseteq Y$ and $X \neq Y$.
Thus there exists an element of Y which is not an element of X

Remark 3. In some books and papers, $X \subsetneq Y$ stands for $X \subset Y$, and $X \subset Y$ stands for $X \subseteq Y$

Definition 1.1.5 (Set operations)**.** We have:

- The *union* of X and Y, denoted $X \cup Y$, is a set containing every element x such that $x \in X$ or $x \in Y$, and no other
- The *intersection* of X and Y, denoted $X \cap Y$, is a set containing every element x such that both $x \in X$ and $x \in Y$ and no other
- The *difference* of X and Y, or *complement of* X *relative to* Y, denoted $X - Y$ or $X \setminus Y$, is a set containing every element of X which is not an element of Y, and no other.
- When the second set is understood, we say simply *complement* of X , written X^c, or $\bar{X}$

Definition 1.1.6 (Set operations on multiple sets)**.** If $S_1, \ldots, S_n$ are sets:

- their *union*, denoted $\cup_{i=1}^{n} S_i$, is the set containing all the elements of $S_1, \ldots, S_n$ and no other
- their *intersection*, denoted $\cap_{i=1}^{n} S_i$, is the set containing all the elements of S_1 which are elements of $S_2, \ldots, S_n$ and no other

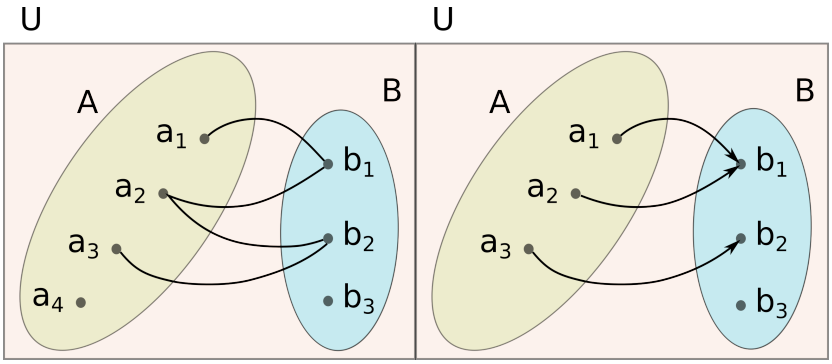


Figura 1.1: Relation (left) and function (right)

Definition 1.1.7 (Set operations on sets of sets)**.** Another way to denote set operation on multiple sets, is to assume S is a *set of sets*; then

- the *union* of the elements (sets) of S, denoted $\bigcup S$, is the set containing all the elements of the elements of S, and no other
- given an element X of S, the *intersection* of the elements of S, denoted $\bigcap S$, is the following subset of X:

  $$\{Y \in X : Y \text{ is an element of all the elements of } S\}$$

  Note this set does not depend on the choice of X .

Definition 1.1.8 (Cartesian product of two set X and Y)**.** Denoted $X \times Y$, is the set containing exactly every ordered pair (x, y) such that $x \in X$ and $y \in Y$

Definition 1.1.9 (Binary relation on X and Y)**.** A subset of $X \times Y$

Definition 1.1.10 (Cartesian product of a sequence of set)**.** If $S_1, \ldots, S_n$ is a set of sets, the cartesian product of the sequence, denoted $S_1 \times \cdots \times S_n$ is the set containing every ordered n-tuple having as i-th component an element of S_i

Remark 4. Writing an ordered n-tuple as $(x_1, \ldots, x_n)$, then $S_1 \times \cdots \times S_n = \{(x_1, \ldots, x_n) : x_i \in S_i, 1 \leq i \leq n\}$.

Remark 5. Function are special cases of binary relations

Definition 1.1.11 (Function)**.** A function on A in B, written $f : A \to B$, is a binary relation on A and B such that for each element $x \in A$ there is one and only one element $y \in B$ such that (x, y) is an element of the relation:

- A is called the function's *domain* and B its *codomain*
- If (x, y) is an element of the function, write $y = f(x)$, and say that y is the *image* of x in f

Capitolo 2

Python

2.1 Algorithms

2.1.1 Executors, variables, expressions, programming constructs

Remark 6. The idea of executing an algorithm, that is, following the step-by-step directions of a "recipe" to accomplish a task is many millennia old; the term algorithm however derives from the name of the Persian scientist of the eighth and ninth century (Al-Khwarizmi).

Remark 7. Although simple algorithms are hidden in many tasks we perform daily, an intuitive, a naive approach to the description of algorithms for mechanical execution is inadequate

Definition 2.1.1. An algorithm is a *non-ambiguous* and *repeatable* sequence of *instructions* which allows to solve a problem in a general way. The instructions are a *set of elementary operations* which are assumed to be executable by a predefined *executor* (eg human, machine/language etc).

Remark 8 (Executors and algorithms). In the definition of algorithm, a "set of elementary operations" and a "predefined executor" are assumed: if not obvious from the context, in practice one specifies

- an *executor model*: a schematic description of internal *architecture* of the executor and its *components*
- a set of *operations* the machine can execute and how the components interact to compute them
- a *language*, a set of rules to write algorithms that use the machine operations

Generally it's only needed to specify the language, the others are implicit

Important remark 3 (Capabilities of executor models). Although executor models are diversified, *some capabilities are recurrent*

- Numerical variables and expressions, assignment

- Conditionals
- For loops
- While loops
- Function definition and invocation

Definition 2.1.2 (Variables)**.** We assume *a countable set of symbols*, called **variables**.
A variable can be:

- *undefined*: it has no value
- *scalar*: has a numerical or logical value,
- *vector*: its value is a finite sequence. If v is a vector variable and i is a non-negative integer v_i is the i-th component of v: it is a scalar variable having as value the i-th element of the value of v

Definition 2.1.3 (Expression)**.** Expressions are constructed with parentheses, numerical and logical constants, scalar variables and components of vector variables, arithmetic or logical operators, and functions, and follow the usual formation rules of arithmetic and logical expressions

Element	Examples
Costant	`1 3.14 True False`
Variable	`a b x y sum`
Operator	`+ - * / = <` $\leq$
Function	`exp log sin cos`

Definition 2.1.4 (Expression evaluation)**.** During algorithm execution, expressions are *evaluated* (= replaced by a single computed value):

1. every scalar variable is replaced by its value
2. all operations are then computed following arithmetic and logic rules, until the value of the expression is obtained

An expression can be evaluated only if *none* of its variables is undefined

Definition 2.1.5 (Assignment)**.** An assignment is a pair consisting of a scalar variable and an expression, separated by an *assignment symbol* (like `<-` or `=`). The execution of an assignment is a two-step process

1. the expression is evaluated
2. the value of the scalar variable is set to the value of the expression (if it had a value, it is lost)

Definition 2.1.6 (`if` conditional statements)**.** It is `if` followed by an expression e (which evaluates to either true or false) a sequence of instructions, `else` and a sequence of instructions

```
if e instructions else instructions
```

The conditional statement execution is:

1. The expression e is evaluated to a truth value v
2. The first sequence of instructions is executed if and only if v is true; at termination of the last instruction in the sequence, the execution of the conditional statement also terminates (thus step 3. is not executed)
3. The second sequence of instructions is executed

Definition 2.1.7 (`for` loops)**.** Consists of `for`, a scalar variable x, `in`, a vector variable v, and a sequence of instructions

```
for x in v instructions
```

The execution:

1. current position i is set to 1
2. x is assigned to the value of v_i
3. instructions are executed
4. if current position i is the last one, the execution terminates
5. current position i is incremented by one
6. execution continues from 2.

Definition 2.1.8 (While loops)**.** It is `while` and an expression e, which evaluates to either true or false, and a sequence of instructions

```
while e instructions
```

Its execution:

1. expression e is evaluated to a truth value v
2. if v is false execution terminates
3. instructions are executed
4. execution continues from 1.

Definition 2.1.9 (Function definition)**.** It is a function name, followed by an open parenthesis `(`, a comma-separated sequence of parameters, a closed parenthesis `)`, and a sequence of instructions (called the body of the function):

```
my_function(x1, ..., xn) instructions
```

Definition 2.1.10 (Function invocation)**.** It is a function name, followed by an open parenthesis `(`, a comma-separated sequence of expressions, and a closed parenthesis `)`.
The execution:

1. expressions in the sequence are evaluated, yielding a sequence of values
2. elements of the sequence of values are orderly substituted for the elements of the sequence of parameters in the body of the function
3. body of the function is executed

2.1.2 Algorithms

Remark 9 (Problems and algorithms). Problems usually require to construct an "object" that satisfies some desired properties, but it is not apparent how to construct it.
Eg to generate all pairs of non-negative integers having sum k, $A_k = \{(i, j) : i + j = k\}$ we can rewrite as $A_k = \{(i, k - i) : i \in \{0, 1, 2, \ldots, k\}\}$ which is clearly computable using a for loop.
Designing algorithms for interesting problems is well beyond rewriting

Remark 10. In computer science we speak about search problems, in general

Definition 2.1.11 (Search problem)**.** It's a pair consisting of:

Instances are generic cases/set of objects

Solution sets an association of a (possibly empty) set of solutions to each instance

Remark 11. We have that:

- instances are tipically n-tuples of elementary objects
- an algorithm for a problem, given an instance, returns one of the solutions (in the solution set for the instance) or a conventional answer (such as "no") if there is no solution
- the specification of a problem usually consists of an instance and a solution for the instance

Definition 2.1.12 (Decision problem)**.** A problem concerning a question with only two possible answers, "yes" and "no"; thus is called a decision problem.
The decision problem is a pair consisting of:

Instances a set of cases/objects

Yes-instances a *subset* of the instances for which the answer to the question is "yes"

Remark 12. *Decision problems* could also be formulated as a *search problems* with solution sets either "yes" or "no".
The specification of a problem usually consists of an instance and the question defining the yes-instances

Example 2.1.1 (Problem example: Minimum of a sequence)**.** We are given a sequence of elements, indexed $0, 1, \ldots, n - 1$, on which a total order is defined. Which is an index of a minimum element of the sequence?

Instance A sequence $v_0, \ldots, v_{n-1}$ of elements and a total order $\leq$ on them

Solution An index i such that $v_i \leq v_j$ for $j = 0, 1, \ldots, n - 1$

Definition 2.1.13 (Pseudo-code)**.** It in an informal language to describe algorithm which imitates some known programming language without adhering precisely to its syntax:

- the assumed executor model understands numerical variables and expressions, assignments, conditionals, for loops, while loops, function definition and invocation
- basic functions with obvious meaning in the context are often used without definition (eg mean)

Finally, if the programming language that will be used to implement the algorithm is known, it is convenient to express the algorithm as a pseudo-code which imitates that language

Remark 13 (Algorithms in natural language). Algorithms can also be written in natural language, e.g., English, typically when both the writer and the reader are experts of the same field

- the constructs/phrases used to express things like variables, looping and boolean expressions evaluations etc are not standard, are a matter of preference;
- algorithms to solve basic subproblems are omitted (eg. an algorithm in natural language might contain steps like "let m be the minimum of sequence s", "let **p** be the first principal component of data D")

Important remark 4 (Algorithm presentation). The recommended format for algorithm presentation consist of:

- an **Input** section: describes a generic instance
- an **Output** section: descres the corresponding solution
- an **Algorithm** section: lists the lines of code in the chosen style (pseudocode). The sequences of instructions in conditional statements, `for` and `while` loops *must be indented* somehow

```
statement head
  instruction
  instruction
  ...
  instruction
```

Example 2.1.2 (Algorithm example: Minimum of a sequence)**.** we have:

Input non-empty sequence v of elements, occupying positions $0, 1, \ldots$ equipped with an order relation $<$

Output the index of a minimum element of v in the ordering

Algorithm we have:

- assign 0 to $imin$
- assign 0 to i
- while i is less than the length of v minus 1
  - if $v_i < v_{imin}$

        * assign i to $imin$
    - increment i by 1
- return $imin$

Example 2.1.3 (Variation: Minimum with starting index)**.** Solves the problem for indices not smaller than a given s:

Input non-empty sequence v with an order relation $\leq$, starting index $s < length(v)$

Output index of a minimum among the elements in v having index at least s

Algorithm `minvs(v, s)`:

- assign s to $imin$
- assign s to i
- while i is less than the length of v minus 1
    - if $v_i < v_{imin}$
        * assign i to $imin$
    - increment i by 1
- return $imin$

```
>>> def minvs(v, s = 0):    # slightly more general with default find the minimum
...     imin = s
...     i = s
...     while (i < len(v)): # in python non ci deve essere il -1 se no non prende
...         if v[i] < v[imin]: # l'ultimo elemento...
...             imin = i
...         i = i + 1
...     return imin
...
>>> l = [3,1,12,4]
>>> minvs(l)
1
>>> minvs(l, 2)
3
```

Definition 2.1.14 (Sequence sorting problem)**.** Given a sequence of elements for which a total order is defined, find the ordered sequence with the same elements:

Instance a sequence $v_0, v_1, \ldots, v_{n-1}$ of elements and a total order $\leq$ on them

Solution a non-decreasing permutation of $v_0, v_1, \ldots, v_{n-1}$

Remark 14. There are at least 3 simple algorithm which are not very efficient, *selection sort* is one example for them

Example 2.1.4 (Sequence sorting algorithm: selection sort)**.** For increasing values of index i, finds a minimum of the elements at indexes at least i and swaps it with element at i:

Input v a vector

Output v, ordered by magnitude

Algorithm
- for i from 0 to the length of v minus 2
  - assign the index returned by `minvs(v, i)` to j
  - assign v_i to tmp
  - assign v_j to v_i
  - assign tmp to v_j
- return v

```
>>> def selection_sort(v):
...     for i in range(len(v) - 1): # bug again in the pseudocode
...         # j is the index having the minimum
...         j = minvs(v, i)
...         tmp = v[i]
...         v[i] = v[j]
...         v[j] = tmp
...     return v
...
>>> selection_sort(l)
[1, 3, 4, 12]
```

Important remark 5 (Patterns). Once the capabilities of the executor are known, how do we design an algorithm solving a given problem?
There is no universal technique to mechanically create an algorithm, given a problem; however, *frequently recurring patterns* do exist.
Complex algorithms make use of simpler patterns several times, often in non-trivial combinations

Definition 2.1.15 (Pattern: Sequence filter)**.** If s is a sequence, a sequence filter consists in accessing each element of the sequence in order of position, and executing an action with it if a property is satisfied:

- for every e in s,
- if property P is true for e,
- perform action A using element e.

Remark 15 (Sequence filter subcases). One can have

- P is *always true*: A is performed with every element of s, and no conditional statement is needed. Example: round every element to the nearest integer
- *Single action*: A is performed with only the first element of s satisfying P. Example: return -1 if there exists a negative element in a sequence
- *Accumulation*: A updates some fixed variable v using e. Example (sum computation): set v to 0 at the beginning. Action A is the addition of e to v

Example 2.1.5. Respectively

1. write algorithms in natural language for each of the examples above (class)
2. write an algorithm in natural language which returns the largest element of a sequence;
3. write an algorithm in natural language which returns the two largest elements of a sequence. Hint: keep two running largest values and update them by comparison with the current element
4. write an algorithm in natural language which returns the arithmetic mean of a numeric sequence.

Definition 2.1.16 (Pattern: Nested loop)**.** Given two set/sequences A and B and a function $f(\cdot,\cdot)$, this pattern solves $f(A \times B)$ problems (apply a function to all the combination from two sets): the pattern is

- for each $x \in A$
  - for each $y \in B$
    - perform $f(x, y)$

Remark 16. Note that changing the order of the loops only changes the order of the computations of f's values, not the values.
If we represent the xs and ys as rows and columns of a matrix, the order determines whether the matrix is accessed row-wise or column-wise

Remark 17. Simplest applications involve the computation of all distances of a set of k-variate data. Let's do is for $k = 1$ (display the distance for a set of numbers)
If we simply display the distances, we use the pattern in a trivial manner: sequences s and t are the same, and action f prints the function value (distance) for a pair of data.

Example 2.1.6 (Distances in data)**.** Display of the distances of univariate data:

Instance A sequence X of numbers $x_0, x_1, \ldots, x_{n-1}$

Solution A display of $|x_i - x_j|$, for all $i, j \in \{0, 1, \ldots, n-1\}$

Algorithm we have:

- for each $x \in X$
  - for each $y \in X$
    - display $|x - y|$

Remark 18 (Collecting all distances in data). If we must return the whole set of distances, then the action must collect the distances into some kind of container as soon as they are computed (eg matrix of data distances).
We choose a sequence of sequences, which can be read as a sequence of matrix rows, one row for every datum

Definition 2.1.17 (Pattern: collection of distances of univariate data)**.** The algorithm is:

Instance a sequence X of numbers $x_0, x_1, \ldots, x_{n-1}$

Solution a sequence $s_0, s_1, \ldots, s_{n-1}$ of sequences of numbers. $(s_i)_j$ is the distance $|x_i - x_j|$

Algorithm
- assign the empty sequence to M
- for each $x \in X$
  - append the empty sequence to M
  - for each $y \in X$
    - append $|x - y|$ to the last element of M
- return M

2.2 Intro to python

Remark 19. Why Python?

- Readability, coherence, and software quality
- Productivity
- Source size is one-third to one-fifth of equivalent code in comparable high-level languages, like C++ or Java
- Immediate execution. Python programs need not be translated to binary programs in advance
- Portability. Python programs run with no changes on all platforms
- Large collection of libraries
- Commmunication with external programs and systems in a variety of ways

Definition 2.2.1 (Tuples)**.** They are:

- *sequences of expressions* which are *immutable* once assigned;
- comma is the *separator character*: it is
  - legal after the last expression
  - required after an expression to create a singleton tuple.

  If omitted, the result is just the expression alone
- *parentheses* are optional except for the empty tuple

Components of a tuple can be read (not written) by writing the position of the component within square brackets; first position is 0. Furthermore:

- ranges `i:j` of positions are allowed, where component `j` is not retrieved
- negative integers starting with -1 are chosen to start selecting from the end (different from R where negative index excludes)
- omitted left or right component in the range is replaced by the first or last position

```
>>> e1 = (1, 2, 3)
>>> e2 = 1, 2, 3
>>> e3 = 1, 2, 3,
>>> e4 = (1, 2, 3,)
>>> # parenthesis are optional, above expression are equivalent
>>> print(e1 == e2 == e3 == e4)
True

>>> # singleton tuple
>>> e = 1,
>>> print(e)
(1,)

>>> # empty tuple
>>> e = ()
>>> print(e)
()

>>> # Accessing components (read only)
>>> e = (2, 3, 5, 7, 11)
>>> print(e[0]) # first
2
>>> print(e[4]) # last
11
>>> print(e[1:4]) # from the second to fifth (excluded)
(3, 5, 7)
>>> print(e[-2]) # from the end (counting the last as -1)
7
>>> print(e[:4]) # up to the fourth
(2, 3, 5, 7)
```

Definition 2.2.2 (Lists)**.** They are *mutable sequence* created by enumerating its elements within *square brackets*:

- components are accessible (read/write) with square brackets
- further operations are available through methods/function members

```
>>> l = ['a', 'b', 'c', 'd']
>>> print(l[1])
b
>>> print(l[1:3])
['b', 'c']
>>> # modifying
>>> l[3] = 'f'

>>> # using methods
>>> l.extend(('x', 'y')) # append
>>> print(l)
['a', 'b', 'c', 'f', 'x', 'y']
>>> l.remove('c') # remove a matched item
```

```
>>> print(l)
['a', 'b', 'f', 'x', 'y']
```

Definition 2.2.3 (`if`, `for` and `while`)**.** We have that

- All must be written with a colon `:` at the end of the first line.
- The sequence of instructions, is usually written on a new line, and all instructions must be indented by the same amount

```
>>> e = (2, 3, 5, 7, 11)

>>> # print the odd elements of e
>>> # % is the modulo op
>>> for x in e:
...     if x % 2 == 1:
...         print(x)
...
3
5
7
11
>>> # moving average with size 2 window
>>> i = 1
>>> while i < len(e):
...     print((e[i-1]+e[i])/2)
...     i = i + 1
...
2.5
4.0
6.0
9.0
```

Definition 2.2.4 (Functions)**.** We have:

- definitions must be introduced by `def` keyword
- first line ends with a colon.
- function body follows the same rules as the instructions in loops.
- value of the function is assigned by a `return` statement (different from R which returns the last evaluated expression)

```
>>> # definition
>>> def ma2(v):
...     res = []
...     i = 1
...     while i < len(e):
...         ma = (e[i-1]+e[i])/2
...         res.append(ma)
...         i = i + 1
...         return res
```

```
...
>>> # invocation
>>> seq = (2, 3, 5, 7, 11)
>>> res = ma2(seq)
>>> print(res)
[2.5]
```

Example 2.2.1. Write Python programs:

- implementing the algorithms designed in Assignments 1-4
- implementing the Minimum of a sequence algorithms. After writing the second variant, modify the first one so that it uses the second one
- implementing the Selection sort algorithm

```
>>> # print the square of some elements
>>> # --------------------------------
>>> for x in (1, 2, 3):
...     print(x ** 2)
...
1
4
9
>>> # print the square of the following vectors
>>> # ----------------------------------------
>>> t = (1, 2, 3)
>>> t1 = (10, 20, 30)

>>> # we need a list of indexes (of same legth of t)
>>> list(range(len(t)))
[0, 1, 2]

>>> # using index to print the squares of both vectors
>>> for i in range(len(t)):
...     print(t[i]  ** 2)    # questa è la suite
...     print(t1[i] ** 2)    # è il modo di dire block of code
...
1
100
4
400
9
900
>>> # stessa stampa dei quadrati mediante l'uso di while
>>> i = 0
>>> while i < len(t):
...     print(t[i] ** 2)
...     print(t1[i] ** 2)
...     i = i + 1
...
```

```
1
100
4
400
9
900
>>> # sum of the elements of t
>>> # ------------------------
>>> s = 0       # initialisation
>>> for x in t:
...     s = s + x
...
>>> # create a list containing the squares of all elements
>>> # of t, preserving order
>>> # ------------------------------------------------------
>>> sql = []  # initialise
>>> for x in t:
...     sql.append(x ** 2)
...

>>> # euclidean distance of two vectors v and w
>>> # -----------------------------------------
>>> v = (1.0, 0.0, -3.0)
>>> w = (1.0, 2.0, 1.0)
>>> s = 0
>>> for i in range(len(v)):
...     s = s + (v[i] - w[i])**2
...
>>> print(s)
20.0

>>> # encapsulate it in a function
>>> # ----------------------------
>>> def sqnorm(x, y):
...     s = 0
...     for i in range(len(x)):
...         s = s + (x[i] - y[i])**2
...     return s # here is different from R, we have to return the value
...
>>> sqnorm(v, w)
20.0

>>> # Implement a function returning a list containing the sums of the
>>> # squares of the differences between elements of vectors v and w, for
>>> # all pairs of vectors v, w in a list of vectors lv (all vectors in lv
>>> # have equal length).

>>> lv = [(1.0, -1.0, 1.0), (0.0, 0.0, 0.0), (1.0, 2.0, -3.0)]

>>> def sqnorm_all(l):
```

```
...     res = []
...     # if we want to compute something for a cartesian product of a
...     # set, the patter is to use a for inside a for, or more generally
...     # a repetition inside a repetion
...     for x in l:
...         for y in l:
...             res.append(sqnorm(x, y))
...     return res
...
>>> sqnorm_all(lv)
[0.0, 3.0, 25.0, 3.0, 0.0, 14.0, 25.0, 14.0, 0.0]

>>> # second version which does not compare a vector with itself
>>> def sqnorm_all2(l):
...     res = []
...     # if we want to compute something for a cartesian product of a
...     # set, the patter is to use a for inside a for, or more generally
...     # a repetition inside a repetion
...     for i in range(len(l)):
...         for j in range(len(l)):
...             if i != j:
...                 res.append(sqnorm(l[i], l[j]))
...     return res
...
>>> sqnorm_all2(lv)
[3.0, 25.0, 3.0, 14.0, 25.0, 14.0]

>>> # Find the position of a minimum element in a tuple t.
>>> # Write a function implementing the search algorithm.

>>> def smin(t):
...     the_min = t[0]
...     pos = 0
...     for i in range(len(t)):
...         if t[i] < the_min:
...             the_min = t[i]
...             pos = i
...     return pos
...
>>> t2 = (1.0, 2.0, -3.0)
>>> smin(t2)
2
```

2.3 Grammar

Important remark 6 (Syntactical correctness). A fundamental part of the design of a programming language is the definition of the sequences of characters which are syntactically correct programs
The definition is based on the notion of *grammar*, which is a general mechanism

to generate syntactically correct sentences in a formal language.
A sequence of characters is syntactically correct in a formal language if and only if it can be generated by its grammar

Definition 2.3.1 (Grammar)**.** It is a set of production rules, a rule of substitution, of the form

```
S ::= s1 | s2 | ... | sn
```

where:

- the application of the rule replaces the left-hand side symbol `S` with one of the alternative sequences `si` at the right-hand side (alternatives are separated by |)
- The `si` may contain both symbols and characters

Important remark 7 (Python's grammar notation)*.* It has its own Backus naur shit with the following characteristics:

- `[s]`: s is optional. Thus `s'[s]` is equivalent to `s' | s's`
- `s *` : s is optional and repeatable (0-n). `*` binds as tightly as possible: `s's*` means that `s` is optional and repeatable, not `s's`
- `s +`: `s` is required and repeatable. Binding as `*`
- `s1 | s2`: `s1` and `s2` are mutually exclusive, but one is required
- `"s"`: `s` must be written as is, character by character
- `( )`: parentheses are used for grouping. For example, `("a" "b")+` can be replaced by any sequence of `ab`
- `::=`: Rule definition

Example 2.3.1. In the following, we will use Python's grammar style, but we will keep writing non-terminal symbols in `< >` for clarity Example: decimal numbers in Python's grammar style

```
<number>         ::= <integer_part> |  <integer_part> "." <decimal_part> | "." <decimal_part>
<integer_part>   ::= <non_zero_digit><digit>*
<decimal_part>   ::= <digit>+
<digit>          ::= "0" | <non_zero_digit>
<non_zero_digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Important remark 8 (Elements of python grammar)*.* There are three basic elements in the grammar of Python:

- *Expressions*: Simple and composite data, such as numbers, strings of characters, tuples, lists, sets, and associations of keys to values
- *Simple statements*: Statements that can be contained in a single logical line (eg assignment, return, importing libraries)
- *Compound statements*: Statements containing sequences of instructions, like `if`, `for`, `while`, function definitions

2.3.1 Expression in python

2.3.1.1 Atoms

Atoms are basic blocks of expressions

```
<atom> ::=  <identifier> | <literal> | <enclosure>
<enclosure> ::= <parenth_form> | <list_display>
                | <dict_display> | <set_display>
                | <generator_expression> | <yield_atom>
```

where

- `identifiers` are names we use for objects. Allowed characters are both lowercase and uppercase letters, the underscore _, and the decimal digits, except for the first character;
- `literals` are constant data, such as floats and strings
- `enclosure` are forms requiring some type of brackets and stand for compound objects of various kinds (eg lists, tuples are enclosures)

2.3.1.2 Literals

Literals denote strings of characters, bytes, integers, floating point and imaginary numbers. We omit their grammar and mention notable cases

```
<literal> ::= <stringliteral> | <bytesliteral>
              | <integer> | <floatnumber> | <imagnumber>
```

where

- Integers follow standard practice, with binary and hex prefixed by 0b and 0x
- Floats admit exponential notation, like 2.5e-10
- String literals are enclosed by single ' ' or double quotes " ", and cannot contain \, a newline, or the quote. Triple single or double quotes create strings which can contain a newline, or the quote, but not \

2.3.1.3 Strings and quotes

Quotes, be they single or double, mark both the beginning and the end. Thus a quote cannot occur between a pair of quotes of the same type. If it does, it marks the end, and what remains is incorrect

```
>>> my_wrong_string = 'It's wrong'
  File "<stdin>", line 1
    my_wrong_string = 'It's wrong'
                             ^
SyntaxError: unterminated string literal (detected at line 1)
```

However, we can use a different quote

```
>>> my_correct_string = "That's ok!"
>>> print(my_correct_string)
That's ok!
```

Newlines cannot occur in strings unless they are delimited by triple quotes. They can be inserted in strings as an *escape sequence* `\n`

```
>>> my_long_string = """Hello.....................
... Hmm, I'd rather type on a new line"""
>>> print(my_long_string)
Hello.....................
Hmm, I'd rather type on a new line
>>> my_short_string = "Now, this is tricky ... \nDone!"
>>> print(my_short_string)
Now, this is tricky ...
Done!
```

2.3.1.4 Identifiers and objects

In most languages variables are modeled as named boxes, which contain the value of the variable (eg R):

- if one assign to a variable x creates a box with it content
- if one assigns again to variable x, it replaces the content

In Python, there is a deeper distinction between the *identifier* and its *associated value*. eg considering a literals:

- whenever a literal is used, an *object* is created for it, with an *identity* (represented by a unique integer which does not change during the lifetime of the object);
- the identifier is associated to the object's identity in an internal look-up table. We say the identifier is a *reference* to the object
- Assigning to an identifier `y` using `x` as right-hand side expression makes `y` *reference the same object* as x

Example 2.3.2. In the following example, when it comes to literals, assigning to x a second time does not affect y in any way: a new object is created and x is set to reference it. From a user's viewpoint, this behaviour is not different from that of the variable-value model (in fig 2.1).

```
>>> x = 1
>>> y = x
>>> print("x:", x, "id:", id(x), "y:", y, "id:", id(y))
x: 1 id: 11817000 y: 1 id: 11817000
>>> x = 2
>>> print("x:", x, "id:", id(x), "y:", y, "id:", id(y))
x: 2 id: 11817032 y: 1 id: 11817000
```

However, *differences between models emerge in assignments to mutables.* Assign any non-empty list `mylist` to `x`, then assign `x` to `y`. Finally, assign any value not in mylist to `x[0]`. What happened to y? Explain it with a diagram, like .

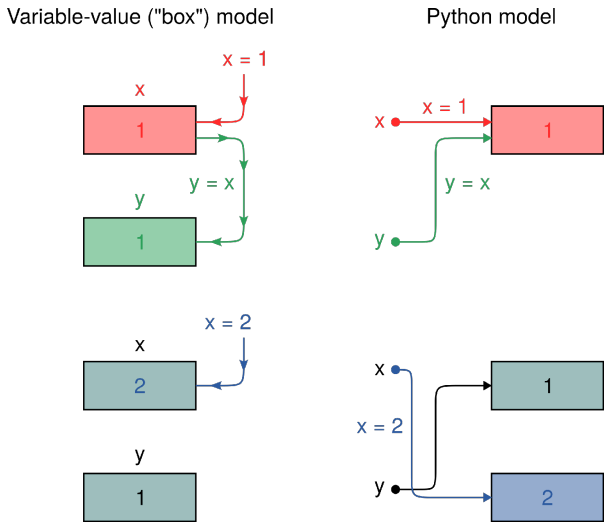


Figura 2.1: Variable-value (box) model on left, python model on the right

```
>>> my_list = [1,2,3,4]
>>> my_second_list = my_list
>>> my_list[0] = 2
>>> print(my_second_list)
[2, 2, 3, 4]
>>> # they both changes

>>> # same things happens if we change my_second_list
>>> my_second_list[0] = -1
>>> print(my_list)
[-1, 2, 3, 4]
```

When working with this kind of data no copies are created, we set/copy just a reference to the same object in memory (a list of length 4)

```
>>> # instead this way we copy the data to a new reference, not the
>>> # reference to old data
>>> my_third_list = my_list.copy()
>>> my_list[0] = 9
>>> print(my_third_list) # no changes as expectex
[-1, 2, 3, 4]
```

2.3.1.5 Enclosures

They include *tuples* (<parenth_form>), *dictionaries*, *lists*, and *sets*:

- A **dictionary** is a collection of mappings between a key and a value, which allow to retrieve the value, given the key.

  ```
  >>> d = {'Gauss': 1777, 'Fermat': 1770, 'Ramanujan': 1887}
  >>> print(d['Fermat'])
  1770
  ```

- A **set** represents the standard notion of finite set of set theory, thus with no duplicate elements (and no order)

```
>>> s = {'Gauss', 'Fermat', 'Ramanujan'}
>>> print(s)
{'Gauss', 'Ramanujan', 'Fermat'}
```

2.3.1.6 Sequences, iterables, mappings

The different kinds of enclosures highlight the difference between a sequence and an iterable.

- **Iterable**: any object which can be iterated over by picking its members one at a time in some order. Iterables can be used in `for` loops as a source of objects for their identifier.
  Eg: *Tuples*, *dictionaries*, *lists*, *sets* but even strings of characters and file objects are iterables.

- **Sequence**: an *iterable* which allows the retrieval of its elements through an integer *index*. It thus extends the concept of mathematical finite sequence to any type of elements. Eg: *Tuples* and *lists* are sequences

- **Mapping**: a container object which supports lookup by key.
  Eg: *dictionaries* are mappings

2.3.1.7 Starred lists: Examples

A function of *two arguments* cannot be invoked with a tuple or list of two elements as argument. Being only one object, that would be intepreted as a single argument, raising an error. However, the contained objects can be unpacked in place using a star operator

```
>>> import math
>>> def euclidean(x, y):
...     dist = 0
...     for i in range(len(x)):
...         dist += (x[i]-y[i])**2
...     return math.sqrt(dist)
...
>>> data = ((0, 0, 0), (1, 1, 1))
>>> print(euclidean(*data))
1.7320508075688772

>>> transpose = zip(*data)
>>> for e in transpose:
...     print(e)
...
(0, 1)
(0, 1)
(0, 1)
```

2.3.1.8 List displays

With dictionary displays, list displays, and set displays, objects are constructed in two ways:

- explicit elements
- *Comprehension*, a computation that performs repetitions and selections

Definition 2.3.2 (List display)**.** The syntax is:

```
<list_display> ::= "[" [ <starred_list> | <comprehension> ] "]"
```

Example 2.3.3. An example:

```
>>> greek_mathematicians = ['Euclid', 'Archimedes']
>>> italian_mathematicians = ['Fibonacci', 'Cardano']
>>> # putting * davanti in a list splits
>>> some_mathematicians = [*greek_mathematicians, *italian_mathematicians, 'Euler']
>>> print(some_mathematicians)
['Euclid', 'Archimedes', 'Fibonacci', 'Cardano', 'Euler']
```

2.3.1.9 Comprehension in lists

Definition 2.3.3 (List comprehension)**.** The syntax is:

```
<comprehension> ::= <assignment_expression> <comp_for>
<comp_for>      ::=  ["async"] "for" <target_list> "in"  <or_test> [<comp_iter>]
<comp_iter>     ::=  <comp_for> | <comp_if>
<comp_if>       ::=  "if" <or_test> [<comp_iter>]
```

Example 2.3.4. An example (iterable `range`(n) returns the first n integers)

```
>>> # squares of the odd numbers using if
>>> print([i**2 for i in range(10) if i%2 == 1])
[1, 9, 25, 49, 81]
>>> # double for in comprehension: using additional [] create a matrix
>>> # eg to create two dimensional structures
>>> print([[(i, j) for j in range(3)] for i in range(2)])
[[(0, 0), (0, 1), (0, 2)], [(1, 0), (1, 1), (1, 2)]]
```

2.3.1.10 Dictionary displays

Definition 2.3.4 (Dictionary display)**.** Comma-separated collection in braces of colon-separated key/value pairs. The syntax is

```
<dict_display>       ::=  "{" [<key_datum_list> | <dict_comprehension>] "}"
<key_datum_list>     ::= <key_datum> ("," <key_datum>)* [","]
<key_datum>          ::= <expression> ":" <expression> | "**"  <or_expr>
<dict_comprehension> ::= <expression> ":" <expression> <comp_for>
```

A few rules:

- objects of all immutable types can be used as keys (left side of `:`): eg we can use tuples as key

- a key is evaluated before its value
- evaluation of the key-value pairs for insertion into the mapping is left-to-right. Thus, if a key occurs more than once, the last mapping for the key is the final one

Example 2.3.5 (Dictionary displays)**.** An example:

```
>>> # using a tuple as key
>>> distances = {("Bologna", "Rimini"): 109,
...              ("Bologna", "Ravenna"): 69,
...              ("Ravenna", "Rimini"): 50}
>>> print(distances[("Bologna", "Rimini")])
109

>>> # Iterating over the dictionary is iterating over its keys
>>> for e in distances:
...     print(e)
...
('Bologna', 'Rimini')
('Bologna', 'Ravenna')
('Ravenna', 'Rimini')
>>> # Dictionary keys and values are collected by member functions of
>>> # the object. The returned objects are iterable

>>> keys = distances.keys()
>>> values = distances.values()
>>> for k, v in zip(keys, values):
...     print("Key", k, "has value", v)
...
Key ('Bologna', 'Rimini') has value 109
Key ('Bologna', 'Ravenna') has value 69
Key ('Ravenna', 'Rimini') has value 50

>>> # dict can be constructed using iterables as well
>>> print(dict((('a', 0), ('b', 1), ('a', 2)))) # only the last kept
{'a': 2, 'b': 1}
```

2.3.1.11 Set displays

Definition 2.3.5. Comma-separated collection, or comprehension, in braces, representing a set:

```
<set_display> ::= "{" (<starred_list> | <comprehension>) "}"
```

with the following specs:

- the order of the elements is undefined
- when written as a list, elements are evaluated from left to right
- when written as a comprehension, the elements returned by the comprehension are used to construct the set. Thus, duplicate elements are eliminated

Example 2.3.6. An example:

```
>>> physicists = {"Newton", "Einstein", "Planck"}
>>> print(physicists)
{'Einstein', 'Planck', 'Newton'}

>>> # set can be constructed from iterable as well using set()
>>> print(set((1, 2, 2, 3)))
{1, 2, 3}
```

2.3.1.12 Empty set and empty dictionary

The empty set is constructed by `set()`, whereas empty dictionary by `dict()` or `{}`.

```
>>> empty_set = set()
>>> empty_dict = {}
>>> print(type(empty_set))
<class 'set'>
>>> print(type(empty_dict))
<class 'dict'>
```

2.3.1.13 Parenthesized form

Definition 2.3.6 (Parenthesized form)**.** It's an optional comma-separated list of expressions in parentheses.
The list of expressions, `starred_expression`, is either the most general form of expression, or a list of possibly starred items

```
<parenth_form> ::= "(" [<starred_expression>] ")"
<starred_expression> ::=  <expression> |  (<starred_item> ",")* [<starred_item>]
```

The evaluation of a parenthesized form evaluates the enclosed starred expression, and does not introduce any additional computation: it yields what the enclosed starred expression yields.
Parenthesized forms group operations, thus potentially changing their evaluation order. They do not create tuples. If the enclosed expression is a comma-separated list, it yields a tuple, otherwise it yields a single object

2.3.1.14 Expression lists

```
<expression_list> ::= <expression> ("," <expression>)* [","]
```

- a list of comma-separated expressions is called a tuple
- enclosing parentheses (round brackets) are optional, except for the empty tuple `()`
- Expressions could require evaluation, for example they could be identifiers or function invocations. Evaluation of the expressions in the list is computed from left to right. Thus, any changes made to an object by an expression might affect the evaluation of the following ones

- The trailing comma is optional, except for the singleton tuple. Parentheses enclosing a single expression are not a tuple: the evaluation returns just the evaluated expression

2.3.1.15 Bitwise operations

For integers, Python supports *bitwise operations*, i. e., defined by direct manipulation of the binary representation of an integer.

- **Shift** The binary digits are shifted left or right by a specified amount. Example: `00000101` left shifted by 2 places is `00010100`
- **And, Xor, Or** The logical and, xor, or operations are computed between bits in the same position: Examples.

```
10110110 and 11111000 = 10110000.
10110110 xor 11111000 = 01001110.
10110110 or 11111000 = 11111110
```

2.3.1.16 Or expressions

The bitwise operators are

```
<< >> & ^ |
```

Syntax:

```
<shift_expr> ::= <a_expr> | <shift_expr> ("<<" | ">>") <a_expr>
<and_expr>   ::=  <shift_expr> | <and_expr> "&" <shift_expr>
<xor_expr>   ::=  <and_expr> | <xor_expr> "^" <and_expr>
<or_expr>    ::=  <xor_expr> | <or_expr> "|" <xor_expr>
```

Example 2.3.7. Some examples

```
>>> # 0b means base 2
>>> byte_1 = 0b10110110
>>> byte_2 = 0b11111000

>>> # and xor or
>>> print(byte_1 & byte_2)
176
>>> print(byte_1 ^ byte_2)
78
>>> print(byte_1 | byte_2)
254

>>> # shift
>>> print(byte_2 << 3)
1984
>>> print(byte_2 >> 2)
62
```

2.3.1.17 Arithmetic expressions

The base case of or expressions are arithmetic expressions, `a_expr`, constructed with binary operators, or unary expression `u_expr` constructed with unary operators

```
<m_expr> ::= <u_expr> | <m_expr> "*" <u_expr> |
             <m_expr> "@" <m_expr> |
             <m_expr> "//" <u_expr> | <m_expr> "/" <u_expr> |
             <m_expr> "%" <u_expr>
<a_expr> ::= <m_expr> | <a_expr> "+" <m_expr> |
             <a_expr> "-" m_expr
```

note:

```
// is the floor division
% is the modulo operators
@ is matrix multiplication, which is not currently implemented
```

```
>>> res = 7.5 // 3
>>> print(res, type(res))
2.0 <class 'float'>

>>> int_res = 7 // 3
>>> print(int_res, type(int_res))
2 <class 'int'>
```

2.3.1.18 Unary expressions

Arithmetic unary operators minus, plus, and power `- + **`, and the bitwise unary negation operator invert `~` create unary expressions

```
<u_expr> ::= <power> | "-" <u_expr> | "+" <u_expr> |  "~" <u_expr>
<power> ::= (<await_expr> | <primary>) ["**" <u_expr>]
```

where:

- `-` evaluates to the inverse element of its argument
- `+` evaluates to what its argument does
- `**` is exponentiation. It binds more tightly than other unary operators at its left, and less tightly at its right. Thus, `-x**2` is `-(x**2)`, whereas `x**-2` is `x**(-2)`. Both integers and floats are supported, with integers returned whenever possible
- `~` is binary negation: it flips all bits in its argument, which must be an integer

2.3.1.19 Primaries

```
<primary> ::=  <atom> | <attributeref> | <subscription> | <slicing> | <call>
<attributeref> ::= <primary> "." <identifier>
<subscription> ::= <primary> "[" <expression_list> "]"
```

we have that:

- An *attribute reference* asks the object obtained by evaluating the primary to produce the attribute named by the identifier
- A *subscription* applies to mappings and sequences. It evaluates the expression list and selects elements from the primary object accordingly. The expression list evaluates to a tuple if it contains at least one comma
- A *slicing* is a colon-separated triple `start:stop:stride` which picks elements starting at position `start`, ending at `stop`-1, where each element is `stride` positions after the last picked element

2.3.1.20 Subscription with a mapping

If the primary object evaluates to a mapping, the expression list must evaluate to one of its keys. Note that its keys might be sequences, in that case the expression list should evaluate to a tuple

Example 2.3.8. The distances dictionary had pairs of strings as keys. We may use tuple-valued expression lists in the subscription, in any of their forms

```
>>> # subscription with tuple-valued expression list
>>> print(distances['Bologna', 'Rimini'],
...       distances['Ravenna', 'Rimini',],
...       distances[('Bologna', 'Ravenna')],
...       distances[('Bologna', 'Rimini',)])
109 50 69 109
```

2.3.1.21 Subscription with a sequence

If the primary object is a sequence, the expression list must evaluate to an integer or a slice object *slice*(`start:stop:step`)
Slice objects are convenient when the slice will be used multiple times. Slicing is a frequently used alternative, especially in scripts.

```
>>> my_slice = slice(1, 7, 2)
>>> li1 = [*range(-4, 4)]
>>> li2 = [*range(0, 8)]
>>> li  = [li1, li2]

>>> print([l[my_slice] for l in li])
[[-3, -1, 1], [1, 3, 5]]
```

2.3.1.22 Slicing

```
<slicing> ::= <primary> "[" <slice_list> "]"
<slice_list> ::= <slice_item> ("," <slice_item>)* [","]
<slice_item> ::= <expression> | <proper_slice>
<proper_slice> ::= [<lower_bound>] ":" [<upper_bound>] [ ":" [<stride>] ]
<lower_bound> ::= <expression>
<upper_bound> ::= <expression>
<stride> :: <expression>
```

```
>>> print(distances['Bologna', 'Rimini'])
109
>>> print([l[1:7:2] for l in li])
[[-3, -1, 1], [1, 3, 5]]
```

2.4 Basics part 2

2.4.0.1 Comparisons

```
<comparison> ::= <or_expr> (<comp_operator> <or_expr>)*
<comp_operator> ::=  ">" | "==" | ">=" | "<=" | "!=" | "<"
                     | "is" ["not"] | ["not"] "in"
```

Customary comparison operators which return one of the truth values `False` or `True`. Chaining comparisons is allowed, for example `a<b<c`. The truth value returned is the logical "and" of the truth values of all the comparisons in the chain.

Note on evaluation. `a<b<c` is equivalent to `a<b and b<c`, but the latter evaluates `b` twice

2.4.0.2 Lexicographic order

Lexicographic order is a generalisation of alphabetical order to sequences built from any finite set which is totally ordered

- Let A be a totally ordered finite set. If $a \in A$ and $a' \in A$, then we write $a < a'$ when a precedes a'
- Sequences α, α' of elements from A are ordered as follows
  - If there exist differing elements in α, α' in the same position, let a, a' be the leftmost such elements. Then $\alpha < \alpha'$ if and only if $a < a'$
  - Otherwise, $\alpha < \alpha'$ if and only if α is shorter than α'

Example. A the alphabet of characters. `variability < variable var < variable`

2.4.0.3 Comparison computation

The computation of a comparison depends on the data type

- *Numeric* comparison agrees with mathematical comparison

```
>>> print(2/3 < 1/2, 10e-3 < 10.0001e-3)
False True
```

- *Strings* are compared lexicographically, using the order of Unicode code points

```
>>> print("Infinitesimal" < "Infinite", "Frucht" < "Früe")
False True
```

- regarding *sequence*:
  - If their types differ, order comparisons raise an error. Equality is false, inequality is true
  - If their types are equal, equality is true if and only if their lengths are equal, and the elements in the same position are equal
  - If their types are equal, and they support order comparisons (<, >, <=, >=), then a comparison has the same result as the comparison of the pair of differing elements in the same position that occurs first from left, or of their lengths if no such pair exists

```
>>> s = (1, 2, 3)
>>> t = [1, 2, 3]
>>> k = [1, 2]
>>> l = [1, 3, 3]
>>> print(s == t, k < t, t < l)
False True True
```

2.4.0.4 Comparison computation: in and is

The:

- `in` operator tests membership. It is computable for sequences, sets, and dictionaries, as well as strings. For the latter two, it tests key membership and substring respectively:

```
>>> d = {'temperature': 29, 'humidity': 75, 'pressure': 100}
>>> s = "machine learning"
>>> print( 'humidity' in d, "learn" in s)
True True
```

- `is` operator tests object identity: x is y evaluates to True if and only if x and y are the same object

```
>>> x = 1e20
>>> y = x
>>> z = 1e20
>>> print(y is x, z is x, id(x), id(y), id(z))
True False 140664038088880 140664038088880 140664038089040
```

2.4.0.5 Boolean operations

Or test expressions compute Boolean operations. They are constructed using comparisons and connectives `and`, `or`, `not`.

```
<or_test> ::= <and_test> | <or_test> "or" <and_test>
<and_test> ::= <not_test> | <and_test> "and" <not_test>
<not_test> ::= <comparison> | "not" <not_test>
```

They are suitable for `if`/`while` instructions:

- when or tests are evaluated, `False`, `None`, numeric zero, and empty strings and containers are interpreted as false. Any other object is interpreted as true
- the evaluation of x `and` y first evaluates x. If it is false, its value is returned. Otherwise, y is evaluated and returned
- the evaluation of x `or` y first evaluates x. If it is true, its value is returned. Otherwise, y is evaluated and returned.

Note that with non boolean stuff `and`/`or` returns one of the two operand objects x, y

```
x and y   y false  y true
x false      x        x
x true       y        y

x or y    y false  y true
x false      y        y
x true       x        x
```

2.4.0.6 Expressions

An expression is either a conditional expression or a lambda expression. The purpose of the former is to choose from two alternatives, using a test expression. The purpose of the latter is to define an anonymous function

```
<conditional_expression> ::=  <or_test> ["if" <or_test> "else" <expression>]
<expression> ::= <conditional_expression> |<lambda_expr>
```

The conditional expression `e1 if t else e0` evaluates `t` first. If t evaluates to a true value (see above), then e1 is evaluated, and the resulting value is the value of the whole conditional expression. Otherwise, e0 is evaluated, and the resulting value is the value of the whole conditional expression

2.4.0.7 Lambda expressions

A lambda expression defines an anonymous function, a nameless function that is completely defined by an expression and parameters

```
 <lambda_expr>::= "lambda" [<parameter_list>] ":" <expression>
```

The lambda expression

`lambda p0, p1, ..., pn : e`

epresents a function that behaves as

```
def some_name(p1, p2, ..., pn):
    return e
```

Example 2.4.1. Passing a function as a parameter is a frequent use case of lambda expressions.
Python's built-in functions which are based on a total ordering of the items, such as min and sorted, have an optional keyword parameter key The key must be a function of one argument. The object it returns is used by the built-in function for comparisons

```
>>> phrases = (("the", "white", "fox"),
...            ("a", "black", "raven"),
...            ("some", "green", "parrot"),
...            ("any", "yellow", "parrot"))

>>> sorted_phrases = sorted(phrases, key=lambda x: (x[2], x[1]))
>>> print(*sorted_phrases, sep='\n')
('the', 'white', 'fox')
('some', 'green', 'parrot')
('any', 'yellow', 'parrot')
('a', 'black', 'raven')
```

2.4.0.8 Compound statements

Compound statements control the execution of groups of other statements they contain. They are

- `if`, `while`, for: control flow
- `try`: exception handlers
- `with`: initialisation and finalisation
- `match`: pattern matching
- `def`: function definition
- `class`: class definition

2.4.0.9 Structure of compound statements

Compound statements are structured into clauses, consisting of a header and a suite

- A header is a unique keyword and a colon. If a statement comprises multiple headers, they must be at the same indentation level
- A suite is any group of statements some clause controls, and can be written in two ways
  - Simple statements on the line of the header, with semicolon as a separating character
  - Statements on subsequent lines, with equal indentation. Compound statements are allowed

2.4.0.10 if statement

Conditional execution of suites, controlled by expressions

```
<if_stmt> ::=  "if" <assignment_expression> ":" <suite>
               ("elif" <assignment_expression> ":" <suite>)*
               ["else" ":" <suite>]
```

The `if` statement:

- evaluates the expressions, from top to bottom, or from left to right, until an expression evaluating to true, as defined for Boolean operations, is found
- The corresponding suite is executed
- No other part of the statement is executed or evaluated

2.4.0.11 while statement

Execution of a suite while an expression is true

```
<while_stmt> ::= "while" <assignment_expression> ":" <suite>
                 ["else" ":" <suite>]
```

The while statement repeats the evaluation of the expression. Immediately after each evaluation, it executes the first suite if the expression evaluated to true, otherwise it executes the suite of the else clause and terminates:

- The execution of a break statement in the first suite terminates the execution of the whole while statement
- The execution of a continue statement in the first suite interrupts the execution of the suite, but not the execution of the while statement, which continues with the evaluation of the expression

2.4.0.12 for statement

Iteration over the elements of an iterable object

```
<for_stmt> ::= "for" <target_list> "in" <expression_list> ":
               <suite>
               ["else" ":" <suite>]
```

The `for` statement evaluates the expression list. The evaluation must result in an iterable object, or a type error is raised. An iterator is created for the result. Then, the for statement repeats the assignment of the item the iterator provides, and the execution of the suite, until the iterator is exhausted. Finally, the suite in the else clause is executed, and the statement is terminated

- `break` operates the same way it does in the `while` statement
- Executing `continue` in the first suite interrupts its execution. The execution of the `for` statement continues with a request to the iterator for its next element

Example 2.4.2.

```
>>> import math
>>> def minkowski(x, y, r):
...     m = 0.
...     for u, v in zip(x, y):
...         m += math.fabs(u - v)**r
...     else: # else clause is executed after the loop reaches its final iteration
...         m = m**(1/r)
...     return m
...
>>> print(minkowski((0, 1), (1, -1), 1))
3.0
>>> print(minkowski((0, 1), (1, -1), 2))
2.23606797749979
>>> print(minkowski((0, 1), (1, -1), 3))
2.080083823051904
```

for: target list assignments Objects obtained from the iterator are assigned to the identifiers in the target list, thereby erasing their references, including those that resulted from assignments in the suite

```
>>> for i in range(2):
...     print("Before assignment:", i)
...     i = 0
...     print("After assignment: ", i)
```

for: target list Identifiers in the target list are preserved after the for statement has terminated
Note, however, that an empty sequence expression results in no assignments to the target list, and its identifiers being undefined. Thus, using them raises a `NameError`

```
...
Before assignment: 0
After assignment:  0
Before assignment: 1
After assignment:  0
>>> for i in range(0):
...     ()
...
>>> print(i)
0

>>> for i in range(3):
...     ()
...
()
()
()
>>> print(i)
2
```

2.4.0.13 Exceptions

An exception is an anomalous condition occurring during program execution. An exception is handled in the program by an exception handler

Example 2.4.3. A program sends hourly reports about data it receives through a data stream transmitted over a network connection. The reports include moments of the data, for example mean and variance. If the link is down, or transmits corrupted data, moments are not computable because the observation count is 0, or observations do not satisfy preconditions of the report generating function, which fails and raises the exception. The exception handler acknowledges an exception has occurred in the program section it monitors, and executes statements to handle it and possibly clean up the program's data

Exception handling is implemented in Python by the `try` statement

Example 2.4.4.

```
while process_stream():
    get(stream, new_data) # receive new observations through network
    append_to(data)   # append to list
    if report_due():  # report deadline?
        try:
            gen_report(data) # generate report
        except Exception: # on failure
            forward(data) # send data for inspection
        finally:
            data = []  # clean-up by emptying data
```

2.4.0.14 try statement

The try statement specifies exception handler statements and cleanup statements for a suite

```
<try_stmt> ::= <try1_stmt> | <try2_stmt>
<try1_stmt> ::= "try" ":" <suite>
                ("except" [<expression>
                           ["as" <identifier>]] ":"
                 <suite>)+
                ["else" ":" <suite>]
                ["finally" ":" <suite>]
<try2_stmt> ::= "try" ":" <suite>
                "finally" ":" <suite>
```

How does it work:

- If an exception occurs in the try clause, the expressions in the except clauses are searched for a match to the exception. When a match is found, the exception is assigned to the target identifier of the clause that matched, and its suite is then executed.
- If no exception occurs in the suite of the try clause, and no return, continue, or break statement was executed, the suite in the optional else clause is executed

- The optional finally clause is executed.

Remark 20. When the try statement is in a function suite, and a return statement is executed, the finally suite is executed regardless. Similarly, when try is in a while suite, or for suite, and break or continue are executed

Example 2.4.5.

```
>>> try:
...     x = 1/0 # division by zero
...     # We'll never reach this point
...     print("Past a division by 0.")
... except ZeroDivisionError as zero_div:
...     print("**** " + str(zero_div))
... finally:
...     print("Cleanup: resetting to 0.")
...     x = 0
...
**** division by zero
Cleanup: resetting to 0.
```

2.4.0.15 Iterators

Iterators are objects which represent streams of data. Iterators are fundamental components underlying Python's mechanism to iterate over iterable objects.

- Member function `__iter__()` creates an iterator from an iterable

  ```
  s = {"an", "awesome", "program"}
  i_s = s.__iter__()
  ```

  An iterator

- remembers the last item which was retrieved from the iterable
- exposes a member function `__next__()` which returns the next, i. e., first unretrieved, item
- Raises an exception `StopException` when `__next__()` is called after the last item has been retrieved

  ```
  print(i_s.__next__(),i_s.__next__(), i_s.__next__())
  ```

  Iterator exhaustion: When *StopException* has been raised, no more items can be retrieved from the iterator. Any subsequent call to `__next__()` returns the exception. The iterator is then said to be **exhausted**

  - exhaustion does not prevent multiple usage of a container, such as a list, because every time `__iter__()` is called is, a fresh iterator is created
  - iterators must contain a `__iter__()` member function. Thus, technically they are iterables as well. Calling `__iter__()` returns the iterator itself, however, so that exhaustion cannot be circumvented

2.5 Translation of algorithms

Remark 21. In the exam we have to translate an algorithm in python. An algorithm is described as a sequence of instructions, in the customary dash plus text format, like

```
- for e in the sequence of elements s
  - do something with e
```

with the following specs:

- Indentation is used as in python;
- Every instruction is one of the 5 basic capabilities for algorithm executors (assignment, for, while, if, function def and invocation);
- algorithm is generally self-contained: there is a function definition in the paper for every invoked function. Occasionally, library functions will be invoked, but clearly specified.

Example 2.5.1 (Exam paper 1: 2019-01-08)**.** Let the uniform kernel sum of a p-dimensional point $x \in \mathbb{R}_p$ with respect to a finite set $D \subset \mathbb{R}^p$ and a radius r be the number of elements of D at Euclidean distance at most $r > 0$ from x.
Assume `data` is a list of tuples or lists of fixed length, and implement in Python a function `top_sum(data, r)` which outputs all objects of data having the largest uniform kernel sum with respect to `data` and `r`.
The function can be computed by the following algorithm:

```
- initialise top to an empty sequence
- initialise max_sum to one
- for all x in data
  - set s to the uniform kernel sum of x with respect
    to data and r
  - if s equals max_sum
    - append x to top
    else
    - if s is strictly greater than max_sum
      - set max_sum to s
      - set top to a sequence containing only x
- return top
```

The uniform kernel sum of x with respect to data and r can be computed by the following algorithm:

```
- set c to zero
- for all y in data
  - if the distance between x and y does not exceed r
    - increment c by one
- return c
```

To compute the euclidean squared distance of two objects use the following Python functions

```
>>> import math
>>> def sq_sum(x):
...     return sum((y ** 2 for y in x))
...
>>> def eucl(x, y):
...     return math.sqrt(sq_sum((z[0] - z[1] for z in zip(x, y))))
...
```

The two function are

```
>>> def unif_kern_sum(x, data, r):
...     c = 0
...     for y in data:
...         if eucl(x, y) <= r:
...             c += 1
...     return c
...

>>> def top_sum(data, r):
...     top = []
...     max_sum = 1
...     for x in data:
...         s = unif_kern_sum(x, data, r)
...         if s == max_sum:
...             top.append(x)
...         elif s > max_sum:
...             max_sum = s
...             top = [x]
...     return top
...
```

Example 2.5.2 (Exam paper 2: 2018-12-19)**.** Implement in Python a function `clustroid(data)` which outputs all clustroids of data; a *clustroid* is any observation in data having the smallest *RowSum*, where *RowSum* is the sum of the squared distances between the observation and the remaining ones in data. Assume `data` is a list of tuples or lists of fixed length. The minimum RowSum can be computed by the following nested-loop algorithm:

```
- set min_rowsum to infinity
- for x in data
  - set rowsum to zero
  - for y in data
    - add the squared distance between x and y to rowsum
  - if rowsum is smaller than min_rowsum
    - set min_rowsum to rowsum
- return min_rowsum
```

Then, the clustroids can be computed by the following algorithm, which assumes the minimum RowSum is stored in object `min_rowsum`:

```
- initialise clustroids to the empty list
- for x in data
```

```
  - set rowsum to zero
  - for y in data
    - add the squared distance between x and y to rowsum
    - if rowsum is greater than min_rowsum
      - stop iterations
  - if rowsum equals min_rowsum
    - append x to clustroids
- return clustroids
```

The object `math.inf` from package `math` represent infinity in Python. To compute the euclidean squared distance of two objects use the following Python functions

```
>>> def sq_sum(x):
...     return sum((y**2 for y in x))
...
>>> def sq_norm_diff(x, y):
...     return sq_sum((z[0] - z[1] for z in zip(x, y)))
...
```

we have

```
>>> def minrowsum(data):
...     min_rowsum = math.inf
...     for x in data:
...         rowsum = 0
...         for y in data:
...             rowsum += sq_norm_diff(x, y)
...         if rowsum < min_rowsum:
...             min_rowsum = rowsum
...     return(min_rowsum)
...

>>> def clustroid(data):
...     min_rowsum = minrowsum(data)
...     clustroids = []
...     for x in data:
...         rowsum = 0
...         for y in data:
...             rowsum += sq_norm_diff(x, y)
...             if rowsum > min_rowsum:
...                 break
...         if rowsum == min_rowsum:
...             clustroids.append(x)
...     return(clustroids)
...
```

Example 2.5.3 (Exam paper 3: 2019-01-29)**.** The *condensed nearest neighbour* rule is a classification rule which assigns to a new observation x the class of the nearest neighbour of x in a subset of the set of examples, called a consistent subset. Let data be a set of classified examples in which the last variable is the class variable, stored as a list of tuples or lists of fixed length. Implement

a Python function `cnn_gen(data)` which outputs a consistent subset of data. The function can be computed by the following algorithm:

```
- initialise n to the number of elements in data
- initialise ndim to the number of variables,
  class variable excluded
- initialise T as a list containing the first example of data
- initialise a boolean object diff_found to true
- as long as diff_found is true
  - set diff_found to false
  - for i in 0,..., n-1
    - apply the nearest neighbour rule to the i-th
      element of data with respect to T and store the
      predicted class into predicted_class
    - if the predicted class differs from the class of
      the i-th element of data
      - add that element to T
      - set diff_found to true
- return T
```

Auxiliary Python functions are written below:

```
>>> def sq_sum(x):
...     return sum((y ** 2 for y in x))
...
>>> def sq_norm_diff(x, y):
...     return sq_sum((z[0] - z[1] for z in zip(x, y)))
...
>>> def nn(objects, x):
...     sq_distances = enumerate([sq_norm_diff(x[:-1], y[:-1]) for y in objects])
...     # this is a jargon to remind, if we want to find minimum in
...     # structured data such as list of list
...     i, d = min(sq_distances, key = lambda t: t[1])
...     return i
...
>>> def nn_rule(x, data):
...     ndim = len(data[0])-1
...     inn = nn(data, x)
...     return data[inn][ndim]
...
```

And finally we have

```
>>> def cnn_gen(data):
...     n = len(data)
...     ndim = len(data[0])-1
...     # 1st pattern to T
...     T = [data[0]]
...     diff_found = True
...     while diff_found:
...         diff_found = False
...         for i in range(n):
```

```
...             predicted_class = nn_rule(data[i], T)
...             if predicted_class != data[i][ndim]:    # not class'd correctly, add
...                 T.append(data[i])
...                 diff_found = True
...     return T
...
```

If we want to test

```
import matplotlib.pyplot as plt
import numpy as np

# reading csv with numpy
fname = 'spline_boundary.csv'
# # data layout
# "x","y","class.id"
# 0.955350372474641,0.772700337693095,1
# 0.651135042309761,0.132741579087451,0
# 0.7030201877933,0.0620764032937586,0
# 0.946504527237266,0.659262671368197,1
# 0.393916479311883,0.452305789105594,1
data = np.loadtxt(fname, delimiter=",", skiprows=1)
# plotting the data
fig, ax = plt.subplots()
# here we use a feature of numpy, data is a numpy array of dim 2 (matrix)
# we check if the last column of data is 0 or 1 and make a logical array
# it's a vector comparison returned as vector array
mask1 = data[:, 2] == 0    # 1st class dummy
mask2 = data[:, 2] == 1    # 2nd class dummy
# plot the two subsets (x and y) with different colors
ax.scatter(data[mask1, 0], data[mask1, 1], c = "black") # plot all obs with class 0
ax.scatter(data[mask2, 0], data[mask2, 1], c = "blue") # plot all obs with class 1

# obtaining the stuff from the data
cond_examples = cnn_gen(data.tolist())
cond_data = np.array(cond_examples)
mask1 = cond_data[:, 2] == 0    # 1st class
mask2 = cond_data[:, 2] == 1    # 2nd class
ax.scatter(cond_data[mask1, 0], cond_data[mask1, 1], c="green") # subset of 0 correct
ax.scatter(cond_data[mask2, 0], cond_data[mask2, 1], c="red")
plt.show()
```

Example 2.5.4 (Exam 2023-01-10)**.** A csv file contains rows consisting of 4 elements: a unique row number, a x-coordinate, a y-coordinate, and a string which is a class label.
Write a python function to process the file called `accumulation(file_name)` that returns a dictionary such that:

```
- the keys are the class labels
- for each key k, its value is a tuple of 4 elements
  - the first element is the count of rows having the key k as the
```

```
    first element
  - the second and third element are the sums of the x and y coordinates of
    row having k as their first element
  - the fourth element is the sum of all squares of x and y coordinates of
    rows having k as their first element
```

The function uses the following function which loads a csv file into a list of lists.

```
import csv
def load(file_name):
    data = []
    with open(file_name, 'r') as f:
        reader = csv.reader(f, delimiter = ",")
        for row in reader:
            # do the import/coercion for each row
            data.append([
                int(row[0]), # first column is an int
                float(row[1]), # second and third are floats
                float(row[2]), # second and third are floats
                row[3] # last is a string
            ])
    return data
```

The accumulation function algorithm will be:

```
- set data to the list of lists returend by the call
  load(filename)
- set acc to empty dictionary
- for each element v in data
  - set k to the last element of v (its class label)
  - if k is not a key of acc
    - create a new entry in dictionary acc, having k as its key and
      the list consisting of 1 the x-coordinate and the y-coordinate of v,
      and the sum  of the squared x-coordinate and y-coordinate of v, as its
      value
    else
    - set val to the value of key k in acc
    - increment the first element of val by 1
    - sum the x-coordinate of v to the second element of val
      and the y-coordinate of v to the third element of val
    - sum the squared coordinates of v to the fourth element of val
- return acc
```

```
def square(v):
    return sum(x**2 for x in v)

def accumulation(file_name):
    data = load(file_name)
    acc = dict()
    for v in data:
        k = v[-1]
        x = v[1]
```

```
        y = v[2]
        if k not in acc.keys():
            acc[k] = [1, x, y, x**2 + y**2]
            # acc[k] = [1] + v[1:3] + [square(v[1:3])] # prof version
        else:
            val = acc[k] # interesting this stuff to ease access
            val[0] += 1
            val[1] += x
            val[2] += y
            # for j in range(1,3): # versione più comoda se si hanno più di due valor
            #     val[j] += v[j]
            val[3] += x**2 + y**2
            #  val[3] += square(v[1:3])
    return acc
```

Per la verifica

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

file_name = "data.csv"
# # prof ha usato sto file
# 0, 0.0, 0.5, a
# 1, 0.5, -0.5, a
# 2, -0.5, 1.0, a
# 3, 0.0, 3.0, b
# 4, -0.5, 2.5, b
# 5, -3.0, 0.0, b

data = np.loadtxt(file_name, delimiter = ",")
summary = accumulation(file_name)
summary
```

Example 2.5.5 (exam 2022-09-03)**.** Let `boundaries` be a non-decreasing list of n numbers, such that it first element is $-\infty$ and its last element is ∞. Two consecutive elements of boundaries are the bound of an interval on the real line. Write a python function `fill(data, boundaries)` that outputs that outputs two objects:

- a list of n lists, called buckets sucht that the i-th bucket, with $i = 0, 1, \ldots, n-1$ contains exactly the numers in list data which belong to the interval boundaries[i], boundaries[i+1]
- a list of the numbers of elements in each bucket

Function implement the following algorithm

```
- let buckets be the empty sequence
- for all j from 1 to the length of boundaries
  - append an empty sequence to buckets
- let counts be a sequence of n zeros, where n is the length of boundaries
```

```
- for all element x of data
  - for all j from 1 to the length of boundaries
    - if the j-th element of boundaries is strictly greater than x
      - stop iteration
    - append x to the sequence in position j-1 in buckets
    - increment the number in position j-1 in counts by 1
- return the sequence buckets and the sequence counts
```

```
def fill(data, boundaries):
    buckets = []
    # when the algorithm supply start and end position
    # for j in range(1, len(boundaries) + 1):
    # the same, since we dont use j in the cycle
    for j in range(len(boundaries)):
        buckets.append([])
    counts = [0] * len(boundaries)
    for x in data:
        for i in range(len(boundaries)):
            if boundaries[i] > x:
                break
            buckets[i-1].append(x)        # cosi è come fa lui
            counts[i-1] = counts[i-1] + 1
            # buckets[i].append(x)        # boh grande casino io farei cosi
            # counts[i] = counts[i] + 1

    return buckets, counts
```

Example 2.5.6 (2018-01-15)**.** Algoritmo ad cazzum di risoluzione di un esame trovato su virtuale (nessun testo disponibile)

```
def sq_sum(x):
    return sum((y ** 2 for y in x))


def sq_norm_diff(x, y):
    return sq_sum((z[0] - z[1] for z in zip(x, y)))


def nn(data, x):
    distances = list(enumerate([sq_norm_diff(x, y) for y in data]))
    m = min(distances, key = lambda t: t[1])
    del distances[m[0]]
    m = min(distances, key = lambda t: t[1])
    return data[m[0]]


def label(data, centers):
    labels = []
    for x in data:
        nearest = nn(data, x)
        mid = [(px + pnn)/2 for px, pnn in zip(x, nearest)]
```

```
        cent_dist = [sq_norm_diff(mid, y) for y in centers]
        c = min(list(enumerate(cent_dist)), key=lambda t: t[1])[0]
        labels.append((x, c))
    return labels
```

2.6 Conda

environment

```
# crea un progetto con lista di pacchetti richiesti oltre a quelli base
conda create -n myproject keras matplotlib
conda activate myproject
# vado in python
import matplotlib.pyplot as plt
# e dovrebbe funzionare
```

once

Capitolo 3

Clustering

Remark 22. Clustering is not formally defined; there are many/different definitions in Statistics, Data Mining, and Machine Learning. We refer to the classic book, by Hastie, Tibshirani Friedman

Definition 3.0.1 (Cluster analysis)**.** Cluster analysis, also called data segmentation, has a variety of goals. All relate to grouping or segmenting a collection of objects into subsets or "clusters", such that those within each cluster are more closely related to one another than objects assigned to different clusters.

Remark 23 (Clustering instances). Since there is no formal definition of clustering, no single formal problem can be defined. Different problem formulations of cluster all agree on the description of the problem instance

Definition 3.0.2 (Clustering problem)**.** Just a forma refrasing dell'idea informale per farlo felice:

Instance A pattern set $\mathcal{D} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$, where each *pattern* $\mathbf{x}_i$ is a sequence of p *measurements*, called features/attributes, and a possibly empty set of additional constraints

Solution Mathematical specification of a relation between the instance/cases and some cluster assignment for the $\mathbf{x}_i$, which implies closeness of patterns in the same cluster

Remark 24. We want basically a function which assigns each observation $\mathbf{x}_i$ to a group/cluster

Important remark 9 (Possible approaches). There are several possible approaches to do that: particularly there are two main approaches to the mathematical definition of "closely related":

- **Similarity measure**: a domain-dependent (eg different from application to application) real function of the pattern/unit pairs, measuring their similarity, is assumed. Similar patterns/units are closely related, dissimilar ones are not
- **Density estimation**: a statistical estimate of the probability density function that generated the data is computed. Patterns in regions where the estimated p.d.f. is large are more closely related

Important remark 10 (Types of clustering models). Beside of the idea of closeness we have to choose the type of clustering where we have

- **Hierarchical or flat**: *Hierarchical* Clusters may be subdivided into smaller, contained subclusters, forming a hierarchy
- **Exclusive**, **overlapping**, or **fuzzy**:
  - *Exclusive*: any pattern is an element of exactly one cluster
  - *Overlapping*: any pattern may be an element of more than one cluster
  - *Fuzzy*: for each cluster, a membership function on patterns onto [0, 1] is defined
- **Complete** or **partial**:
  - *Complete*: any pattern is an element of some cluster
  - *Partial*: a pattern may belong to no cluster; patterns not belonging to any cluster can be deemed as *noise*, or *outliers*

3.1 Similarity and dissimilarity

Definition 3.1.1 (Similarity)**.** Similarity is a real function of two arguments

$$s : (x, y) \in \mathcal{D} \times \mathcal{D} \to s_{xy} \in \mathbb{R} \tag{3.1}$$

where $\mathcal{D}$ is a pattern set, usually satisfying a few simple properties. Some common are:

- Simmetry: $s_{xy} = s_{yx}$
- Boundedness between 0 and 1: $s_{xy} \in [0, 1]$ (0 for least similar, 1 for most similar)
- Identity of maximally similar patterns: $s_{xy} = 1 \implies x = y$

Remark 25. Some considerations:

- the choice of a suitable function is entirely application-dependent
- dissimilarity, denoted d_{xy} , is often used instead of similarity, modeled as 3.1

Remark 26. If the patterns are described as multivariate observations over p-variables (eg vectors), then a more specific form of the similarity function (and dissimilarity) can be given

Definition 3.1.2 (Similarity measure (Everitt))**.** A similarity measure s_{xy} is a function of the observed values of p variables of each pattern:

$$s_{xy} = f(x, y)$$

where $x = (x_1, x_2, \ldots, x_p)$, $y = (y_1, y_2, \ldots, y_p)$.

$\mathbf{x}\downarrow \mathbf{y}\rightarrow$	1	0	
1	a	b	a+b
0	c	d	c+d
	a+c	b+d	p

Tabella 3.1: table

Important remark 11 (Similarity in case of binary variables). If two patterns $\mathbf{x} = (x_1, x_2, \ldots, x_p)$, $\mathbf{y} = (y_1, y_2, \ldots, y_p)$, with x_k , y_k binary (that is, $x_k \in \{m_0, m_1\}$, $y_k \in \{m_0, m_1\}$, for simplicity assume $m_0 = 0$, $m_1 = 1$) we can define the table 3.1 of matches and mismatches between the values of the p features for $\mathbf{x}, \mathbf{y}$; here

- a is the number of *features* which are both 1 in the first and second unit ($\mathbf{x}$ and $\mathbf{y}$)
- c is the number of features where $\mathbf{x}$ has value 0 while $\mathbf{y}$ has value 1
- ...
- p is obv the number of features

In this case similarities are ratios combining a, b, c, d:

- *Matching coefficient* (percentage of coinciding features among two units):

$$M_{xy} = \frac{a+d}{p}$$

- *Jaccard's coefficient* is similar but ignore the negative/0 concordances

$$J_{xy} = \frac{a}{a+b+c}$$

  eg in biology used to look which species has most in common (both 1) ignoring the 0s; since there are a lot of features and we don't want to give weight to things (classify similar) that both two species don't have

Important remark 12. Matching coefficient for categorical features ranging over more than two values is the following

$$s_{xy} = \frac{\sum_{k=1}^{p} c_{xy}^k}{p} \tag{3.2}$$

where c_{xy}^k is a score for feature k and observation $\mathbf{x}$ and $\mathbf{y}$ which is just an agrement between modalità

$$c_{xy}^k = \begin{cases} 1 & \text{if } x_k = y_k \\ 0 & \text{otherwise} \end{cases}$$

Important remark 13 (Frequent properties of dissimilarities). we have

- Symmetry: $d_{xy} = d_{yx}$

- Identity of indiscernibles: $d_{xy} = 0 \implies x = y$

Boundendess does not hold in general: There are important dissimilarities that do not satisfy boundedness (see, for example, the Minkowski metrics below). If one want to develop a dissimilarity bounded by $[0, 1]$ we can just transform a bounded similarity by

$$d_{xy} = 1 - s_{xy}$$

Definition 3.1.3 (Distances)**.** A metric or distance is a function which captures some geometrical properties of space, having some properties:

- non-negativity: $dxy \geq 0$
- identity of indiscernibles: $d_{xy} = 0 \iff x = y$
- symmetry: $d_{xy} = d_{yx}$
- triangular inequality: $d_{xy} + d_{xz} \geq d_{yz}$

NB: dice che sono equivalenti alla classica $d_{xy} + d_{yz} \geq d_{xz}$

Example 3.1.1 (Minkowski metrics)**.** Family of metrics characterized by an integer r, defined as summationover all the features of two patters:

$$d_{xy} = \left(\sum_{k=1}^{p} |x_{ik} - x_{jk}|^r \right)^{1/r}$$

As a special case we have eudlidean distance and the manhattan distance The dissimilarities derived from M_{xy} and J_xy as $1-M_{xy}$ and $1-J_{xy}$ are also distances

3.2 Hierarchical methods

Important remark 14 (Hierarchical clustering). Hierarchical clustering produces a *hierarchical*, *exclusive*, and *total* type of clustering; its output model can be described as a N-tree.

Definition 3.2.1 (N-tree)**.** T is a N-tree on a set $A = \{a_1, a_2, \ldots, a_N\} \iff$ T is a collection of *subsets* of A satisfying

$$\begin{aligned}
&\emptyset \in T \\
&A \in T \\
&\{a_i\} \in T, \quad i = 1, 2 \ldots, N \\
&X, Y \in T \implies X \cap Y \in \{\emptyset, X, Y\}
\end{aligned}$$

the first three are the empty, whole and singleton sets are part of the collection of subsets of A; the last: if our collection contain two sets, then there are only three cases about their intersection, is empty (they don't have elements in common) or one set between X and Y includes the other

Example 3.2.1. The subsets constituting an N-tree can be organised into a tree graph (fig 3.1). Each node represents a subset. A directed edge goes from a node to the smallest of its containing nodes. (There is only one smallest node among the containing nodes of any node, except the empty set.)

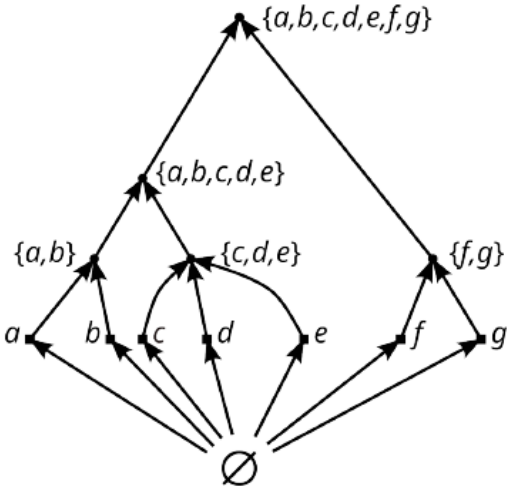


Figura 3.1: An N-tree on $\{a, b, c, d, e, f, g\}$

Important remark 15 (Classes of clustering algorithms). Hierarchical clustering algorithms compute a sequence of partitions of $\mathcal{D}$, such that at any step, the collection of all subsets in all computed partitions, that is, their union, is a N-tree on $\mathcal{D}$.
Two classes of hierarchical clustering algorithms exist: *agglomerative* (starts from singleton and sum up to the whole) and *divisive* (start from the whole and end with singleton): they differ as to how the next partition in the sequence is obtained from the last one, and the initial partition.
Precisely:

- in *agglomerative clustering*, the initial partition is the partition of all singletons $\{\{\mathbf{x}_1\}, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$: the next partition in the sequence is obtained by replacing a pair of clusters with their union
- in *divisive clustering*, the initial partition contains only the pattern set $\{\mathcal{D}\}$: the next partition in the sequence is obtained by replacing a cluster with one of its partitions into two clusters

Remark 27. Agglomerative clustering algorithms are simpler and more popular; *Anderberg's algorithm* is a *general scheme for agglomerative clustering* which includes classical algorithms as special cases

Important remark 16 (Anderberg's agglomerative algorithm). The algorithm is as follows:

1. Initialise
   (a) N clusters $C_1, C_2, \ldots, C_N$ containing a unique pattern each
   (b) a $N \times N$ matrix in which entry (i, j) is d_{ij}
   (c) the number of clusters to N
2. Find a pair of most similar clusters in the matrix, according to a fixed *cluster dissimilarity*, which extends pattern dissimilarity (usable for sin-

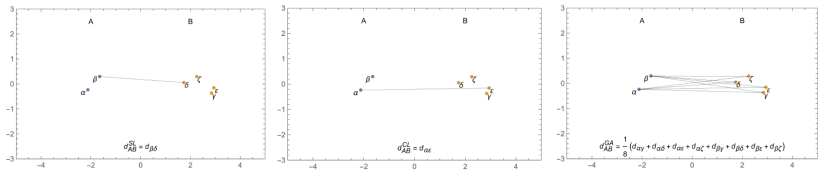


Figura 3.2: Links vari: a) single link, b) complete link, c) group average

gleton sets). Several choices are available (see fig 3.2), suppose A and B are two cluster:

- in *single link* the distance between two cluster is the minimum distance among their elements: $d_{AB}^{SL} = \min_{x \in A, y \in B} d_{xy}$
- in *complete link* is the maximum $d_{AB}^{CL} = \max_{x \in A, y \in B} d_{xy}$
- in *group average* is the mean distance $d_{AB}^{GA} = \frac{1}{|A|\,|B|} \sum_{x \in A, y \in B} d_{xy}$

Let p, q be the indices of most similar clusters

3. Create the union of C_p and C_q and assign index q to it
4. Update the matrix by computing the dissimilarities between C_q and the other clusters, and deleting the row and column with index p
5. Decrement the number of clusters by 1
6. If the number of clusters equals 1, terminate, else go to Step 2

3.3 Partitioning methods

Definition 3.3.1 (Partitioning model)**.** The partitioning model produces a *flat*, *exclusive*, and *complete* type of clustering model. It consists of a partition of the data set $\mathcal{D}$ containing exactly K classes (with K a parameter of the algorithm).

Remark 28. The partitioning model only specifies the type of solution, and an additional constraint on the number of clusters, which we include in the instance definition as an integer $K \geq 2$

Remark 29. Partitioning models can be simply represented as *encoders*.

Definition 3.3.2 (Encoder)**.** It's a function

$$C : \mathcal{D} \to \{1, 2, \ldots, K\}$$

mapping each pattern to a positive integer cluster label

Remark 30 (Partition of an encoder). Given an encoder C, the corresponding partition is

$$\{\{\mathbf{x} \in \mathcal{D} : C(\mathbf{x}) = k\} : k \in \{1, 2, \ldots, K\}\}$$

Remark 31. The relation between instance and solution for partitioning models is specified as a loss function (eg to solve the problem we need to specify it)

Definition 3.3.3 (Loss function)**.** A real function $L : C \to L(C) \in \mathbb{R}$ which maps an encoder C to a real number representing the *inaccuracy* of the clustering encoded by C

Remark 32. Ideally we want to find the optimal C by minimizing $L(C)$

Remark 33 (Within clusters and between clusters). Considering that patterns in a cluster should be more closely related than patterns in different clusters:

- any monotonic function W of *dissimilarities within* each cluster is a potential loss function to *minimise*
- any monotonic function B of the *dissimilarities between* patterns in a cluster and all other patterns should instead be *maximised* (or equivalently, its opposite is a loss function to minimise). Such a function is a measure of *cluster separation*

Let's see some function which can be used for what above: *clique* is an example of W where the monotonic function is the sum, while *sum of cuts* is an example of the second

Definition 3.3.4 (Clique)**.** Clique of the encoder C is the sum of all within-cluster dissimilarities (between patterns in the same cluster):

$$clique(C) = \frac{1}{2} \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(i')=k} d_{ii'}$$

where, where $k = 1, \ldots, K$ are cluster and

- the innermost summation $\sum_{C(i')=k} d_{ii'}$, depending on i and k, represents the sum of dissimilarities between pattern i and all patterns i' in cluster k.
- Summation $\sum_{C(i)=k}$ then is applied to all patterns in cluster k
- finally $\sum_{C(i)=k} \sum_{C(i\prime)=k} d_{ii'}$ is the sum of all dissimilarities between patterns in cluster k, for which we sum over all the cluster

Definition 3.3.5 (Sum of cuts)**.** Its the sum of all dissimilarities between patterns in different clusters
The sum of cuts of encoder C is the sum of all between-cluster dissimilarities:

$$cut(C) = \frac{1}{2} \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(i')\neq k} d_{ii'}$$

where the innermost summation computes the sum of dissimilarities between pattern i and all patterns i' not in cluster k, so that $\sum_{C(i)=k} \sum_{C(i\prime)\neq k} d_{ii'}$ is the sum of all dissimilarities between patterns in cluster k and patters in other clusters

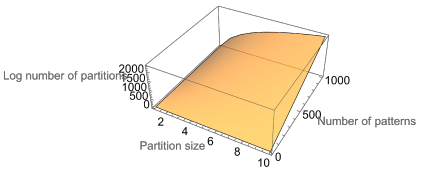


Figura 3.3: Search space

Important remark 17 (Minimisation). Ideally since we want to minimize W-like and maximize B-like function, overall we should minimize

$$W - B = clique(C) - cut(C)$$

However, point is that sum of clique and between-cluster dissimilarities is a *constant*

$$\begin{aligned} clique(C) + cut(C) &= \frac{1}{2}\sum_{k=1}^{K}\sum_{C(i)=k}\sum_{C(i')=k} d_{ii'} + \frac{1}{2}\sum_{k=1}^{K}\sum_{C(i)=k}\sum_{C(i')\neq k} d_{ii'} \\ &= \frac{1}{2}\sum_{k=1}^{K}\sum_{C(i)=k}\left(\sum_{C(i')=k} d_{ii'} + \sum_{C(i')\neq k} d_{ii'}\right) \\ &= \text{constant} \end{aligned}$$

Therefore *minimising clique is equivalent to maximising the sum of cuts* and we can choose either one or the other for optimization purposes

Remark 34 (Search space). We now have a complete specification of the problem (we're computing an argmax/argmin choosing C). Constructing the solution is not trivial

- The number of partitions of size K for pattern set $\mathcal{D}$ is huge (fig **??**
- Scanning the whole space of encoders with a minimum algorithm is clearly infeasible

Remark 35. Since scanning the whole space is infeasible someone tried to find an approximate solution (not an exact), by minimising for squared distances

In a dot product space (with vectors of p real numbers), consider the clique with *squared* distances as our loss function; let $\bar{\mathbf{x}}_k$ denote the mean of the k-th cluster, as defined by encoder C:

- The squared distance between two units can be written as

$$\begin{aligned} d^2_{ii'} &= \|\mathbf{x}i - \mathbf{x}_{i'}\|^2 \\ &= (\mathbf{x}_i - \bar{\mathbf{x}}_k + \bar{\mathbf{x}}_k - \mathbf{x}_{i'}) \cdot (\mathbf{x}_i - \bar{\mathbf{x}}_k + \bar{\mathbf{x}}_k - \mathbf{x}_{i'}) \\ &= \|\mathbf{x}_i - \bar{\mathbf{x}}_k\| + \|\bar{\mathbf{x}}_k - \mathbf{x}_{i'}\| + 2(\mathbf{x}_i - \bar{\mathbf{x}}_k) \cdot (\bar{\mathbf{x}}_k - \mathbf{x}_{i'}) \end{aligned}$$

- thus the clique (within loss function, W for this) can be redefined as:

$$W_{sq}(C) = \frac{1}{2}\sum_{k=1}^{K}\sum_{C(i)=k}\sum_{C(i')=k} d_{ii'}^2$$
$$= \sum_{k=1}^{K}\sum_{C(i)=k} \|\mathbf{x}_i - \bar{\mathbf{x}}_k\|$$

  So we summing over all the cluster, the number of that cluster N_k multiplied by the sum of squared distances between each unit in the cluster and the cluster mean

Remark 36. The algorithm that uses this distance is k-means

Definition 3.3.6 (K-Means)**.** It is

- iterative, numerical data, local optimum only (it find a local minimum of the loss function, not necessarily the overall minimum)
- the goal is minimises the *sum of squared deviations* (SSD) of clusters, weighted by cluster size:

$$W_{kmeans}(C) = W_{sq}(C) = \sum_{k=1}^{k} N_k \sum_{C(i)=k} \|\mathbf{x}_i - \bar{\mathbf{x}}_k\|^2$$

- how does k-means do that: the key observation is that the mean minimizes the sum of squares of deviation and auxiliary variables for the means do not change the minimisation problem; suppose that C^* is the encoder minimizing the loss function

$$C^* = \arg\min_C W_{kmeans}(C)$$

  If we "extend" the solution searched by putting the centroid of the cluster as well in the solution searched for (we have encoders and their produced centroids), that would be minimizing over C and auxiliary variables searched for centroids $\mathbf{m}_1, \dots \mathbf{m}_k$ on a function called W'_{kmeans} (depending on them)

$$(C, \bar{\mathbf{x}}_1, \dots, \bar{\mathbf{x}}_K) = \arg\min_{C, \mathbf{m}_1, \dots, \mathbf{m}_K} W'_{kmeans}(C, \mathbf{m}_1, \dots, \mathbf{m}_K)$$

  This is also same definition as above where we substituted $\bar{\mathbf{x}}_k$, $\mathbf{m}_k$

$$(C, \bar{\mathbf{x}}_1, \dots, \bar{\mathbf{x}}_K) = \arg\min_{C, \mathbf{m}_1, \dots, \mathbf{m}_K} \sum_{k=1}^{k} N_k \sum_{C(i)=k} \|\mathbf{x}_i - \mathbf{m}_k\|^2$$

  The introduction of these variables $\mathbf{m}_1, \dots, \mathbf{m}_K$ does not change the optiomization, because we'll end with the same results because the encoders minimizes the SSD (which is minimum only when the centroid mean is chosen as $\mathbf{m}_k$)

Remark 37. The idea is to exploit these auxiliary variable explicitly in an algorithm

Important remark 18 (K-Means algorithm). The steps are:

1. Select and initial encoder C
2. Given the current encoder C, minimize $W_{kmeans}(C, m_1, \ldots, m_K)$ w.r.t. $m_1, \ldots, m_K$, yielding the means of the clusters
$$\arg\min_{m_1,\ldots,m_K} W'_{kmeans}(C, m_1, \ldots, m_K)$$
3. Given the current means $m_1, \ldots, m_K$, minimize $W_kmeans'(C, m_1, \ldots, m_K)$ with respect to C
$$\arg\min_{C} W'_{kmeans}(C, m_1, \ldots, m_K)$$
4. Repeat 2 and 3 until C and $m_1, \ldots, m_K$ do not change

Remark 38. This is an alternating minimization procedure that usually will produce a value not to faar from the optimal
pros: it's fast and available everywhere generalization are used to relax the initial selection of encoder C

3.4 Clustering in python

Clustering methods are implemented in libraries provided by two projects:

1. `SciPy`: libraries for math, science, and engineering (Linear algebra, statistics, integration, Fourier transforms, signal and image processing, and clustering)
2. `Scikit-learn`: machine Learning algorithms which uses SciPy

Scikit-learn also implements methods from the data mining literature, explicitly designed for processing large data sets

Hierarchical clustering in SciPy The two stuff are

```
>>> import scipy.cluster.hierarchy as hy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3/dist-packages/scipy/cluster/__init__.py", line 27, in <modul
    from . import vq, hierarchy
  File "/usr/lib/python3/dist-packages/scipy/cluster/vq.py", line 74, in <module>
    from scipy.spatial.distance import cdist
  File "/usr/lib/python3/dist-packages/scipy/spatial/__init__.py", line 110, in <modu
    from ._kdtree import *
  File "/usr/lib/python3/dist-packages/scipy/spatial/_kdtree.py", line 4, in <module>
    from ._ckdtree import cKDTree, cKDTreeNode
  File "_ckdtree.pyx", line 1, in init scipy.spatial._ckdtree
```

```
ValueError: numpy.dtype size changed, may indicate binary incompatibility. Expected 96 from C h
>>> from scipy.spatial.distance import pdist
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3/dist-packages/scipy/spatial/__init__.py", line 110, in <module>
    from ._kdtree import *
  File "/usr/lib/python3/dist-packages/scipy/spatial/_kdtree.py", line 4, in <module>
    from ._ckdtree import cKDTree, cKDTreeNode
  File "_ckdtree.pyx", line 1, in init scipy.spatial._ckdtree
ValueError: numpy.dtype size changed, may indicate binary incompatibility. Expected 96 from C h
```

which

-
- Compute the condensed (upper triangular) distance matrix, possibly from a sample of the data set
- Compute a matrix representation of the N -tree
- Draw the dendrogram, selectively colouring the branches (use a merging distance cutoff)

```
>>> # act_data = data[sam_ids]
>>> # distances = pdist(act_data, metric='euclidean')
>>> # tree = hy.linkage(distances, method='complete')
>>> # cutoff = 0.25 * np.max(tree[:, 2])
>>> # dgram = hy.dendrogram(tree, orientation='top',
>>> #                       color_threshold=cutoff)
```

K-means clustering in SciPy

```
>>> import numpy as np
>>> from scipy.cluster import vq
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3/dist-packages/scipy/cluster/__init__.py", line 27, in <module>
    from . import vq, hierarchy
  File "/usr/lib/python3/dist-packages/scipy/cluster/vq.py", line 74, in <module>
    from scipy.spatial.distance import cdist
  File "/usr/lib/python3/dist-packages/scipy/spatial/__init__.py", line 110, in <module>
    from ._kdtree import *
  File "/usr/lib/python3/dist-packages/scipy/spatial/_kdtree.py", line 4, in <module>
    from ._ckdtree import cKDTree, cKDTreeNode
  File "_ckdtree.pyx", line 1, in init scipy.spatial._ckdtree
ValueError: numpy.dtype size changed, may indicate binary incompatibility. Expected 96 from C h
```

It

- Compute the centroids of the data set
- Compute the centroid number for each observation
- Construct the clusters as a list of lists of observations

```
>>> # k = 24
>>> # centroids, variance = vq.kmeans(data, k)
>>> # identified, distance = vq.vq(data, centroids)
>>> # clusters = []
>>> # for c in range(k):
>>> #     clusters.append(data[identified == c])
```

3.4.1 Leader-follower

An old and simple clustering algorithm which handle memory limitation by storing in memory not single observation but summary of observations

basic idea of the algorithm is: we have a dataset/patterns

we pick an element from the dataset, each pattern has p features, we put these features in a structure called *summary*; each cluster will have one summary.

we copy the features of the first element in the summary, so after read the first element there's only one cluster containing one element

we also store the number of element of the cluster and the squared norm of the element

the idea is to continue reading elements and modifying the summaries

how to create more classes/cluster: pick a newer element, search the summaries and try to evaluate if the new element can fit reasonably in that cluster looking at the summary. If so we add to the cluster and change its stats, otherwise we initialize another summary

called leader follower because the first element add to a cluster is considered a leader, the others followers

we use variability to decide wheter a certain element can fit in a cluster or not: if after adding the element the variability of the cluster increases too much than we put the object in another cluster, mabye new

Dataset da provare l'algoritmo in sezione the clustering problem

```
>>> ## leader follower data clustering algorithm

>>> # Let F be a statistics measuring the variability of a set of
>>> # patterns. The leader follower algorithm will be:
>>> #
>>> # 1. Read a pattern as x from the data source, and compute statistics
>>> #    F_0 as a representative of the 1st cluster C_0 = {x}
>>> #
>>> # 2. While there are patterns in the data source:
>>> #
>>> #    1. Read a pattern as x, and find, among the stored statistics
>>> #       F_i, with i=1,..., k, statistics F_{i^*} which would have the
>>> #       minimum value (minimum increase in variability) if all
>>> #       statistics were updated to include x in the represented
>>> #       clusters.
>>> #
>>> #    2. Once found the best candidate (lowest increase in variability)
>>> #       we have to choose if we add to it or create a new cluster
>>> #       because the variability increase is too much. So we use a
```

```
>>> #       treshold. If the minimum is smaller than a threshold parameter
>>> #       \theta, update F_{i^*} using x. Otherwise compute a new summary
>>> #       F_{k+1}, which will be representative of a new cluster, C_{k+1} = {x}
>>> #
>>> #
>>> #  It's simple, memory efficient, and reads data only once: in
>>> #  practice we would run this shit for many time and choose the theta
>>> #  which produce the "best" cluster according to an external criterion
>>> #  A drawback of the algorithm is that it depends on the order of the
>>> #  data, producing different clusters with same data permuted (it
>>> #  depends on the point which become leaders)

>>> # --------------------
>>> # some helper function
>>> # --------------------

>>> # sum of squares of a vector using a sum on a comprehension
>>> def ssq(x):
...     return sum(v**2 for v in x)
...
>>> # the cluster stats maker/updater.
>>> def h(summary, x):
...     # update the linear sum by adding the x features, summing
...     # parallel vectors/list using zip
...     ls = [u + v for u, v in zip(summary['LS'], x)]
...     return {"N": summary["N"] + 1,
...             "LS" : ls,
...             "SS" : summary["SS"] + ssq(x)}
...

>>> # function tells values used to decide if we can add an element to a
>>> # summary or not and/or to select the best candidate
>>> def g(summary, x):
...     new_summary = h(summary, x) # how summary would be if we added x
...     # a variability measure: we divide every element of the linear sum by N, compute
...     # the square and then sum: secondo lui è una varianza
...     squared_mean = sum((v/new_summary['N'])**2
...                        for v in new_summary['LS'])
...     # we return sum of squares divided by n and subtract from that the squared mean
...     return new_summary['SS']/new_summary['N'] - squared_mean
...
>>> def argmin(x):
...     # we use min with lambda function selecting the second element of enumerated sequence
...     # it will extract the lowest value (second element) and its associated progressive id (
...     return min(enumerate(x), key = lambda x: x[1])[0] # keep just the first element (id)
...
>>> # # what does enumerate do: add a progressive numeric to a sequence
>>> # lst = ['a','b','c']
>>> # list(enumerate(lst)) # [(0, 'a'), (1, 'b'), (2, 'c')]
>>> # list(enumerate([0, -1, 6]))
```

```
>>> # min(enumerate([0, -1, 6]))
>>> # argmin([0,-1,2])
>>> # 1

>>> # Main algorithm
>>> def leader_follower(ds, theta): # ds is the data as sequence, theta is the parame
...     # we pick an x
...     x = ds[0]
...     # we initialize a summary template: it will have count of elements, linear su
...     # of the features, sum of squares. We'll update it using a function h
...     zero_summary = {
...         "N": 0,  # number of elements
...         "LS": [0] * len(x), # a vector of zeros length the number of
...                             # features containing sum of the features
...         "SS": 0   # sum of squares
...     }
...     # h is the function which initialize the summary with actual data
...     summary = [h(zero_summary, x)]
...     # we'll grow this list (of dictionaries output by h) with other
...     # elements (one per cluster).
...
>>>     # step 2 here. we start from the second unit
>>>     for x in ds[1:]:
  File "<stdin>", line 1
    for x in ds[1:]:
IndentationError: unexpected indent
>>>         # compute the loss/variance associated to add of x at each cluster
>>>         loss = [g(s, x) for s in summary]
  File "<stdin>", line 1
    loss = [g(s, x) for s in summary]
IndentationError: unexpected indent
>>>         # select the cluster with the lowest loss: argmin (to be
>>>         # implemented) return the position with minimum loss
>>>         j = argmin(loss)
  File "<stdin>", line 1
    j = argmin(loss)
IndentationError: unexpected indent
>>>         # decide if x is a follower (add to a cluster) or a leader
>>>         # (start a new cluster)
>>>         if loss[j] <= theta:
  File "<stdin>", line 1
    if loss[j] <= theta:
IndentationError: unexpected indent
>>>             # acceptable loss: add it to the best, it's a follower
>>>             summary[j] = h(summary[j], x)
  File "<stdin>", line 1
    summary[j] = h(summary[j], x)
IndentationError: unexpected indent
>>>         else:
  File "<stdin>", line 1
```

```
    else:
IndentationError: unexpected indent
>>>             # the loss is too much: it will be a leader. We add a
>>>             # new summary
>>>             summary.append(h(zero_summary, x))
  File "<stdin>", line 1
    summary.append(h(zero_summary, x))
IndentationError: unexpected indent

>>>     # this algorithm does not output labels but the groups statistics:
>>>     # it's a price to pay. However if we have the n for each cluster
>>>     # the linear sum of features and the sum of squares one can
>>>     # compute a lot of descriptive statistics.  to get the labelling
>>>     # one has to scan the dataset twice: one to find the statistics of
>>>     # leader follower and the second to compute the assignment
>>>     return summary
  File "<stdin>", line 1
    return summary
IndentationError: unexpected indent

>>> # ------------------------------------------
>>> # a numpy version
>>> # ------------------------------------------
>>> # advantages over standard python:
>>> # - numpy allows to use vector operators
>>> #   without writing loop/comprehension
>>> # - faster

>>> import numpy as np


>>> # np.inner does the product between two vector: applied to himself
>>> # def ssq(x):
>>> #     return np.inner(x, x)


>>> # add the summary (np.array) another np.array with 1 (count increase, the
>>> # values of the unit to linear sum and the sum of squares to the last
>>> # here we use position of an array to find stuff instead of dictionary
>>> def h(summary, x):
...     return summary + np.array([1, *x, np.inner(x, x)])
...
>>> # function to compute the new situation if we add an element to a cluster
>>> def g(summary, x):
...     new_summary = h(summary, x)
...     new_mean = new_summary[1:len(x)+1]/new_summary[0] # LS/N
...     return new_summary[-1]/new_summary[0] - np.inner(new_mean, new_mean)
...

>>> def leader_follower(ds, theta): # ds is numpy array instead of a list of list
```

```
...     # we pick an x
...     x = ds[0]
...     # we create a summary: it will have count of elements, linear sums
...     # of the features, sum of squares
...     zero_summary = np.array([0, *[0]*len(x), 0])
...     # we reshape the 1d array of the first cluster (output of h) in a
...     # 2d object (it will contain the summaries for each cluster). we
...     # specify (for now) it has 1 row, and a number of colums equal to
...     # actual number of elements of the zero summary
...     summary = h(zero_summary, x).reshape(1, len(zero_summary))
...     for x in ds[1:]:
...         losses = np.array([g(s, x) for s in summary])
...         j = np.argmin(losses)
...         if losses[j] <= theta:
...             # replace the summary with the new one
...             summary[j] = h(summary[j], x)
...         else:
...             # create a new cluster
...             summary = np.concatenate((summary,
...                                       h(zero_summary, x).reshape(1, len(zero_summ
...     return summary
...


>>> # ---------------------
>>> # testing the algorithm
>>> # ---------------------
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/l/.local/lib/python3.12/site-packages/seaborn/__init__.py", line 9, in
    from .matrix import *  # noqa: F401,F403
    ^^^^^^^^^^^^^^^^^^^^
  File "/home/l/.local/lib/python3.12/site-packages/seaborn/matrix.py", line 11, in <
    from scipy.cluster import hierarchy
  File "/usr/lib/python3/dist-packages/scipy/cluster/__init__.py", line 27, in <modul
    from . import vq, hierarchy
  File "/usr/lib/python3/dist-packages/scipy/cluster/vq.py", line 74, in <module>
    from scipy.spatial.distance import cdist
  File "/usr/lib/python3/dist-packages/scipy/spatial/__init__.py", line 110, in <modu
    from ._kdtree import *
  File "/usr/lib/python3/dist-packages/scipy/spatial/_kdtree.py", line 4, in <module>
    from ._ckdtree import cKDTree, cKDTreeNode
  File "_ckdtree.pyx", line 1, in init scipy.spatial._ckdtree
ValueError: numpy.dtype size changed, may indicate binary incompatibility. Expected 9

>>> # from leader_follower import leader_follower
```

```
>>> data = np.array([(1, 0), (2, 1), (0, 2), (1, 3), (6, 0), (7, 0), (8, 1), (8, 2)])

>>> # this will collect the summary for each treshold tested
>>> summary = []

>>> # do the stuff for several threshold theta
>>> thresholds = [x/10 for x in range(10,20)]
>>> for t in thresholds:
...     summary.append([t, leader_follower(data, t)])
...
>>> print(summary)
[[1.0, array([[  2,   3,   1,   6],
       [  2,   1,   5,  14],
       [  3,  21,   1, 150],
       [  1,   8,   2,  68]])], [1.1, array([[  2,   3,   1,   6],
       [  2,   1,   5,  14],
       [  3,  21,   1, 150],
       [  1,   8,   2,  68]])], [1.2, array([[  2,   3,   1,   6],
       [  2,   1,   5,  14],
       [  3,  21,   1, 150],
       [  1,   8,   2,  68]])], [1.3, array([[  2,   3,   1,   6],
       [  2,   1,   5,  14],
       [  3,  21,   1, 150],
       [  1,   8,   2,  68]])], [1.4, array([[  3,   3,   3,  10],
       [  1,   1,   3,  10],
       [  4,  29,   3, 218]])], [1.5, array([[  3,   3,   3,  10],
       [  1,   1,   3,  10],
       [  4,  29,   3, 218]])], [1.6, array([[  3,   3,   3,  10],
       [  1,   1,   3,  10],
       [  4,  29,   3, 218]])], [1.7, array([[  3,   3,   3,  10],
       [  1,   1,   3,  10],
       [  4,  29,   3, 218]])], [1.8, array([[  4,   4,   6,  20],
       [  4,  29,   3, 218]])], [1.9, array([[  4,   4,   6,  20],
       [  4,  29,   3, 218]])]]

>>> # Visualisation: for every execution (theta) we want a plot
>>> for i in range(len(summary)):
...     # extract the i-th iteration,  just the data, fuck the theta
...     stats = summary[i][1]
...     # means for the p variables (exclude first and last):
...     means = [s[1:-1]/s[0] for s in stats]
...     # above it's a list of arrays, with each one having the mean for all the
...     # variable inside a cluster; now we want a list of list organized
...     # per each variable (not cluster). This is how to do it
...     mean_cols = list(zip(*means))
...     # data per columns, not per rows
...     data_cols = list(zip(*data))
...     plt.figure(i)
...     # display both data and means for first variable (below not sum,
```

```
...     # but a list concatenation) on the x
...     sns.scatterplot(x = data_cols[0] + mean_cols[0],
...                     # do the same for second variable on the y
...                     y = data_cols[1] + mean_cols[1],
...                     # colors: create a groups of colors by concatenating cols for
...                     hue = ("data",) * len(data) + tuple(range(1, len(mean_cols[0]
...
<Figure size 2007.6x2007.6 with 0 Axes>
Traceback (most recent call last):
  File "<stdin>", line 15, in <module>
NameError: name 'sns' is not defined. Did you mean: 'sys'?
>>> # the list(zip(*shit)) pattern is basically a transposition: looking
>>> # at the last cluster results we have
>>> summary[-1]
[1.9, array([[ 4,  4,  6, 20],
       [ 4, 29,  3, 218]])]
>>> means
[array([1.5, 0.5]), array([0.5, 2.5]), array([7.        , 0.33333333]), array([8., 2.]
>>> mean_cols
[(np.float64(1.5), np.float64(0.5), np.float64(7.0), np.float64(8.0)), (np.float64(0.
>>> data
array([[1, 0],
       [2, 1],
       [0, 2],
       [1, 3],
       [6, 0],
       [7, 0],
       [8, 1],
       [8, 2]])
>>> data_cols
[(np.int64(1), np.int64(2), np.int64(0), np.int64(1), np.int64(6), np.int64(7), np.int

>>> # # other shit
>>> # data_randomly_reordered = data.copy()
>>> # np.random.shuffle(data_randomly_reordered)
>>> # data_opt = np.array([(1, 0), (0, 2), (2, 1), (1, 3), (6, 0), (8, 1), (7, 0), (8
```

3.5 Other shitty python

3.5.1 K-means clustering

Example 3.5.1 (k-means clustering).

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.cluster import vq
import random
import matplotlib.colors as colors
from pathlib import Path
import math
```

```
k = 24
data_fname = "rings_spheres"
data = np.loadtxt("../ClusterData/" + data_fname + ".csv", delimiter=",")
centroids, variance = vq.kmeans(data, k)
fig = plt.figure()
plt.scatter([x[0] for x in data], [x[1] for x in data])
plt.scatter([x[0] for x in centroids], [x[1] for x in centroids])
plt.show()
identified, distance = vq.vq(data, centroids)
# separate clusters in data
clusters = []
for c in range(k):
    clusters.append(data[identified == c])
# plot clusters with colors
init_color_index = random.randint(0, len(colors.cnames))
lc = len(colors.cnames)
color_bag = []
for i in range(math.ceil(k / lc)):
    color_bag = color_bag + list(colors.cnames)[init_color_index:] + list(colors.cnames)[:init_
python_colors = color_bag[0::max(1, lc // k)]
fig1, ax = plt.subplots()
ax.set_facecolor('black')
pt_size = 8
for c in range(k):
    ax.scatter([x[0] for x in clusters[c]], [x[1] for x in clusters[c]],
               c=python_colors[c], s=pt_size)
ax.grid()
plt.show()
p = Path('Figures')
p.mkdir(exist_ok=True)
fig.savefig("Figures/kmeans_" + data_fname + "_centroids.png")
fig1.savefig("Figures/kmeans_" + data_fname + "_clusters.png")
```

```
k = 4
data_fname = "sine_spheres"
data = np.loadtxt("../ClusterData/" + data_fname + ".csv", delimiter=",")
centroids, variance = vq.kmeans(data, k)
fig = plt.figure()
plt.scatter([x[0] for x in data], [x[1] for x in data])
# plt.show()
plt.scatter([x[0] for x in centroids], [x[1] for x in centroids])
plt.show()
identified, distance = vq.vq(data, centroids)
# separate clusters in data
clusters = []
for c in range(k):
    clusters.append(data[identified == c])
# plot clusters with colors
```

```
init_color_index = random.randint(0, len(colors.cnames))
lc = len(colors.cnames)
color_bag = []
for i in range(math.ceil(k / lc)):
    color_bag = color_bag + list(colors.cnames)[init_color_index:] + list(colors.cnam
python_colors = color_bag[0::max(1, lc // k)]
fig1, ax = plt.subplots()
pt_size = 8
for c in range(k):
    ax.scatter([x[0] for x in clusters[c]], [x[1] for x in clusters[c]],
               c=python_colors[c], s=pt_size)
ax.grid()
plt.show()
p = Path('Figures')
p.mkdir(exist_ok=True)
fig.savefig("Figures/kmeans_" + data_fname + "_centroids.png")
fig1.savefig("Figures/kmeans_" + data_fname + "_clusters.png")
```

3.5.2 Hierarchical clustering

Example 3.5.2.

```
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import numpy as np
import scipy.cluster.hierarchy as hy
from scipy.spatial.distance import pdist, squareform

# --------------------------------
# change the folder containing
# the data set here
# --------------------------------
data_folder = "c:/Users/s/Desktop/"
# --------------------------------
# data_fname = "sine_spheres"
data_fname = "sine_spheres_ez"     # easier version of sine_spheres
# data_fname = "ring_spheres"
data = np.loadtxt(data_folder + data_fname + ".csv", delimiter=",")
#
# create a scatter plot the data set
# arguments are the two lists of the 1st
# and the 2nd coordinate
# an optional third argument is a format string
# specifying the lines or markers
# 'ko' stands for black circles
# (see the "Notes" section in
# https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html)
plt.ioff()
pl =  plt.plot([x[0] for x in data], [x[1] for x in data], 'ko')
# pyplot provides a general function to set
# properties of an object, setp
# set the size of the markers to a smaller value
```

```
plt.setp(pl, markersize=1)
# display the plot
plt.show()
# If the data set size is greater than a 20-30,
# the bottom part of the dendrogram tend to become
# cluttered with segments and object labels
# let's sample the data
np.random.seed(2450958)
sam_size = 100
sam_ids = np.random.choice(np.array(data.shape[0]),
                           size=sam_size, replace=False)
# use the sample
act_data = data[sam_ids]
#
# or use the whole data set
# act_data = data
#
# display a plot of the sample
pl =  plt.plot([x[0] for x in act_data], [x[1] for x in act_data], 'ko')
plt.show()
# A distance matrix is a
# symmetric matrix and has a main diagonal of zeros.
# Thus only elements with a row index smaller than
# the column index need be stored.
# pdist computes only such elements of the
# distance matrix of the data and stores them in
# an array of size n(n-1)/2.
distances = pdist(act_data, metric='euclidean')
#
# compute the complete-linkage agglomerative clustering
# the output is a matrix with shape (n-1, 4).
# Entry i stores information on the
# merge operation performed at
# step i: indices of the merged clusters, their distance,
# and the cardinality of the new cluster
tree = hy.linkage(distances, method='single')    # 'complete' 'average' 'ward'
# set a cutoff as a fraction of the maximum
# distance between clusters which have been merged
# for highlighting a partition in the dendrogram
cutoff = 0.25 * np.max(tree[:, 2])
#
# create a list representation of the dendrogram
# having the root at the top.
# - Every upside-down "U" represents a cluster
#   containing all the subclusters that can be reached
#   navigating downwards from it
# - The "U"s of the clusters which were merged
#   are connected to its bottom ends
# - The y-coordinate of its top end
#   is the distance between the two clusters
```

```
#   which were merged
# The color threshold is a y-coordinate value such that
# maximal subtrees with a top end smaller than the
# threshold are coloured. The collection of the
# top clusters of such subtrees form one of the partitions
# generated by the algorithm
dgram = hy.dendrogram(tree, orientation='top', color_threshold=cutoff)
# Make the plot visible
plt.show()
# -------------------------------
# draw the highest level partitions
# induced by the n-tree
# -------------------------------
# compute the encoder of the partition
sam_size = 500 # data.shape[0]
np.random.seed(2450958)
sam_ids = np.random.choice(np.array(data.shape[0]),
                           size=sam_size, replace=False)
act_data = data[sam_ids]
distances = pdist(act_data, metric='euclidean')
tree = hy.linkage(distances, method='single')    # 'complete' 'average' 'ward'
# plot the partitions defined by the largest 30
# merge distances (exclude the final 1-cluster
# partition)
n_part = 30
for cutoff in tree[-n_part:-1, 2]:    # last 50 rows, 3rd column
    # hy.fcluster computes a partition defined by cutoff as
    # follows: exactly the clusters merged at a distance at most cutoff
    # which are not contained in a cluster merged at a distance at most cutoff
    encoder = hy.fcluster(tree, cutoff, criterion="distance")
    print(set(encoder))
    fig3, ax = plt.subplots(facecolor='white')
    ax.set_facecolor('grey')
    ax.set_aspect('equal')
    pt_size = 8
    for i in range(len(act_data)):
        ax.scatter([act_data[i][0]], [act_data[i][1]],
                   c=list(colors.cnames)[encoder[i]], s=pt_size)
    plt.show()
```

3.5.3 Birch

Example 3.5.3.

```
import numpy as np
from sklearn.cluster import Birch
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from itertools import cycle

# -------------------------------
# change the folder containing
```

```
# the data set here
# --------------------------------
data_folder = "c:/Users/s/Desktop/"
# --------------------------------
data_fname = "rings_spheres"
data = np.loadtxt(data_folder + data_fname + ".csv", delimiter=",")

# number of clusters
k = 24
# threshold for assigning objects to clusters
t = 0.02
# model set up
birch = Birch(branching_factor=50, n_clusters=k, threshold=t, compute_labels=True)
# Clustering Feature Tree creation
birch.fit(data)
# plot of the model
labels = birch.labels_
n_clusters = np.unique(labels).size
fig = plt.figure(figsize=(16, 8))
# add subplots in a grid of two side by side
# (a grid with 1 row and 2 columns)
gs = fig.add_gridspec(1, 2)
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[0, 1])
# set up a dict with as many int keys as
# there are clusters, and a colour for each key
# need to use cycle() because the number of clusters may
# exceed the number of colours
# the zip function generates a sequence of n-tuples from
# n lists, the i-th n-tuple contains the i-th elements
# of the lists
cluster_colors = dict(zip(list(range(n_clusters)), cycle(colors.cnames.keys())))
pt_size = 8
# plot the clusters
for k in range(n_clusters):
    index_mask = labels == k
    ax1.scatter(data[index_mask, 0], data[index_mask, 1],
                c=cluster_colors[k], s=pt_size)
# same, but
# without final clustering
# shows all the sub-clusters associated to
# the clustering features in the leaves
# of the tree
birch = Birch(branching_factor=50, n_clusters=None, threshold=t, compute_labels=True)
birch.fit(data)
labels = birch.labels_
n_clusters = np.unique(labels).size
cluster_colors = dict(zip(list(range(n_clusters)), cycle(colors.cnames.keys())))
for k in range(n_clusters):
    index_mask = labels == k
```

```
    ax2.scatter(data[index_mask, 0], data[index_mask, 1],
                c=cluster_colors[k], s=pt_size)
print("Number of leaf CFS:", n_clusters)
plt.show()
```

Capitolo 4

Association rule discovery

Remark 39. **Frequent pattern extraction** from a dataset is a a major issue in data mining, because it is an essential step for extracting correlations and associations

Remark 40. Examples of extracted patterns are the **itemset**, the **subsequence**, and the **substructure**:

- **Set of items** (or itemset): subset of a finite set of items in by a commercial transaction
- **Subsequence**: sequence determined by a subset of the indices of a time sequence of events
- **Substructure**: structure obtained from a structure, typically a graph, by eliminating some of its elements and the relationships between them

Remark 41. We focus with set of items/itemset

Example 4.0.1 (Itemset associations and sales)**.** Eg

- Managing supermarkets involves decisions such as the choice of products to sell, product layout on shelves for profit maximization, choice of special offers, CRM policies, which can be made much more effective and timely by the analysis of the association of items in basket data
- The physical location and the promotion of items that are frequently purchased together affect the number of sales
- The cost for the acquisition of knowledge on items that are frequently purchased together must be justified by a sufficiently large increase in sales revenue. Computer algorithms can be a cost-effective way of acquiring such a knowledge

Remark 42 (Itemset associations: Available data)*.* we have:

- Until the 1990s, the only information available on computers about large-scale purchase transactions was given as cumulative purchases per week, month, or year

- The widespread use of barcode technology has enabled the storing of huge amounts of **basket data**, that is, data on the products contained in every purchase transaction
- In the past, such historic data were not generally on-line, that is, stored in databases on secondary memory devices permanently connected to the computer's input-output channels, because of the limited capability of database management systems to efficiently extract information contained therein

Remark 43 (Association rules example). "In 90% of the transactions in which bread and butter are: purchased, milk is purchased as well.":

- The statement can be concisely written as a **rule**, composed of an **antecedent** ("bread and butter are purchased") and a consequent ("milk is purchased as well") and a **trust** ("90%") expressed as percentage
- The problem is to **design algorithms for processing queries that extract rule sets**
- Example of a query: *Find all the rules with "Diet Coke" as a result.* These rules could suggest a strategy to increase sales of the Diet Coke, for example to make its purchase more probable when one of the antecedent is purchased

Important remark 19 (Model of association rules). Formally:

- Let $\mathcal{I} = I_1, I_2, \ldots, I_m$ a set of **items**
- A **transaction** T is a set of items (representing what customer purchased)
- By **association rule** we mean an implication of the form $A \longrightarrow B$ (not in the strict logical sense, here we have a percentage of B occurring) , where $A, B \subseteq \mathcal{I}$, and $A \cap B = \emptyset$ (clearly if one buy butter it buy butter but that's not interesting)
- Let $\mathcal{D}$ be a database consisting of a set of transactions (basket data). The associative rule $A \longrightarrow B$ has:
  - **support** s in $\mathcal{D}$ if and only if the number of transactions of $\mathcal{D}$ containing $A \cup B$ is a fraction s of the cardinality of $\mathcal{D}$
  - **confidence** c in $\mathcal{D}$ if and only if the number of transactions of $\mathcal{D}$ containing both A and B is a fraction c of the number of transaction of $\mathcal{D}$ containing A

Remark 44. Meaning of support and confidence

- Support and confidence express different aspects of the practical import of a rule
  - Actions prompted by the discovery of a rule with low support cannot be justified on economic grounds, as their impact will be negligible. If such a rule has high confidence, it may have low utility regardless
  - Likewise, rules with low confidence are not reliable, and they may not be useful, whatever their support

- Support and confidence are clearly related to conditional probabilities. If we let $E \Longleftrightarrow$ "T contains A", $F \Longleftrightarrow$ "T contains B" then
- $\mathbb{P}(E \cap F)$ represents **support**
- $\mathbb{P}(F|E) = \frac{\mathbb{P}(E \cap F)}{\mathbb{P}(E)}$ represents **confidence**

Remark 45. Discovery of association rules:

- Given $\mathcal{I}$ and $\mathcal{D}$, we are interested in generating all the rules that are valid in $\mathcal{D}$, which satisfy some measures or constraints of interest
- The *support* must not be less than a threshold `min_sup`
- *Confidence* must not be less than a threshold `min_conf`
- Rules that meet the two contraints above are called **strong**
- *Syntactic constraints* may also be specified, forcing the antecedent to be an element of a given set X , or the consequent to be contained in a given set Y , or both

Remark 46 (Types of association rules). Association rules can be classified according

- **Types of values**: Rules that associate presence or absence of items are *Boolean association rules*. Rules associating numeric attributes are called *quantitative association rules*
- **Dimensions**

  $$age(30:39) \text{ and } salary(42:48) \longrightarrow buy(laptop)$$

  is a *multidimensional* rule, where age, salary, and buy are its dimensions, whereas

  $$buy(laptop) \longrightarrow buy(usb\_key)$$

  is a *one-dimensional* rule

Remark 47. The most famous/easy algorithm to find association rules is the apriori algorithm

Important remark 20. Apriori: Decomposition of the problem:

- Association rules for transactions are *Boolean*, *one-dimensional* rules (dimension name omitted)
- The **Apriori** algorithm discovers Boolean one-dimensional rules
- A set of items is called **itemset**
- An itemset is said **frequent** if the number of transactions that contain it equals at least $min\_sup\,|\mathcal{D}|$
- If $A \longrightarrow B$ is a strong associative rule, as consequence of definition, then $A \cup B$ is a frequent itemset

- Therefore the discovery process consists of two stages
  1. Find **frequent itemsets**: this is the focus
  2. Generate **strong associative rules** from frequent itemsets: this is done by splitting the itemset in two parts finding the alternative "implications" and checking in the data

Important remark 21 (Apriori: Generating frequent itemsets). We have:

- An itemset containing k items is called k-itemset;
- The set of frequent itemsets of cardinality k is denoted by L_k;
- Apriori uses a layered iterative search technique, called **level-wise search**, which generates frequent itemsets in order of increasing cardinality. The generation of all itemsets of cardinality k uses all itemsets of cardinality $k-1$, for $k > 1$.

Remark 48. The Apriori algorithm:

1. Generate all the frequent itemset with just 1 item, L_1, from dataset $\mathcal{D}$
2. set the level k to 2
3. while L_{k-1} is not empty
   (a) Generate L_k using L_{k-1} and $\mathcal{D}$
   (b) Increment k by one
4. Output the union of all the L_k

Most important is 3.1

Join step:

- Generate a set of *candidate* k-itemsets C_k , $k > 1$, defined as

$$C_k = \ell_1 \cup \ell_2 : \ell_1, \ell_2 \in L_{k-1}, |\ell_1 \cap \ell_2| = k-2$$

  where ℓ_1 and ℓ_2 are two frequent itemset with $k-1$ elements and having $k-2$ common elements. C_k is a set of sets.
  We have that C_k is smaller than the set of all k-itemsets, but sufficient to contain L_k, so frequent itemset with k-items are a subset of C_k.[1]

- to generate C_k we procede as follows: if we assume sets $\ell_1, \ell_2 \in L_{k-1}$ are ordered lexicographically (thus $\ell_i = \ell_{i1}\ell_{i2}\dots\ell_{i(k-1)}, then \ell_{ij} < \ell_{i(j+1)}$, $1 \leq j < k-1$) we can compare the elements of two itemsets efficiently in a single pass and C_k can be algorithmically computed as follows (**Join step**):
  - Initially set $C_k = \emptyset$

[1] To prove it assume the opposite, that is, some $\ell \in L_k$ is not in C_k . Pick two distinct $(k-1)$-subsets ℓ_1, ℓ_2 of ℓ, thus $\ell = \ell_1 \cup I_1 = \ell_2 \cup I_2$, with $I_1 \neq I_2$. Then $\ell = \ell_1 \cup \ell_2$. Moreover, there is no item $I \in \ell$ distinct from I_1 and I_2 missing from $\ell_1 \cap \ell_2$, otherwise I would miss from ℓ_1 or ℓ_2. Therefore $|\ell_1 \cap \ell_2| = k-2$. Since by the Apriori property ℓ_1, ℓ_2 are frequent, they are in $L_k - 1$, and ℓ must be C_k, against the assumption.

  - For all ordered pairs of itemsets $\ell_1, \ell_2 in L_{k-1}$
    * If $\ell_{11} = \ell_{21}$ and ... and $\ell_{1(k-2)} = \ell_{2(k-2)}$ and for the last element on the contrary $\ell_{1(k-1)} < \ell_{2(k-1)}$ then we set a new set with as last element the major of the twos $\ell = \ell_{11} \dots \ell_1(k-1)\ell_2(k-1)$
    * Set $C_k = C_k \cup \{\ell\}$

- C_k is then filtered using the Apriori property (**Prune step**):
  - for every $c \in C_k$, if c has a subset of cardinality $k-1$ which is not frequent, c is not frequent either and is removed from C_k.
    The frequencies of the subsets of c are efficiently computable without accessing the database. In fact, all frequent $(k-1)$-itemsets are in L_{k-1} , which is already available when L_k is computed
  - for the remaining itemsets, their cardinalities are computed through accesses to the database. Items with a cardinality lower than threshold $min\_sup\,|\mathcal{D}|$ are deleted

- Once we have generated L_k, the **generation of association rules** is simple.
  For every frequent itemset ℓ, all its nonempty subsets are generated: for each non-empty subset $s \subset \ell$, the rule

$$s \longrightarrow \ell \setminus s$$

  is generated (note that $s \cup (\ell \setminus s) = \ell$, thus the rule satisfies min_supp because ℓ is frequent) when

$$\frac{|\{T \in \mathcal{D} : \ell \subseteq T\}|}{|\{T \in \mathcal{D} : s \subseteq T\}|} \geq min\_conf$$

Example 4.0.2 (Verify this code is correct)**.**

```
def one_itemset_support(d_name):
    with open(d_name, "r") as csv_file:
        d = csv.reader(csv_file, delimiter=',')
        supports = {}
        count = 0
        for t in d:
            count += 1
            for item in t:
                if item not in supports:
                    supports[item] = 1
                else:
                    supports[item] = supports.get(item) + 1
    return supports, count


def prune_itemset(i, itemsets):
    j = 0
    while j < len(i) and i[:j] + i[j+1:] in itemsets:
        j += 1
```

```
    # return true if j exceeds the length minus 1
    return j > len(i) - 1


def itemset_support(i, d_name):
    with open(d_name, "r") as csv_file:
        d = csv.reader(csv_file, delimiter=',')
        support = 0
        for t in d:
            if set(i).issubset(set(t)):
                support += 1
    return support


def apriori(d_name, min_support):
    frequent_itemsets = []
    s, c = one_itemset_support(d_name)    # supports of 1-itemsets, count trans
    abs_min_support = min_support * c
    freq = {(i,) for i in s if s.get(i) >= abs_min_support}
    frequent_itemsets.extend(freq)
    k = 2
    last_freq = freq
    while len(last_freq) > 0:
        # join
        cand = [i1 + (i2[k-2],)
                for i1 in last_freq for i2 in last_freq
                if i1[:k-2] == i2[:k-2] and i1[k-2] < i2[k-2]]
        # prune: infrequent k-1-subset --> infrequent
        pruned = [i for i in cand if prune_itemset(i, last_freq)]
        # check support in dataset
        freq = {i for i in pruned if itemset_support(i, d_name) >= abs_min_support}
        frequent_itemsets.extend(freq)
        last_freq = freq
        k += 1
    return frequent_itemsets

# test
import csv, os

dataset_name = "trans1.csv"
with open(dataset_name, "w", newline='') as f:
    w = csv.writer(f, delimiter=',', lineterminator=os.linesep)
    d = (('a', 'b', 'c'), ('a', 'b'), ('a', 'd'), ('d', 'b'), ('c', 'a'))
    for row in d:
        w.writerow(row)


for supp in (0.0, 0.2, 0.4, 0.6, 0.8, 1.0):
    print("min_supp = ", supp)
    print(ap.apriori(dataset_name, supp))
```

Capitolo 5

Supervised learning

Definition 5.0.1 (Learning)**.** A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E

Design vs. learning *Traditional software* development constructs and tests programs which should behave in accordance with specifications.
However there are *problems where such an approach is unfeasible*(eg when theory/expertise is missing, when the task undergoes frequent changes).
In machine learning:

- algorithms are not designed to perform tasks in class T with high performance P
- algorithms are designed to be able to *learn how to perform the task well* by reading a dataset of *examples*

We will be concerned with learning the **learning task**, that is, learning how to predict future or unseen values from existing known data

Supervised learning Prediction learning as a supervised learning problem:

- **Instance** A multivariate data set of N observations over p **input variables** $X_1, \ldots, X_p$; a multivariate data set of N observations over K **output variables** $Y_1, \ldots, Y_K$.
- **Solution** A function $f(\mathbf{x})$ which predicts the value vector $\mathbf{y}$ of the output variables, given an input vector $\mathbf{x}$ of values of the input variables

Some definition:

- Most often the first p-variate data set is represented as a $N \times p$ **design matrix** $X = [x_{ij}]$, with rows denoted x_i
- The second K-variate data set is represented as a $N \times K$ matrix $Y = [y_{ik}]$, with rows denoted y_i
- Collectively, the rows in $[XY]$ are called **examples**

Training and testing We have:

- Function $f(\mathbf{x})$ is produced by a learning algorithm which takes the examples $[XY]$ as its input, and searches a parametric family of functions $f(\mathbf{x};\boldsymbol{\theta})$ for the $f(\mathbf{x};\boldsymbol{\theta}^*)$ that best fits $[XY]$
  - that is, a $f(x;\boldsymbol{\theta}^*)$ such that $f(x_i^T;\boldsymbol{\theta}^*)$ best approximates y_i^T for $i = 1, 2, \ldots, N$
  - For example, the best approximation is computed by taking into account the differences $y_i^T - f(x_i^T;\boldsymbol{\theta})$
- It is the performance on unseen data that matters in a prediction task. Thus, a **test** data set $[X_{test}Y_{test}]$ is used to compute the actual performance P
- The training examples and the test set must be generated by the same **data generating process**, that is, by the same distribution. Moreover, they must be **independent**.
  This is often done using random splitting of available data.

Example: linear regression

Example 5.0.1 (Linear regression)**.** Task T is the prediction of an output variable as a linear transformation of the inputs

$$\hat{y} = \mathbf{w}^T\mathbf{x}$$

where the *parameters* of the model are the *weights* $\mathbf{w}$.
Learning is the computation of the weights $\mathbf{w}$ from a set of examples $[XY]$, which minimise the mean squared error

$$MSE(w) = \frac{1}{N}\sum_{i=1}^{N}(y_i\mathbf{w}^T x_i^T)^2$$

As performance measure P , the mean squared error over the test set is usually chosen (same formula, different dataset):

$$MSE_{test} = \frac{1}{N_{test}}\sum_{i=1}^{N_{test}}(y_{i,test}\mathbf{w}^T x_{i,test}^T)^2$$

For linear regression, a solution in *closed form* (so there's no need in optimization as above) when X^TX is non-singular is given by the **normal equations**

$$w = (X^TX)^{-1}X^T[y_1 \ldots y_N]^T$$

which results from equating the gradient of the MSE, which is convex, to 0

Underfitting and overfitting we have that:

- training by any method reduces the **training error**. However, it is the *performance on unseen data that matters* in a prediction task

- the **generalisation**, or **test**, error is computed on a set that is drawn from the same distribution as the training set and is independent from it
- The expected test error is greater or equal to the expected training error
- Thus, a good learning algorithm will reduce both the training error and the difference between training and test error
- **Underfitting** is the result of a training which could not achieve a low training error
- **Overfitting** is the result of a training which could not achieve a low difference between test and training error

Model capacity We have

- the **capacity** of a model is its ability to adapt to a broad spectrum of functions
- a *low capacity* model may be prone to underfitting
- a *high capacity* model may achieve a low training error, but may cause overfitting because it can capture random details of the particular training set which increase the test error

classification and regression Supervised learning problems can be divided into:

- **Classification** The predicted variables are qualitative, or categorical
- **Regression** The predicted variables are quantitative

We have that:

- in either case, we denote the inputs by X and the outputs by Y . Individual variables are identified by subscripts: $X_1, \ldots, X_p$ and $Y_1, \ldots, Y_K$
- the number of observations in the data set is denoted by N
- the data set is thus represented by a $N \times p$ matrix $[x_{ij}]$, where x_i is the i-th observation
- the outputs for each observation and output variable are a $N \times K$ matrix $[y_{ik}]$

Classification

Definition 5.0.2 (**Classification**)**.** Assignment of each object of a dataset to a category, or class, chosen from a fixed finite set. The class is an unordered categorical attribute

Classification task in ML and DM consists of **two stages**:

1. **Learning**: A learning algorithm

- accepts as input a training set, in which each object has a defined value for the class attribute; class values are assumed to be correct (in many cases, they are assigned by an expert)
- produces a classifier as output, that is, a mechanism to associate any object with a class

2. **Testing** The classifier is applied to a **test set** of objects with a defined class; a measure of predictive accuracy is derived from the differences between the real class and the class assigned by the classifier, for all objects in the test set. The test set and the training set must be independent

NB: saltato e passato diretto ai trees

Expected Prediction Error Assume a given distribution $F(X, Y)$, where Y is a single response variable and X is a vector of predictors:

-
- The goal is to find a function f such that $f(x)$ is a good prediction of the response Y given values x of the predictors X, that is, one which minimises a prediction error
- The prediction error can be quantified by the expectation of a suitably defined **loss** function $L(Y, f(X))$, the **Expected Prediction Error** (EPE)

$$EPE(f) = \mathbb{E}L(Y, f(X))$$

NB: saltato

Expected Prediction Error in classification Assume Y is a categorical, or class, variable, having range $\mathcal{Y} = \mathcal{Y}_1, \ldots, \mathcal{Y}_K$, where the $\mathcal{Y}_k$ are usually called classes
We need a loss function L defined on pairs of classes that quantifies the penalty for predicting class $\mathcal{Y}_\ell$ when the actual class is $\mathcal{Y}_k$. That is,

$$L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$$

satisfying

$$L(\mathcal{Y}_k, \mathcal{Y}\ell) = \begin{cases} 0 & \text{if } k = \ell \\ \geq 0 & \text{if } k \neq \ell \end{cases}$$

NB: saltato

Minimising classification EPE To minimise the EPE we first express the loss function as a conditional expectation

$$EPE(f) = \mathbb{E}L(Y, f(X)) = \mathbb{E}\sum_{k=1}^{K} L(\mathcal{Y}_k, f(X))\mathbb{P}(\mathcal{Y}_k|X)$$

Observe that the outermost expected value is minimised by choosing the function value $f(x)$, for every value x of X , yielding the smallest value of the conditional expectation (**pointwise** minimisation)

$$f(x) = \arg\min_{y \in \mathcal{Y}} \sum_{k=1}^{K} L(\mathcal{Y}_k, y)\mathbb{P}(\mathcal{Y}_k|X = x)$$

saltato

0-1 loss function A common choice for L is the **zero-one** loss function, assigning unit penalty when $\ell \neq k$
It can be represented as a matrix $\mathbf{1}\mathbf{1}^T - I_K$, in which the (k, ℓ)-entry is $L(\mathcal{Y}_k, \mathcal{Y}_\ell) = 0$ on the main diagonal and $L(\mathcal{Y}_k, \mathcal{Y}_\ell) = 1$ elsewhere

The Bayes classifier We need thus to choose ℓ so that the columnwise summation of the terms $L(\mathcal{Y}_k, y)\mathbb{P}(\mathcal{Y}_k|X = x)$ is minimum **NB**: saltato

$$[\mathbb{P}(\dagger_1|X = x) \quad \dots \quad \mathbb{P}(\dagger_K|X = x)] \begin{bmatrix} 0 & 1 & \dots & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots & \dots & 1 \\ 1 & \vdots & \ddots & \vdots & \ddots & 1 \\ 1 & 1 & \dots & 1 & \dots & 1 \end{bmatrix}$$

Every such summation is of the form

$$1 \cdot \mathbb{P}(\mathcal{Y}_1|X = x) + \cdots + 0 \cdot \mathbb{P}(\mathcal{Y}_\ell|X = x) + \cdots + 1 \cdot \mathbb{P}(\mathcal{Y}_K|X = x)$$

and is minimised by choosing ℓ so that $\mathbb{P}(\mathcal{Y}_\ell|X = x)$ is maximum, yielding thus the Bayes classifier

$$f(x) = argmax_{y \in \mathcal{Y}} \mathbb{P}(y|X = x)$$

Expected Prediction Error in regression Assume now the response Y is numeric **NB**: saltato
The prediction error can be quantified by the expectation of the **squared error loss function**

$$L(Y, f(X)) = (Y - f(X))^2$$

so that the Expected Prediction Error is

$$EPE(f) = \mathbb{E}(Y - f(X))^2$$

introducing conditional expectation

$$= \mathbb{E}\mathbb{E}_{Y|X}((Y - f(X))^2|X)$$

The regression function The outermost expected value is minimised by choosing the function value $f(x)$, for every value x of X , yielding the smallest value of $\mathbb{E}_{Y|X}((Y - f(X))^2|X)$: **NB**: saltato

$$f(x) = \arg\min_c \mathbb{E}Y|X((Y - c)2|X = x)$$

that is, the EPE is minimised pointwise

The requested c is the conditional expectation of Y . The resulting f is called the **regression function**

$$f(x) = \mathbb{E}(Y|X = x)$$

The k-nearest neighbor classifier The true joint distribution of X and Y is unknown. The expressions of the optimal f above are not directly usable for prediction

NB: saltato

- The *k-nearest neighbor* (kNN) rule outputs the value of a function of a set of k objects which are similar to x
- In classification, **majority voting** approximates the Bayes classifier

$$kNN(\mathbf{x}) = \arg\max_{c \in C} \sum_{(\mathbf{x}', y) \in Nk(\mathbf{x})} I(y = c)$$

where C is the class label set, $N_k(\mathbf{x})$ is the set of the k-nearest neighbours of a test object $\mathbf{x}$, and I is the indicator function

NB: saltato

kNN with distance-weighted voting One have that:

- The value of k is critical: for small values, noise objects cause excessive sensitivity; for large values, the neighbourhood may include a majority of objects of another class
- The majority class could be the class of the farthest objects in the neighbourhood, whereas near objects are more reliable predictors
- The inaccuracies due to the criticality of k and the insensitivity to distance may be alleviated by **distance-weighted voting**

$$kNN(\mathbf{x}) = \arg\max_{c \in C} \sum_{(\mathbf{x}', y) \in N_k(\mathbf{x})} \frac{I(y = c)}{d(\mathbf{x}, \mathbf{x}')^2}$$

where d is the dissimilarity function used to compute $N_k(\mathbf{x})$

NB: saltato

kNN for regression The regression function can be approximated by averaging values of Y over a neighbourhood of the input point x

$$kNN(\mathbf{x}) = N_k(\mathbf{x})^{-1} \sum_{(\mathbf{x}', y) \in N_k(\mathbf{x})} y$$

Note that both the kNN classifier and the rule above perform two distinct approximations each:

1. maximisation is approximated by the majority of the data, and expectation is approximated by averaging the data
2. Conditioning is applied to a neighbourhood, since the input point x is most likely not in the data

Capitolo 6

Decision trees

We have that

- A **graph** is a set V of **vertices** and a set E of **edges**, which are pairs of elements of V (eg social networks)
- A **tree** is a special type of graph with no cycles, such that no two pairs exist in E with identical second elements

Example 6.0.1. In fig 6.1 on the left the graph

$$V = \{A, B, C, D, E\}$$
$$E = \{(A, B), (B, C), (C, A), (A, D), (B, D), (D, E), (E, C)\}$$

on the right the tree

$$V = \{A, B, C, D, E\}\, E = (A, B), (B, C), (B, D), (D, E)$$

Definition 6.0.1 (Decision tree)**.** Directed tree such that each vertex which is not a leaf is associated with one condition on the attributes, which must evaluate to either true or false for any object of the dataset, and each leaf node is associated with a class (see fig 6.2)

NB: questi sono poi i classification tree categorici

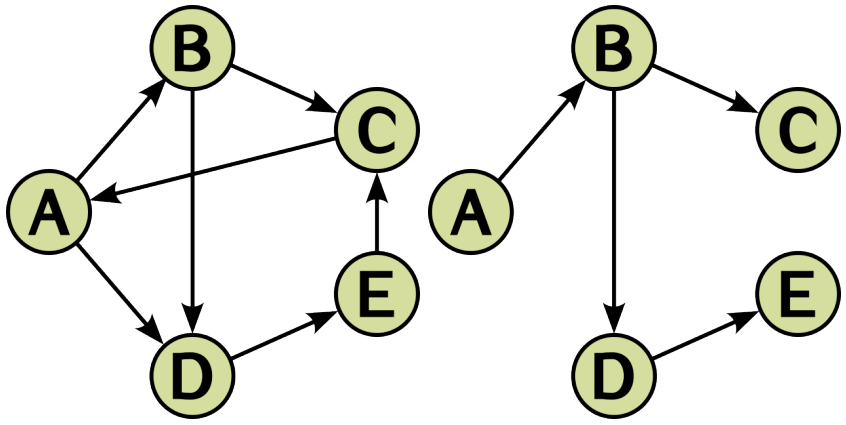


Figura 6.1: Graph and tree

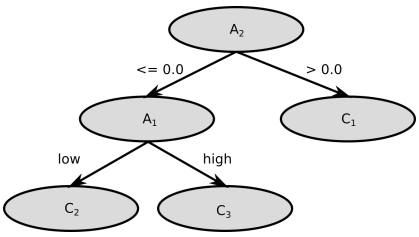


Figura 6.2: Decision tree

Application of a decision tree to a dataset for classification

- For each object of the dataset:
  - Set the root node as the **current node**
  - While the current node is not a leaf node:
    - Calculate the condition associated with the current node
    - Position in the node having as ingoing arc the one associated with the result of the condition and set that node as the current node
- Associate the object with the current node class

Learning a decision tree **Problem**: *Given a dataset D, find in the set of all possible decision trees with conditions defined on the attributes of D, the trees with best performance*:

- an exhaustive search of the best tree, by **generation and test**, is not feasible because the search space is too large
- a family of methods to learn decision trees is based on **greedy search**: the tree is built from the root by repeatedly adding nodes, choosing the attribute and the condition so that the attribution of classes to objects is the most discriminating possible
- a precursor of current methods is **Hunt's algorithm**

Hunt's algorithm We have:

- **Input** Dataset, class set
- **Output** A decision tree

The algorithm is:

- Create the root of the tree and assign the dataset to the root
- Repeat while modifications to the tree are possible
  1. Choose a node
  2. If a unique class is assigned to all the objects of the node (*pure*), assign that class to the node; otherwise

- Choose an attribute and determine a finite set of conditions on it, which must be exhaustive and mutually exclusive (eg age $<$ 5 or age $\geq$ 5): a value of the attribute must make one and only one condition true
- Apply the conditions to each of the objects which are assigned to the node
- Create a child node for each condition
- Assign each node object to the unique child node such that both object and child node verify the same condition

Selection of attributes and conditions

- The selection of the attributes and the conditions on them is a crucial point
- Note that:
  - The attribute space is known: it is the set of attributes of the dataset
  - The space of the conditions must be defined
- A criterion must be chosen for the selection, with the goal of achieving a good predictive performance

Expressing conditions For discrete categorical attributes:

NB: cioè qua non sono solo binary in generale

- **Multiple split**: if attribute A has values $v_1, v_2, \ldots, v_r$, conditions are $A = v_i$ for $1 \leq i \leq r$; no choice is required
- **Set of binary splits**: the set $v_1, v_2, \ldots, v_r$ is partitioned into two subsets V_1 and V_2 , in all possible ways; then, choose the pair of conditions $A \in V_1$, $A \in V_2$; the search space of the choice is generated by varying V_1, V_2

For continuous attributes:

- **Binary split**: choose a pair of conditions of the form $A < v$, $A \geq v$; the search space is generated by varying v
- **Multiple split**: Choose a set of conditions

$$A \leq v_i \quad v_i < A \leq v_{i+1}, i = 1, \ldots, k-1 \quad A > v_k$$

Measures for split selection Goal of the split selection: to create child nodes with the greater possible *homogeneity* of the class attribute
Measurements compare the heterogeneity of the parent node and the child nodes. Possible Measurements of the degree of heterogeneity (c is the number of classes) are

$$entropy = -\sum_{i=0}^{c-1} p_i \log p_i \tag{6.1}$$

$$Gini = 1 - \sum_{i=0}^{c-1} p_i^2 \tag{6.2}$$

$$Classificationerror = 1 - \max\{p_i : i = 0, 1, 2, \ldots, c-1\} \tag{6.3}$$

In case of two class Comparison of heterogeneity measures in fig6.3

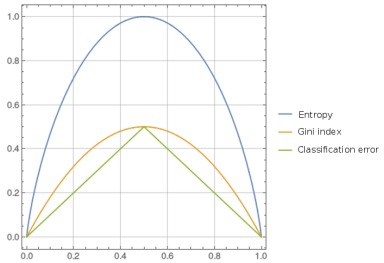


Figura 6.3: Entropy, gini and classification error

	Predicted class = 1	Predicted class = 0
Real class = 1	$f_{11}(TP)$	$f_{10}(FN)$
Real class = 0	$f_{01}(FP)$	$f_{00}(TN)$

Tabella 6.1

Confusion matrix and metrics for evaluation In the confusion matrix:

- f_{ij} denotes the number of objects of class i which were predicted as being of class j
- The number of correct predictions is $f_{11} + f_{00}$
- The number of incorrect predictions is $f_{10} + f_{01}$
- Performance metrics:

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{f_{11} + f_{00}}{f_{11} + f_{00} + f_{10} + f_{01}}$$

$$\text{Error rate} = \frac{\text{Number of wrong predictions}}{\text{Total number of predictions}} = \frac{f_{10} + f_{01}}{f_{11} + f_{00} + f_{10} + f_{01}}$$

$$\text{True positive rate (TPR)} = \frac{TP}{TP + FN}$$

$$\text{False positive rate (FPR)} = \frac{FP}{FP + TN}$$

$$\text{Precision (p)} = \frac{TP}{TP + FP}$$

$$\text{Recall (r)} = \frac{TP}{TP + FN}$$

Decision trees in Machine Learning and Data Mining Several decision tree learning algorithms have been proposed:

- CLS (Concept Learning System, Hunt, Marin, & Stone, 1963)
- ID3 (Quinlan, 1979)
- C4.5 (Quinlan) evolution of ID3
- C5.0 / See5 (Quinlan) commercial version

- CART (Breiman, Friedman, Olshen, & Stone) Binary trees
- ID3 / C4.5 / C5.0 / See5 and CART have been developed independently

ID3 Decision trees

- Initially, the tree contains only one node, representing all the training data. If all the data belong to the same class, the node becomes a leaf and is labeled with the class
- Otherwise, a entropy-based heuristic measure, the **information gain**, is used to select the attribute that best separates the data into classes. The attribute becomes the test attribute of the node. A new node, and a directed edge into it, are created for each possible test result and the data are partitioned accordingly among the new nodes. If, for a test result, there is no object in the training data that satisfies it, create a leaf labeled with the most frequent class in the data
- The algorithm is called recursively for each new non-leaf node and the corresponding set of training data. The test attribute of the current node becomes ineligible for selection for all new nodes and all their descendants.
- The recursion ends:
  1. When all of the training data of a node belongs to one class
  2. When no attributes remain to further partition the data. In this case the node is converted into a leaf node and the more frequent class of the node is chosen as the node class

Selection of the test attribute

- The selection of the attribute to associate with a node is based on an **attribute selection measure**, or **measure of the goodness of split**
- ID3 uses a measure of *information gain*
- The aim is to minimize the amount of information needed to classify the training data of the node, in order to reduce the number of tests to classify an object and simplify the tree
- The measure is based on entropy

Attribute selection measure

- Let S be a set of s objects
- Let the class attribute have m distinct values C_i and s_i be the number of objects for which the value of the class attribute is C_i
- The expected value of the information needed to classify S is

$$I(s_1, \ldots, s_m) = -\sum_{i=1}^{m} p_i log2(p_i)$$

  where p_i is the probability that an arbitrary object belongs to the C_i class and it is set equal to s_i/s

- Statistically, $I(s_1, \ldots, s_m)$ represents a measure of the heterogeneity of S with respect to the class attribute
- Let A be an attribute with v distinct values $\{a_1, \ldots, a_v\}$
- A can be used to partition S into v subsets $\{S_1, \ldots, S_v\}$, where S_j contains the objects in S with value a_j in A (that is, if A is selected as an attribute for a current node associated to set S, the S_j would correspond to the created outgoing edges)
- If s_{ij} is the number of objects of the C_i class in the collection S_j , the entropy that would result from partitioning according to A is

$$\varepsilon(A) = \sum_{j=1}^{v} \frac{s_{1j} + \cdots + s_{mj}}{s} I(s_{1j}, \ldots, s_{mj})$$

where

$$I(s_{1j}, \ldots, s_{mj}) = -\sum_{i=1}^{m} p_{ij} \log_2(p_{ij}) = \sum_{i=1}^{m} \frac{s_{ij}}{|S_j|} \log_2 \left(\frac{s_{ij}}{|S_j|} \right)$$

- Choosing attribute A to partition the set S changes the expected entropy $I(s_1, \ldots, s_m)$ to the expected entropy $\mathbb{E}(A)$
- Therefore the **information gain** determined by the choice of A to test the current node can be considered as the expected value of the reduction of entropy caused by knowledge of the value of the A attribute

$$Gain(A) = I(s_1, \ldots, s_m) - \varepsilon(A)$$

- At each step the algorithm chooses A in order to maximize $Gain(A)$, that is, the algorithm tends to choose A so to reduce the class heterogeneity of the parts of S, as determined by the values a_j of A

AdaBoost We have that

- **Ensemble learners** are methods which employ multiple learners to improve generalization ability
- Let $\mathcal{X}$ be the space of training examples and $\mathcal{Y} = \{-1, +1\}$ the set of class labels
- Assume a training set $\mathcal{D} = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$, with $\mathbf{x}i \in \mathcal{X}$, $y_i \in \mathcal{Y}$, for $i = 1, 2, \ldots, m$, and that a base learning algorithm $\mathcal{L}$ is available
- AdaBoost initialises a weight distribution by assigning equal weights $1/m$ to all training examples. Let $D^{(1)}$ be such weight distribution
- AdaBoost executes a sequence of T rounds. At round t, it generates a weak learner $h_t : \mathcal{X} \to \mathcal{Y}$ from algorithm $\mathcal{L}$ using the weight distribution $D(t)$. Then, it tests h_t on the training examples and increases the weights of the incorrectly classified examples, generating a weight distribution $D^{(t+1)}$ for the next round; it also assigns a weight α_t to the generated classifier h_t

Algorithm 1 AdaBoost$(\mathcal{D}, \mathcal{L}, T)$

for $i \in \{1, \ldots, m\}$ **do**
  $D_i^{(1)} = 1/m$
end for
for $t = 1, 2, \ldots, T$ **do**
  $h_t = \mathcal{L}(\mathcal{D}, D^{(t)})$ // generate weak learner
  $\epsilon_t = \frac{1}{m} \sum_{i=1}^{m} D_i^{(t)} I(h_t(\mathbf{x}_i) \neq y_i)$ // learner error
  $\alpha_t = \frac{1}{2} \log \frac{1-\epsilon_t}{\epsilon_t}$ // learner weight
  $D_i^{(t+1)} = \frac{D_i^{(t)}}{Z_i} \cdot \begin{cases} e^{-\alpha_t}, & \text{if } h_t(\mathbf{x}_i) = y_i \\ e^{\alpha_t}, & \text{if } h_t(\mathbf{x}_i) \neq y_i \end{cases}$ // update weights
end for
return $H(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})\right)$

- Finally, the output classifier $H(\mathbf{x})$ is the weighted majority voting of the T weak learners, using weights α_t , $t = 1, 2, \ldots, T$

The adaboost pseudocode is

where Z_i is such that $\sum_{i=1}^{m} D_i^{(t+1)}/Z_i = 1$. In the code above, the base learner uses the weight distribution $D^{(t)}$ directly. Alternatively, the training examples can be sampled according to the weight distribution

6.1 Python exercises

Example 6.1.1 (Exploring overfitting depending on tree depth).

```
Esempio 6.1.1 (Esplorazione overfitting al variare della profondità).
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

# generate a dataset with y = x + random
n_points = 1000
x = np.linspace(-.5, .5, n_points) # 1000 equispaced points between -.5 and .5
x_feat = x.reshape([len(x), 1]) # x è un array x_feat è una matrice (array 2dim) (vettore verti
y = x + np.random.randn(n_points)*0.1 #randn is N(0,1)

# different depths for decision trees will be tried and evaluated
depth_list = list(range(1, 12)) # depths from 1 to 11
mse_depths = [] # where we collect mses for each dept
prediction_depths = []
cross_val_n_steps = 10 # number of k in k-fold cv?

for depth in depth_list:
    sq_err_sum = 0.0 # initialize the sum of squared error
    # initialize prediction_sum to np array of zeros: prediction in the test
    # group in cross validation? eg x_feat.shape[0]/cross_val_n_steps) is 100 here
    prediction_sum = np.zeros(int(x_feat.shape[0]/cross_val_n_steps))
    # cross validation cycle
    for i in range(cross_val_n_steps):
```

```
        # create indexes for train and test using division remainder %
        # this way the 1, 11, 21 will be in group 1 (divided by 10
        # they give 1) and 1 will be used as test in the second
        # iteration (first is 0) while train in all the others
        test_ind = [a for a in range(n_points) if a%cross_val_n_steps == i]
        train_ind = [a for a in range(n_points) if a%cross_val_n_steps != i]
        # define test and training
        x_train = x_feat[train_ind]
        x_test = x_feat[test_ind]
        y_train = y[train_ind]
        y_test = y[test_ind]
        # train trees of various depths and accumulate the errors
        tree_mod = DecisionTreeRegressor(max_depth=depth)
        tree_mod.fit(x_train, y_train)
        prediction = tree_mod.predict(x_test)
        e = y_test - prediction
        # update errors and prediction: add the error for this fold to
        # the total cv error (referring to a certain depth)
        sq_err_sum += np.average(e * e)
        prediction_sum += prediction

    # compute the mean squared error per cv fold, referring to a certain depth
    mse_depths.append(sq_err_sum/cross_val_n_steps)
    prediction_depths.append(prediction_sum/cross_val_n_steps)

# plot error per depth
plt.plot(depth_list, mse_depths)
plt.xlabel('Depth')
plt.ylabel('MSE')
plt.title('MSE vs. depth - ' + str(n_points) + 'points')
plt.savefig('tree-overfitting-' + str(n_points) + '.png', dpi=600)
plt.show()

# as depth increase first the mse decrease (before 3/4 underfitting)
# then it start increas for depth > 5 (overfitting): optimal level is
# 4 of depth
```

Example 6.1.2 (Random forest with wine example)**.**

```
import csv
import numpy as np
from sklearn.tree import DecisionTreeRegressor
from numpy.random import choice # new for sampling
from math import sqrt
import matplotlib.pyplot as plt

# csv raw importing
data = []
labels = []
with open("winequality-red.csv") as f:
    reader = csv.reader(f, delimiter=';')
```

```
    header = next(reader)
    for row in reader:
        # import the first as floats
        data.append([float(v) for v in row[:-1]])
        # import the last which is the class/label
        labels.append(int(row[-1]))

# numpy translation
x = np.array(data)
y = np.array(labels)
n_rows, n_cols = x.shape

# Split into training and test
n_train = int(n_rows * 0.7) # 70% of the observations
train_ind = choice(range(n_rows), n_train, replace=False) # like sample without replacement
test_ind = [i for i in range(n_rows) if i not in train_ind]
x_train, y_train = x[train_ind], y[train_ind]
x_test, y_test = x[test_ind], y[test_ind]

# random forest setup
max_n_trees = 100
n_attr = 4 # number of variables considered for each tree
depth = 12 #
n_bag_samples = int(len(x_train) * 0.5) # n of each bootstrap sample (1/2 of the total)

# empty list for models and predictions
models = []
predictions = []

# choose the optimal number of trees for the ensemble
for i in range(max_n_trees):
    # given a certain number of tree, get the
    # 1) row/obs indexes for the training dataset
    i_bag = choice(range(len(x_train)), n_bag_samples, replace=True)
    # 2) the choosen variables
    i_attr = choice(range(n_cols), n_attr, replace=False)
    # So get the bootstrap sample
    x_train_rf = x_train[i_bag] # select the units
    x_train_rf = x_train_rf[:, i_attr] # select the features
    y_train_rf = y_train[i_bag]
    # in the test just select the variables
    x_test_rf = x_test[:, i_attr]
    # create the tree (DecisionTreeRegressor) and append it to the
    # models list
    models.append(DecisionTreeRegressor(max_depth=depth))
    # train the last added model
    models[-1].fit(x_train_rf, y_train_rf)
    # do the predictions and append them
    pred = models[-1].predict(x_test_rf)
    predictions.append(pred)
```

```
rmse = []
bagging_predictions = []
# for each tree we end appending one rmse
for i in range(len(models)):
    # initialize computed prediction as structure of zeros following
    # the shape of predictions[0]
    pred = np.zeros_like(predictions[0])
    for j in range(i):
        pred += predictions[j] / float(i+1)
    bagging_predictions.append(pred)
    e = y_test - pred
    rmse.append(sqrt(np.average(e * e)))


# display number of trees of the forest and rmse relation (in this
# case we used a single depth)
n_models_seq = [i+1 for i in range(len(rmse))]
plt.plot(n_models_seq, rmse)
plt.xlabel("Number of decision trees")
plt.ylabel("RMSE")
plt.title("RMSE vs. ensemble size, depth " + str(depth) )
plt.ylim((0.0, max(rmse)))
plt.savefig("wine-quality_rf_" + str(depth) + ".png", dpi=600)
plt.show()
# number of trees going from 1 to 100: the rmse go slow fast and then
# has a plateau

# check
min(rmse)
```

Capitolo 7

Neural networks

7.1 Introduction

Neuron is a cell with input and output; brain is made of billions of thos cells connected with each other. We take another approach to the introduction by looking at boolean operation as function

Boolean operations and planes The `or` Boolean operation, as a function

$$or : \{0,1\} \times \{0,1\} \longrightarrow \{0,1\}$$

is defined by a table bearing some resemblance to that of addition, restricted to the same domain $\{0,1\} \times \{0,1\}$

```
or |0 1          + |0 1
---|---         ------
 0 |0 1          0 |0 1
 1 |1 1          1 |1 2
```

Can we find a function $z = ax + by + c$ with the same values as `z = or(x, y)` on its domain? No, because 3 outputs are 1, and the only plane containing them is $z = 1$, incompatible with the 4th output 0

Regression of Boolean operations? However we could resort to regression to approximate `or`, $z = or(x, y)$. The normal equations give

```
x   <- c(0,0,1,1)
y   <- c(0,1,0,1)
zor <- c(0,1,1,1)
(mod <- lm(zor ~ x + y))
## Call:
## lm(formula = zor ~ x + y)

## Coefficients:
## (Intercept)            x            y
##        0.25         0.50         0.50
```

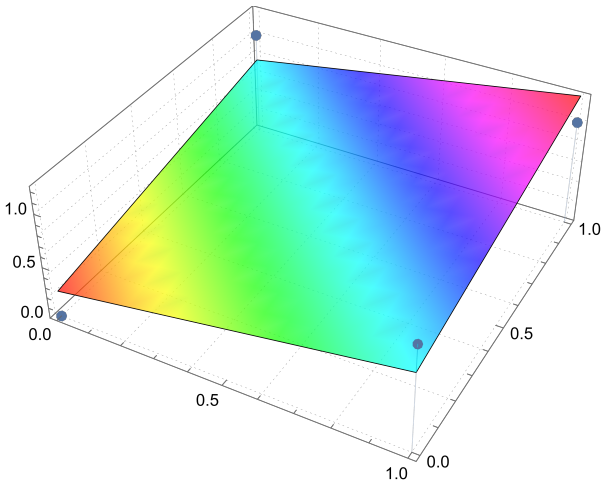


Figura 7.1: or

$$\hat{or}(x, y) = \frac{1}{4} + \frac{x}{2} + \frac{y}{2}$$

depicted in fig 7.1

A closest, or rounding, criterion would compute the output correctly.

```
> predict(mod)
   1    2    3    4
0.25 0.75 0.75 1.25
> round(predict(mod))
1 2 3 4
0 1 1 1
```

xor Again, no plane can contain the whole graph of xor. Now however the normal equations give

$$\hat{or}(x, y) = \frac{1}{2}$$

depicted in fig 7.2. No uniform closest criterion can predict the output.

Step back: or Addition reflects the overall behaviour of the `or` function, but it is too large at $(1, 1)$. It suffices to process its output through a simple "clipping", or **thresholding**, function.

$$f(x, y) = Positive(x + y)$$

$$Positive(u) = \begin{cases} 1 & \text{if } u > 0; \\ 0 & \text{otherwise} \end{cases}$$

in figure 7.3.

The *generalisation* with weights for x and y, and a threshold value for the clipping function, is the **McCulloch-Pitts** (MCP) **neuron**.

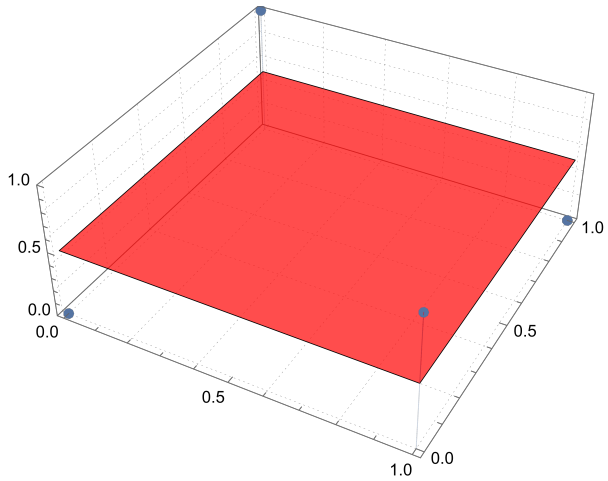


Figura 7.2: xor

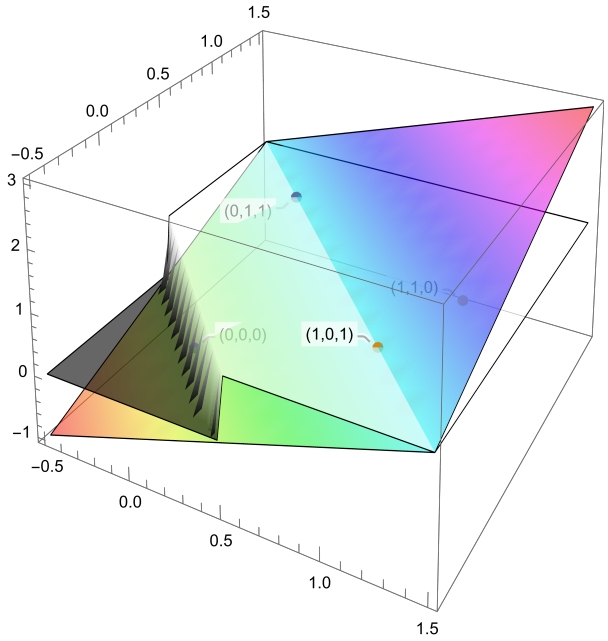


Figura 7.3: stepbackor

The McCulloch-Pitts (MCP) neuron It is a very simplified representation of neuron as linear model with threshold consisting of:

- a weighted summation function which linearly combines inputs in $x_i \in \{0, 1\}$, with weights in $w_i \in \{-1, 1\}$
- a thresholding function which evaluates to 1 if its argument exceeds a threshold ϑ, and to 0 otherwise

Let $\mathbf{x} = [x_1, \ldots, x_p]^T$, and $\mathbf{w} = [w_1, \ldots, w_p]^T$; the output $f(\mathbf{x})$ of the unit at input $\mathbf{x}$ is

$$f(\mathbf{x}) = H(\mathbf{w}^T\mathbf{x} - \vartheta)$$

where H is a unit step function

$$H(u) = \begin{cases} 1 & \text{if } u \geq 0; \\ 0 & \text{otherwise} \end{cases}$$

Perceptron Let's view the MCP neuron as a binary classifier, on a space of real valued features:

- the search space $f(\mathbf{x}; \boldsymbol{\theta})$ is the set of all possible functions which are 1 on one side, and 0 on the other side, of the half-spaces separated by a $(p-1)$-dimensional boundary hyperplane (Eg: for functions of two variables, the two half-planes separated by a line)
- thus, if a training algorithm learns parameters $\boldsymbol{\theta} = (w_1, \ldots, w_p, \vartheta)$ defining such a hyperplane, the classifier is completely defined

There are actually several iterative learning algorithm which can be adopted and performs multiple passes over the training set:

- they use the disagreement between prediction and actual class to direct its search for the optimal parameters.
- if the examples are *linearly separable*, the algorithm returns the correct parameters

Perceptron learning The situation:

- **Input**: training set on p features and a class in $\{0, 1\}$, a learning rate parameter $r \in (0, 1]$
- **Output**: parameters $\mathbf{w}$ of a $(p-1)$-dimensional hyperplane

The perceptron learning algorithm is:

- initialise the parameters $\mathbf{w}, \vartheta$ randomly
- repeat
  - for each example $\mathbf{x}$ a class prediction c is computed for $\mathbf{x}$, using the current parameter values and the formula $H(\mathbf{w}^T\mathbf{w} - \vartheta)$

- if the actual class k of $\mathbf{x}$ differs from the predicted c:
  - spiegazione anno scorso: the non-class part of $\mathbf{x}$ is first multiplied by the learning rate r then added to the parameters if $k = 1$ (thus if they differ we have that here predicted $c = 0$ and we increase the parameters) or subtracted from the parameters if $k = 0$ (actual prediction $c = 1$).
  - spiegazione quest'anno: 1 is appended to the p features of $\mathbf{x}$, and the resulting vector is multiplied by the learning rate r, then added to the parameters, if $k = 1$, or subtracted from the parameters, otherwise

Perceptron learning in Python

```
import numpy as np

def train_perceptron(data, r, param, iters):
    for i in range(iters):
        for x in data:
            x_inp = np.hstack((np.array(1), x))[:-1]
            pred = np.heaviside(np.inner(param, x_inp), 1)
            param = param + r * (x[-1] - pred) * x_inp
    return param
```

dove:

- `iters` let be the number of iteration we set to end the algorithm
- `x_inp` è ottenuto ponendo un 1 ad inizio vettore e togliendo l'ultimo valore (che è la classe effettiva; sull'aggiungere 1 perché lui dice per i bias dei cazzi suoi), `hstack` serve per concatenare
- con heaviside la definizione di numpy è

```
                    /  0,   if x < 0
heaviside(x, h0) = | h0,   if x == 0
                    \  1,   if x > 0
```

  quindi qui di fatto è come la funzione identita $I(x \geq 0)$e dunque era $\vartheta = 0$, sto coglione inutile

xor as a composition

Remark 49. ad ora abbiamo visto come creare un `or`, mentre l'`and` è una semplice moltiplicazione.
However it remains the xor problem: there is no McCulloch-Pitts neuron which contains the whole graph of `xor`, because no line can separate the 0 and 1 output values (fig 7.2). The perceptron learning algorithm *cannot* learn the `xor` function, because the sequence of output parameters converges only if the examples are linearly separable.

Note however that, to obtain `xor` from `or`, we only have to mask out the 1 value at $(1, 1)$, that is, to `and` the table 0 1

```
  |0 1
--|---
0 |1 1
1 |1 0
```

with `or`'s output. As that is the table of `not(and(x, y))`, `xor(x, y)` is equivalent to `and(or(x, y), not(and(x, y))`.

MCP neurons for and, or, not Recalling the previous expression for `or`, to get MCP neuron we must adjust the intercept of the linear part, because the unit step function H is 1 at 0.
MCP neuron for `and` is similar, with a smaller intercept.
Finally `not` is obtained by reversing the argument's axis. So remembering

$$f(\mathbf{x}) = H(\mathbf{w}^T\mathbf{x} - \vartheta)$$
$$H(u) = \begin{cases} 1 & \text{if } u \geq 0; \\ 0 & \text{otherwise} \end{cases}$$

we can obtain logical operator as

$$\begin{aligned} \texttt{and}(x, y) &= H(x + y - 2) \\ \texttt{or}(x, y) &= H(x + y - 1) \\ \texttt{not}(x) &= H(-x) \end{aligned}$$

which are depicted in figure 7.4.
Finally if we substitute the obtained McCulloch-Pitts neuron expressions in `xor`, we get

$$\begin{aligned} \texttt{and}(\texttt{or}(\texttt{x},\texttt{y}),\texttt{not}(\texttt{and}(\texttt{x},\texttt{y}))) &= H(H(x + y - 1) + H(-H(x + y - 2)) - 2) \\ &\overset{(1)}{=} H(H(x + y - 1) - H(x + y - 2) - 1) \end{aligned}$$

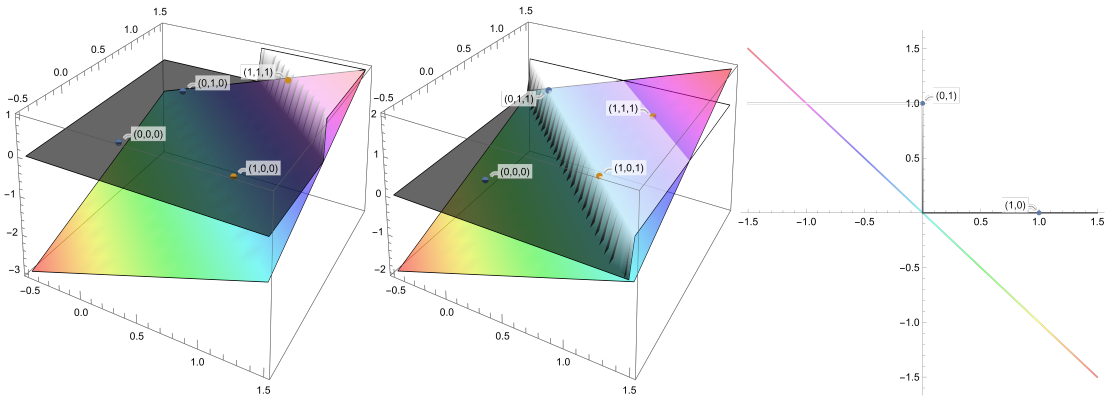
where in (1) we used that $H(-x) = 1 - x$ for $x \in \{0, 1\}$. All the final shit is represented with two plots in fig 7.5

A MCP feedforward network: diagram A MCP feedforward network is the first example of neural network. In figure 7.6 how neural network is represented graphically for the `xor` developed above:

- X_1 and X_2 are the input
- Y is the output
- Z_1 and Z_2 are the mittle nodes (`or` and `and` respectively)
- -1 and -2 tra cerchi verdi are the dummy variables used for biases (testuali parole)
- i numerini attaccati alle rette sono i weights per il calcolo del valore successivo

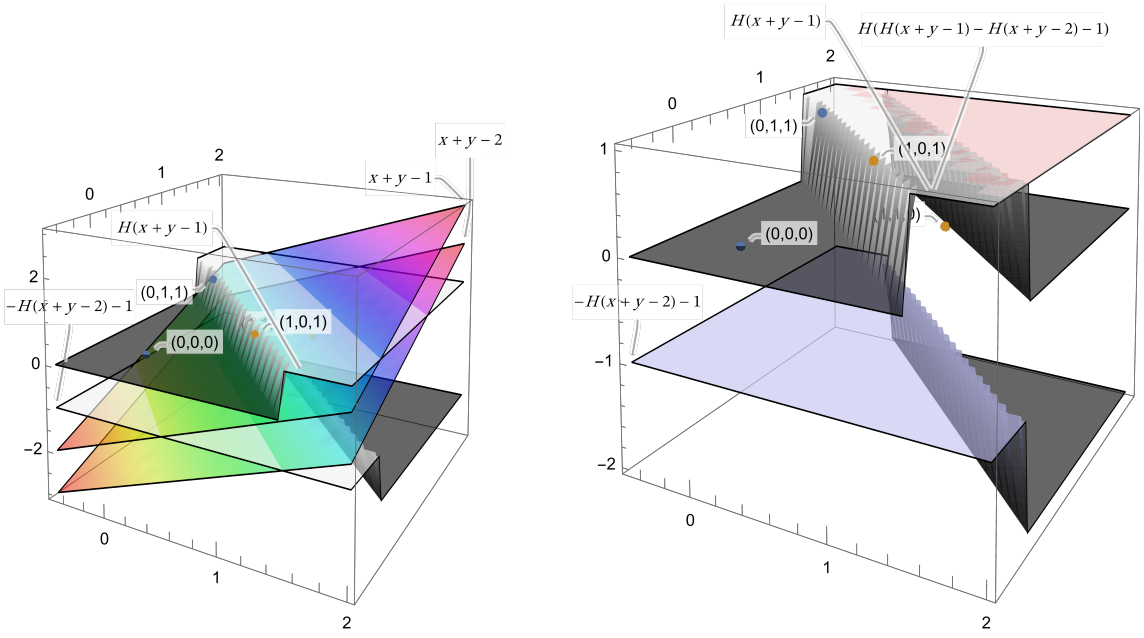
ad esempio:

$$\begin{aligned} Z_1 &= 1 \cdot X_1 + 1 \cdot X_2 - 1 \\ Z_2 &= 1 \cdot X_1 + 1 \cdot X_2 - 2 \\ Y &= 1 \cdot Z_1 - 1 \cdot Z_2 - 1 \end{aligned}$$



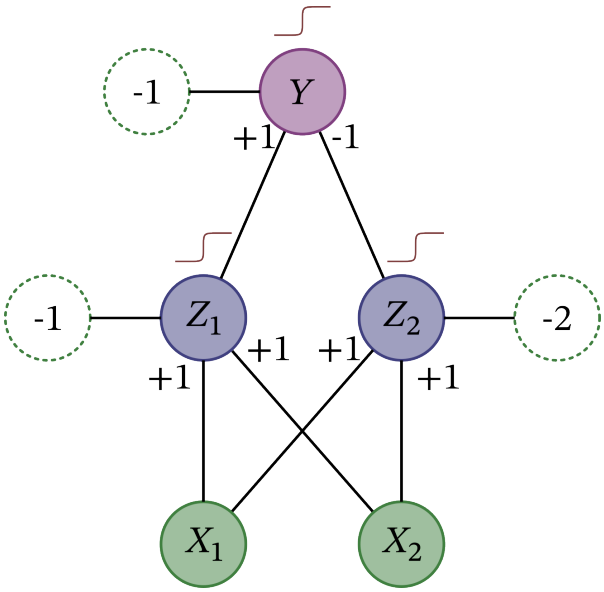
(a) (b) (c)

Figura 7.4: MCP neurons for `and` (a) `or` (b) e `not` (b)



(a) (b)

Figura 7.5: MCP feedforward



(a)

Figura 7.6: `xor` Feed diagram

Linear transformation A more concise expression can be obtained if we consider the two planes as the vector output of a linear transformation, represented by matrix W , of the vector $[1\ x\ y]^T$, followed by the nonlinear stage H , operating componentwise. The linear part, as a column vector in x and y, is

$$W \begin{bmatrix} 1 \\ x \\ y \end{bmatrix} = \begin{bmatrix} -1 & 1 & 1 \\ -2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \end{bmatrix} = \begin{bmatrix} -1+x+y \\ -2+x+y \end{bmatrix}$$

Interestingly multiplying W for the input vectors (where we add a first row of ones) gives the images of the examples in transformation W as columns

$$W \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 1 \\ -2 & -1 & -1 & 0 \end{bmatrix} = U$$

Non separability and effect of non linearity U is a matrix of 4 column vectors, all located on a single line (see figura 7.7 In the figure, plot markers are the outputs of xor). The two central vectors are the images of $[101]^T$ and $[110]^T$, corresponding to [01] and [10] in the `xor` table. Thus, the `xor` values corresponding to the 4 vectors are [0110]: linear transformation W has mapped a non-separable problem into *another non-separable* one.
The subsequent applicaton of H , however, skews the vectors, and makes them linearly separable, because there are only 3 distinct vectors, and they are not aligned.

$$H(U) = H\left(\begin{bmatrix} -1 & 0 & 0 & 1 \\ -2 & -1 & -1 & 0 \end{bmatrix}\right) = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In this case, a regression plane fits exactly (fig 7.8)

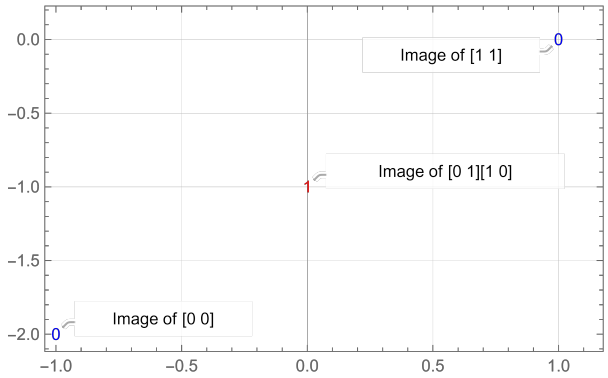
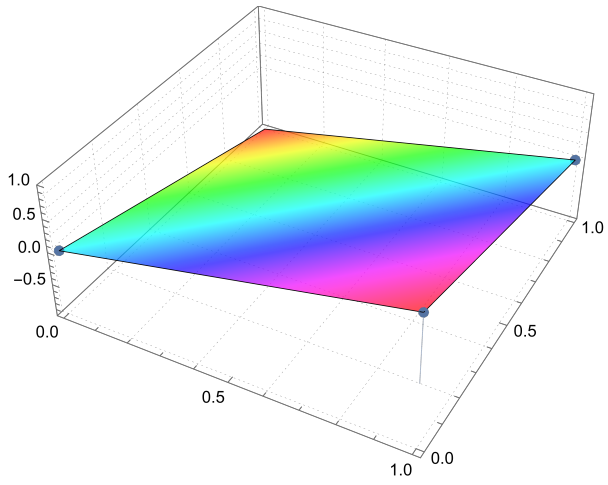
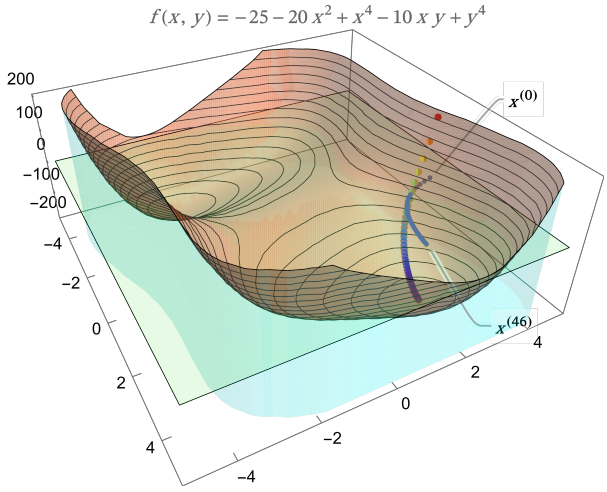


Figura 7.7: non separability



(a)

Figura 7.8: exact fit



(a)

Figura 7.9: gradient descent

7.2 Gradient descent

Remark 50. Gradient descent is a well studied method to minimise a differentiable multivariate function $f(x_1, \ldots, x_n)$.

Definition 7.2.1. The **gradient** of f is the vector function of its partial derivatives

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right)$$

Important remark 22. $\nabla f(\mathbf{x})$ points in the direction of maximum steepness of f at $\mathbf{x}$.

In essence, gradient descent exploit this fact by starting at some $\mathbf{x}(0)$ and repeatedly modifying the point recursively going in the *opposite direction*

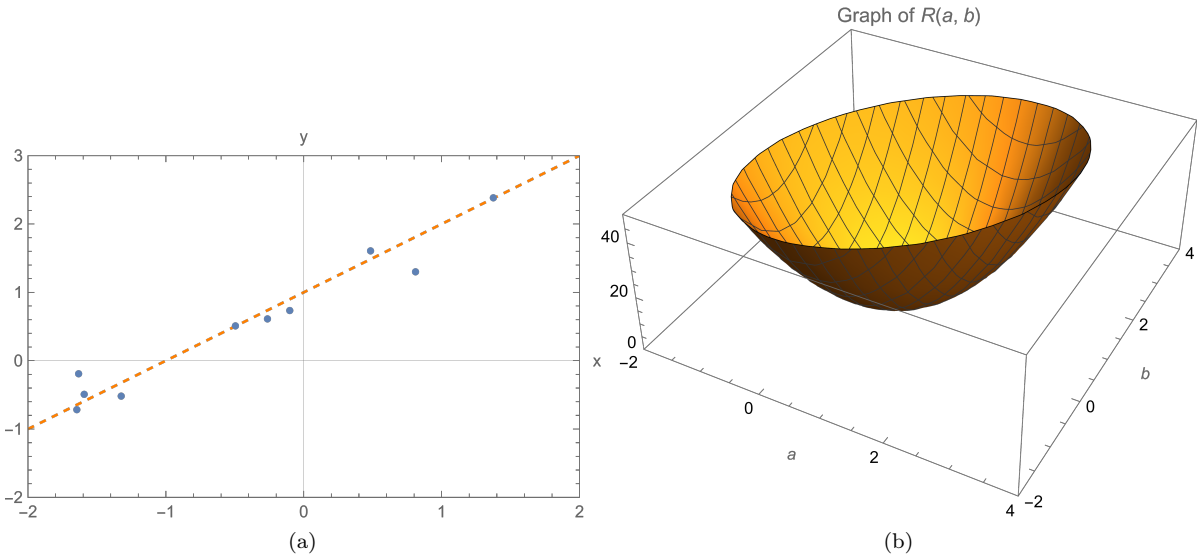
$$\mathbf{x}(r+1) = \mathbf{x}(r) - \gamma_r \nabla f(\mathbf{x}(r))$$

where γ_r can be chosen at every iteration, thereby computing a vector sequence $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(r)}, \ldots$ which converges to a vector $\mathbf{x}^{(*)}$ which minimises the function locally, under assumptions on f.

Example 7.2.1. In fig 7.9 gradient descent with start vector $\mathbf{x}(0) = [0.874.0]^T$ and constant $\gamma = 0.001$, stopped after 46 iterations. The output sequence is in blue on the $z = 0$ plane, and the corresponding sequence on the function's graph is in rainbow colours.

7.2.1 Esempi credo e roba introdotta ultimo anno

Univariate regression Let us review univariate regression for a set of pairs $\{(x_i, yi) : i = 1, 2, \ldots, N\}$: hypothesis is $\hat{y} = ax + b$ while sum of square errors



(a) (b)

Figura 7.10: error surface

is

$$R(a,b) = \sum_{i=1}^{N}(y_i - ax_i - b)^2 = a^2\sum_{i=1}^{N}x_i^2 + 2ab\sum_{i=1}^{N}x_i + Nb^2 - 24\sum_{i=1}^{N}x_iy_i - 2b\sum_{i=1}^{N}y_i - \sum_{i=1}^{N}y_i^2$$

Error surface in fig 7.10

The Gradient is

$$\frac{\partial R}{\partial a} = 2\sum_{i=1}^{N}(ax_i + b - y_i)x_i$$

$$\frac{\partial R}{\partial b} = 2\sum_{i=1}^{N}(ax_i + b - y_i)$$

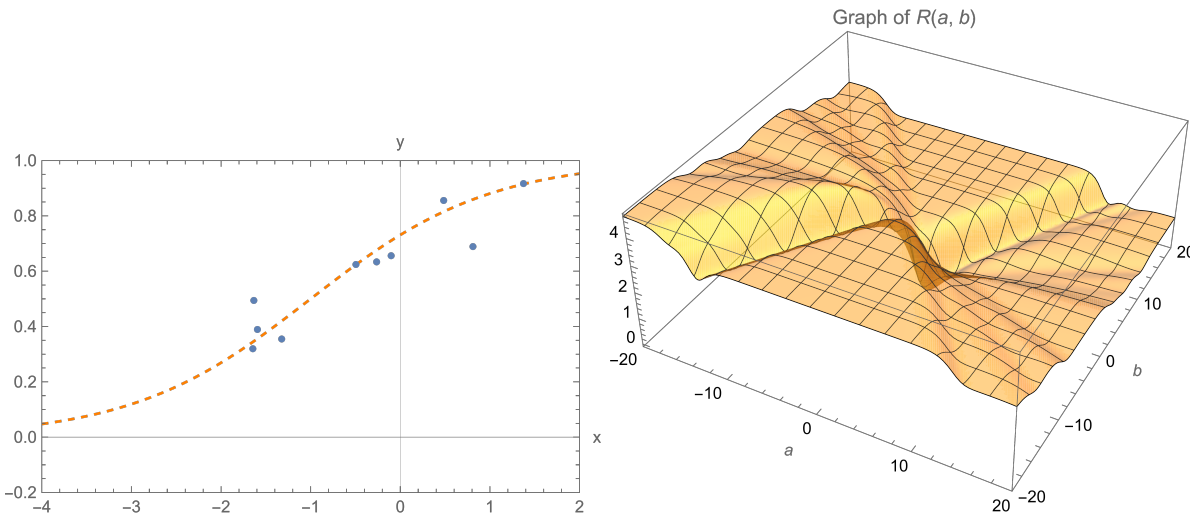
Logistic function univariate regression As a comparison, consider the simplest neural network model, with just one hidden layer and an identity output layer. The hypothesis is

$$\hat{y} = \frac{1}{1 + e^{-(ax_i+b)}}$$

THe sum of square errors

$$R(a,b) = \sum_{i=1}^{N}\left(y_i - \frac{1}{1 + e^{-(ax_i+b)}}\right)^2$$

The model has only two parameters, thus the error function can be plotted as an error surface. We use the x-values of the regression example and generate



(a) (b)

Figura 7.11: logistic error surface

the y-values as

$$y = \frac{1}{1 + e^{x+1}} + \epsilon$$

where $\epsilon \sim \mathcal{N}(0, 0.1)$

Logistic function error surface The error surface has an apparent global minimum, however it also reveals two regions in which the gradient is small, and generally, that the gradient is uneven 7.11

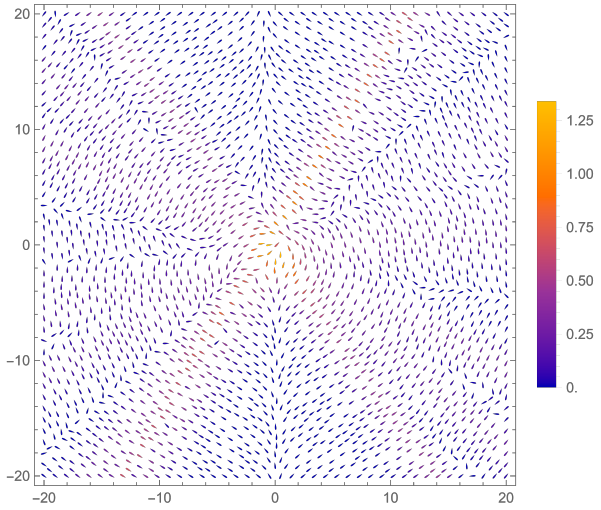
Logistic function gradient field Let us plot the gradient field. Every drop-shaped marker has the direction of the gradient at the point and a colour proportional to its norm (see legend).7.12

Logistic function gradient field - inset The following is an inset of the central region. Even at this scale, the gradient field has a fairly complex structure

7.13

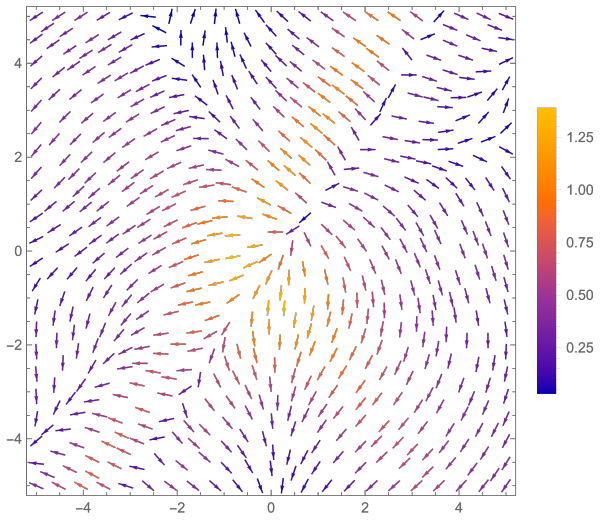
7.2.2 Back on track with NN

Parametric network Let's represent our example network with symbolic weights, as a matrix W for the middle, or **hidden**, layer, and a vector **b** for the output layer, and symbolic activation functions, σ and g. We will also write column vectors **Z** and **X** for the hidden and input variables, and $\boldsymbol{\sigma}$ as a vector function, applying σ componentwise. (equazione sotto e figura 7.14)



(a)

Figura 7.12: gradient field



(a)

Figura 7.13: gradient field

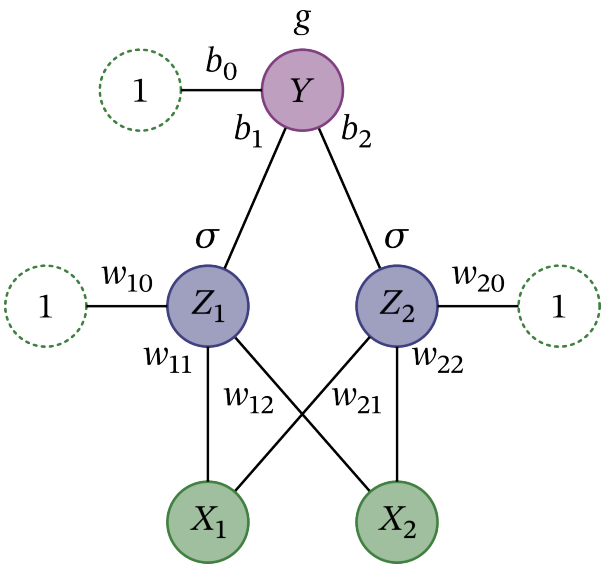


Figura 7.14: paramewtric network

$$\mathbf{X} = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}, \mathbf{Z} = \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix}, \sigma(\cdot) = \begin{bmatrix} \sigma(\cdot_1) \\ \sigma(\cdot_2) \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

Derivatives of the loss Gradient descent is used to *descend a loss function*, as a function of the weights. Here the parameters we have to choose are 9 (that is the component of $\mathbf{W}$ and $\mathbf{b}$). When the number of nodes/layes increase the number of parameters increase exponentially.
For our example, we assume squared error (between actual and predicted values). Let's find the first derivatives of the loss

$$R = (y - f(\mathbf{X}))^2$$

for a training example

- with respect to b_j we have

$$\frac{\partial R}{\partial b_j} = \frac{\partial}{\partial b_j}(y - f(\mathbf{X}))^2 = -2(y - f(\mathbf{X}))g'\left(\mathbf{b}^T \begin{bmatrix} 1 \\ \mathbf{Z} \end{bmatrix}\right) Z_j$$

- with respect to w_{jk}, on the other hand we have

$$\begin{aligned} \frac{\partial R}{\partial w_{jk}} &= -2(y - f(\mathbf{X}))g'\left(\mathbf{b}^T \begin{bmatrix} 1 \\ \mathbf{Z} \end{bmatrix}\right) b_j \frac{\partial}{\partial w_{jk}} \sigma\left(W \begin{bmatrix} 1 \\ \mathbf{X} \end{bmatrix}\right) \\ &= -2(y - f(\mathbf{X}))g'\left(\mathbf{b}^T \begin{bmatrix} 1 \\ \mathbf{Z} \end{bmatrix}\right) b_j \sigma'\left(W \begin{bmatrix} 1 \\ \mathbf{X} \end{bmatrix}\right) X_k \end{aligned}$$

We want to use this shit for our gradient descent; the interesting fact here is that the

$$-2(y - f(\mathbf{X}))g'\left(\mathbf{b}^T \begin{bmatrix}1\\ \mathbf{Z}\end{bmatrix}\right)$$

part is common between them

Propagation The common factor in the derivatives allows to rewrite them as:

$$\frac{\partial R}{\partial b_j} = \delta Z_j, \quad \frac{\partial R}{\partial w_{jk}} = s_j X_k,$$

with

$$\delta = -2(y - f(\mathbf{X}))g'\left(\mathbf{b}T \begin{bmatrix}1\\ \mathbf{Z}\end{bmatrix}\right)$$
$$s_j = \delta b_j \sigma'\left(W \begin{bmatrix}1\\ \mathbf{X}\end{bmatrix}\right)$$

they can be computed proceeding "backwards" from the output layer: δ is computed first, and propagated to the s_j . Then the derivatives are computed simply by multiplication.

Learning with back-propagation

Remark 51. The learning algorithm computes R_i, δ_i and s_{ji} for every i-th example, and descends the cumulative squared error $R = \sum_{i=1}^{N} R_i$.

Important remark 23 (Back-propagation algorithm). Repeat until convergence:

1. Given $\mathbf{W}, \mathbf{b}$ (initialized randomly to small values) compute the outputs $\hat{f}(\mathbf{x}, i = 1, 2, \ldots, N$
2. Compute δ_i and substitute its value for δ_i in s_{ji}, $i = 1, 2, \ldots, N$
3. Compute the derivatives and the next iterate of the weights by the gradient descent equations

$$b_j^{(r+1)} = b_j^{(r)} - \gamma_r \sum_{i=1}^{N} \left(\frac{\partial R_i}{\partial b_j^{(r)}}\right), \quad w_{jk}^{(r+1)} = w_{jk}^{(r)} - \gamma_r \sum_{i=1}^{N} \left(\frac{\partial R_i}{\partial w_{jk}^{(r)}}\right)$$

where γ_r is the learning rate

Single hidden layer back propagation network Generalising further, we obtain one of the simplest neural network model, the **single hidden layer back propagation** network suitable as regression or K-class classification model. We have

- 1 *input layer* with $X_1, \ldots, X_p$ inputs
- 1 *hidden layer* (so called because its output is not observable, applies a linear transformation to the inputs and an activation function σ, obtaining hidden variables $Z_1, \ldots, Z_M$)

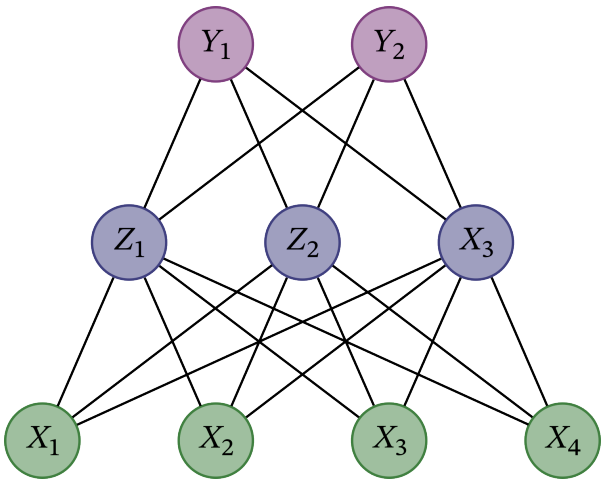


Figura 7.15: Single hidden layer network diagram

- 1 *output layer* with $Y_1, \ldots, Y_K$ outputs, applies a linear transformation to the hidden layer variables and an output function

In classification, output unit k models the probability of class k. Class target measurements are binary variables in $\{0, 1\}$

Example 7.2.2. In figure 7.15 : connection diagram for an example of single hidden layer network with $p = 4$ inputs, $M = 3$ hidden variables, and $K = 2$ outputs. Weights are a straightforward generalisation. Note that the output vector b is an output matrix B, if two or more output nodes exist.
The number of parameters here (consider the rette e che ciascun tondo intermedio ha il parametro di penalizzazione/bias) will be $(4 * 3 + 3) + (3 * 2 + 2) = 23$

Network input-output function If we re-write for the last neural network we get

$$\mathbf{Z} = \begin{bmatrix} \sigma(a_{10} + a_{11}X_1 + \cdots + a_{1p}X_p) \\ \vdots \\ \sigma(a_{M0} + a_{M1}X_1 + \cdots + a_{Mp}X_p) \end{bmatrix}$$

$$\mathbf{Y} = \begin{bmatrix} g1(b_{10} + b_{11}Z_1 + \cdots + b_{1M}Z_M) \\ \vdots \\ gK(b_{K0} + b_{K1}Z_1 + \cdots + b_{KM}Z_M) \end{bmatrix}$$

which can be written more compactly as

$$\mathbf{Z} = \boldsymbol{\sigma}([\mathbf{a}A][1\mathbf{X}^T]^T)$$
$$\mathbf{Y} = \mathbf{g}([\mathbf{b}B][1\mathbf{Z}^T]^T)$$

setting $A = [a_{mj}] \in \mathbb{R}^{M \times p}$, $B = [b_{km}] \in \mathbb{R}^{K \times M}$, $\mathbf{a} = [a_{10} \ldots a_{M0}]^T$, $\mathbf{b} = [b_{10} \ldots b_{K0}]^T$, $\boldsymbol{\sigma}(\mathbf{T}) = [\sigma_1(\mathbf{T}) \ldots \sigma_K(\mathbf{T})]^T$, $\mathbf{g}(\mathbf{T}) = [g_1(\mathbf{T}) \ldots g_K(\mathbf{T})]^T$

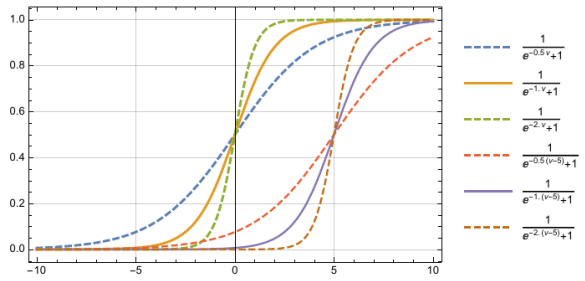


Figura 7.16: sigmoid

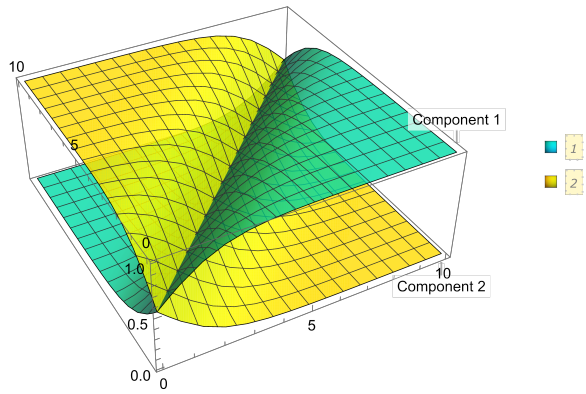


Figura 7.17: Components of the softmax function for K = 2.

Activation function: sigmoid The traditional choice for σ is the sigmoid function $1/(1+e^{-v})$, or its scaled and shifted variant $1/(1+e-h(v-v_0))$

Output function The output function is usually set differently for regression and classification problems

- g_k is the identity for regression. Thus, the output layer is purely linear

$$g_k(\mathbf{T}) = T_k$$

- g_k is often the **softmax function** for classification

$$gk(\mathbf{T}) = \frac{e^{Tk}}{\sum_{\ell=1}^{K} e^{T\ell}}$$

If Figure 7.17: a Softmax function: example Components of the softmax function for $K = 2$.

Capacity and number of units: examples Consider a single hidden layer network with two inputs and one output:

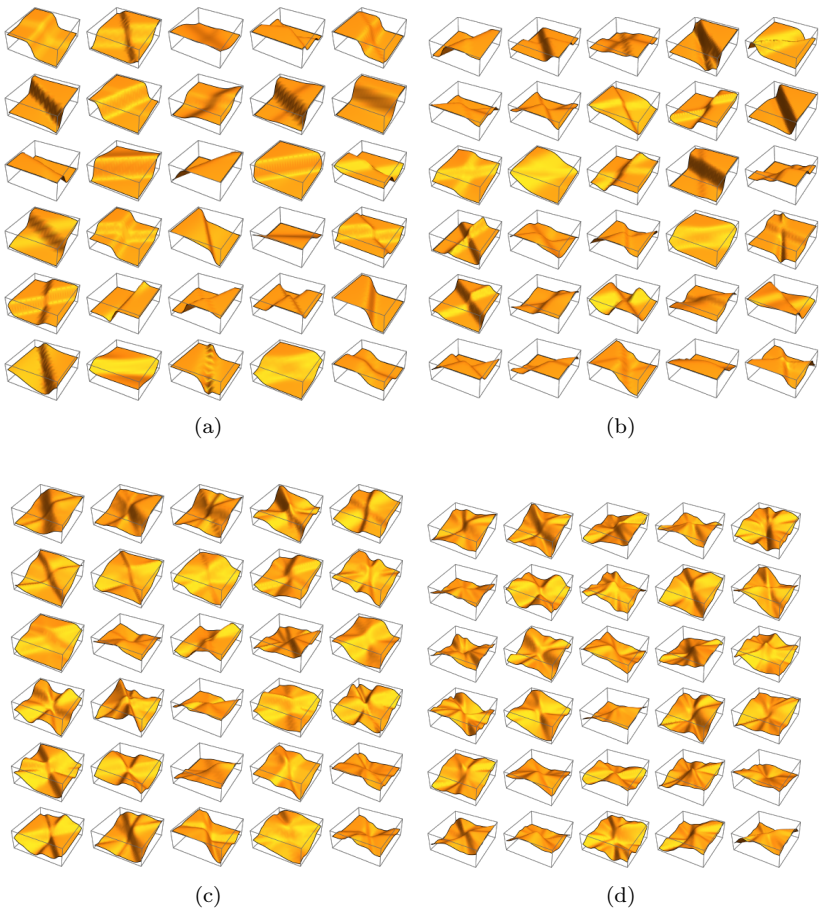


Figura 7.18: 2 (a), 3 (b), 10 (c) and 100 (d) hidden units

- The number of units in the hidden layer determines the number of parameters: Each additional unit contributes with 3 additional parameters (two weights for the inputs, plus the intercept)
- Thus, intuitively, the higher the number of units, the higher the model's capacity
- Let us plot the input-output function of 30 networks with random parameters and increasing number of hidden units

In fig:

- 7.18 2 hidden units,

Loss function we have

- training has to learn $M(1+p)$ weights $\mathbf{a}$, A for the hidden layer and $K(1+M)$ weights $\mathbf{b}$, B for the output layer

- we assume the **squared error loss** as loss function for *regression*

$$R(\mathbf{a}, A, \mathbf{b}, B) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - fk(x_i))^2$$

- **cross-entropy** for *classification*

$$R(\mathbf{a}, A, \mathbf{b}, B) = \sum_{k=1}^{K} \sum_{i=1}^{N} y_{ik} - log f_k(x_i)$$

  with $\arg\max_k f_k(x)$ to choose the class

Deep networks While the simple single hidden layer network above has high *capacity* (fit different functions), in the last ten years state-of-the-art prediction performance has been achieved by networks with more than one hidden layer and diversified connection schemes and activation functions:

- in **feedforward** networks the output vector of one layer is the only input vector of the next one. Layer $\ell > 0$ computes

$$\mathbf{z}^{(\ell)} = g^{(\ell)}(W^{(\ell)}\mathbf{z}^{(\ell-1)} + \mathbf{b}^{(\ell)})$$

  and layer 0 is the input vector, $\mathbf{z}(0) = \mathbf{x}$. The number of layers is the **depth** of the network

- in **recurrent** networks **feedback** connections also send the output to the input

Here the term **architecture** refers to the number of layers, the number of units in each layer, and the connections of the output of the units to the inputs of other units

Why deep? The fundamental **universal approximation theorem** implies that a feedforward network with one hidden layer with a sigmoid activation function and a linear output layer can approximate any Borel measurable function between finite-dimensional spaces up to any approximation, *if the width is unbounded*.
However, attention has shifted away from single hidden layer networks because later studies have shown that:

- the width of such a network can be infeasibly large in the worst case

- families of functions which can be represented efficiently by a deep network require an exponential number of hidden units in a single hidden layer network

- in addition, in some applications apriori knowledge may suggest that factors of variation can be recursively described in terms of other simpler factors

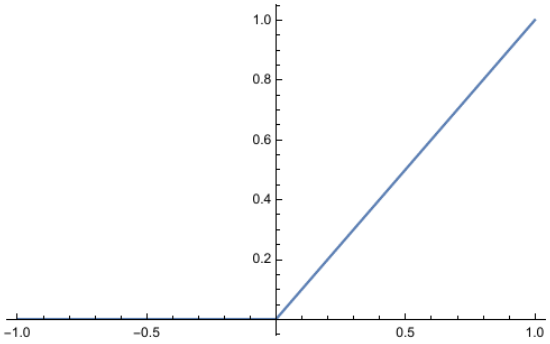


Figura 7.19: relu

Rectified Linear Unit (ReLU) Sigmoid is still used but has drawbacks (linear neighborhood of zero, very steep which is not good for gradient descent). Rectified Linear Unit is an alternative became extremely popular: it's a simple **piecewise linear** function, that is zero for negative values and the identity for positive ones. It is widely used as activation function in deep networks (fig 7.19)

$$g(z) = \max\{0, z\} = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

Some features:

- Relu has unit, thus large, derivative, when $z > 0$, thus optimisation is easier in that range
- However, when the value of ReLU is zero for some examples, its gradient is also zero. Thus, gradient descent cannot learn from those examples. We can fix this however

A parametric piecewise linear family which includes ReLU as special case is

$$g(z, \alpha) = \max\{0, z\} + \alpha \min\{0, z\}$$

Notable choices for $\alpha_i : -1$ gives **absolute rectification**, equivalent to $g(z) = |z|$, employed in object recognition from images. $0 < \alpha_i \ll 1$ gives **leaky ReLU** When α_i is learnable one obtains **parametric ReLU**

Other activation functions There are also other activation functions. For example

1. **Maxout**: vector $\mathbf{z}$ is divided into groups of k values. Let $\mathbb{G}^{(i)}$ denote the indices of the units for group i.
   The maxout function is

   $$g(z)_i = \max_{j \in \mathbb{G}^{(i)}} z_j$$

   A maxout unit is piecewise linear and can approximate any convex function well, if k is large

2. **Radial Basis Function (RBF)** defined as

$$g(z) = e^{-\frac{1}{\sigma_i^2}\|W_{:,i}-\mathbf{z}\|^2}$$

if $\|W_{:,i} - \mathbf{z}\| \to \infty$ $g(z) \to 0$ thus hard to optimize

7.3 Python

Example 7.3.1 (Spyral)**.** Take this shit muthafucka

```
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

# generazione dati
num_classes = 2
pars = np.arange(0, 2*np.pi, 0.1)
rads = np.arange(0, 3, .1)
class_0 = np.array([(r * np.sqrt(t)*np.cos(t), r*np.sin(t))
                    for t in pars for r in rads if r > 1 and r <= 2])
class_1 = np.array([(r * np.sqrt(t)*np.cos(t), r*np.sin(t))
                    for t in pars for r in rads if r <= 1 or r > 2])
fig = plt.figure()
ax = plt.axes()
ax.scatter(class_0[:,0], class_0[:,1], s=.1)
ax.scatter(class_1[:,0], class_1[:,1], s=.1)
# putting dataset together
x = np.vstack([class_0, class_1])
y = np.array([0]*class_0.shape[0] + [1]*class_1.shape[0])

# we permute the data for some reason using np.arange (equivalent of
# range) in choice
n = x.shape[0]
perm_ind = np.random.choice(np.arange(0, n), n, replace=False)
x = x[perm_ind]
y = y[perm_ind]
# to perform classification in keras we need to convert the class
# variable to categorical type
y = keras.utils.to_categorical(y, num_classes)

# Now we split the dataset in training and testing part
# we keep for train the 70% chosen randomly
train_mask = np.random.binomial(1, .7, n).astype('bool')
test_mask = np.logical_not(train_mask)
x_train, y_train, x_test, y_test = x[train_mask], y[train_mask], x[test_mask], y[test_mask]
```

```
# We now create a model neural network
model = Sequential()
# inside dense one has to input how many units??, while in activation
# the activation function, input shape specifies the data structure (2 variable)
model.add(Dense(512, activation='relu', input_shape=(2, )))
# creating other layers, we don't need to specify the input shape animore
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(8, activation='relu'))
# last output units are number of classes, softmax normalize the outuptu
model.add(Dense(num_classes, activation='softmax'))


# compile the model: as loss we adopt the entropy to minimize
model.compile(loss = keras.losses.categorical_crossentropy,
              optimizer = keras.optimizers.Adadelta(learning_rate=0.01),
              metrics = ['accuracy'])
# fit
model.fit(x_train, y_train,
          validation_data=(x_test, y_test),
          epochs=300,   # no rule of thubbs: with 300 it will converges
          batch_size=10 # boh non c'ho voglia
          )

# computed predictions: argmax because?
model.predict(x_test)
y_hat = np.argmax(model.predict(x_test), axis=-1)

# the first occurrence
class_0_ind = np.where(y_hat == 0)[0]
class_1_ind = np.where(y_hat == 1)[0]

# plot
fig = plt.figure()
ax = plt.axes()
ax.scatter(x_test[class_0_ind, 0], x_test[class_0_ind, 1])
ax.scatter(x_test[class_1_ind, 0], x_test[class_1_ind, 1])
```

Capitolo 8

Convolutional NN

Convolution of sequences Remember $\{a_n\}$ denotes a real sequence $a_0, a_1, \ldots$. The **convolution** of $\{a_n\}$ and $\{b_n\}$ is the real sequence whose m-th term is defined by:

$$c_m = \sum_J a_j b_{m-j}$$

Note that:

- c_m is weighted sum of consecutive elements preceding b_m
- Thus, the sums for c_m and c_{m+1} share many terms
- Convolution has generally the effect of *smoothing* the sequences

In data processing, the sequence of data values is called the **signal**, whereas the other sequence is called the **kernel**

Example 8.0.1. Signal: two consecutive square pulses. Kernel: sequence of values from a Gaussian function in fig 8.1

The convolution operation The **convolution operation** *between functions* (instead of discrete sequences), that is between an input function $x(t)$ and a kernel function $w(t)$ is the integral with similar definition:

$$(x * w)(t) = \int_{-\infty}^{+\infty} x(\tau) w(t - \tau)\, \mathrm{d}\tau$$

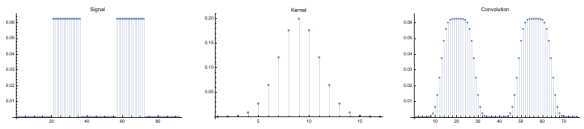
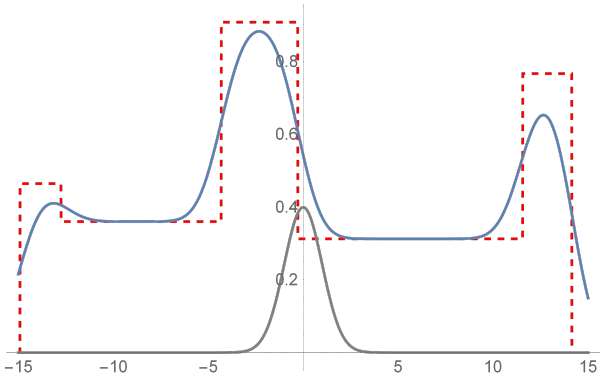


Figura 8.1: Convolution example



(a)

Figura 8.2: Convolution operation

Convolution has a *smoothing effect*: eg if we consider the convolution

$$\int_{-\infty}^{\infty} x(\tau)\frac{e^{-(t-\tau)^2/2}}{\sqrt{2\pi}}\,\mathrm{d}\tau$$

in fig 8.2 input x has the red dashed graph, kernel w has the grey graph for $t = 0$)

Discrete convolution The discrete convolution is the series

$$(x * w)(t) = \sum_{n=-\infty}^{\infty} x(n)w(t-n)$$

and, for functions of two arguments

$$(x * w)(u, v) = \sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty} x(m, n)w(u-m, v-n)$$

Example 8.0.2. Convolve a noisy time series with a monotonically decreasing kernel which averages recent values of the series to give a less noisy current measurement in fig 8.3 (series: red, kernel: grey, convolution: blue)

Convolution in neural network computations We have

- Input is finite. All functions can be set to zero outside a finite range of indices, and the summations are finite sums $\sum_{m=m'}^{m''}\sum_{n=n'}^{n''}$
- In addition, convolution is commutative, thus

  $$(x * w)(u, v) = \sum_{m}\sum_{n} x(u-m, v-n)w(m, n)$$

  which is computationally more straightforward because the kernel w has smaller support than the input

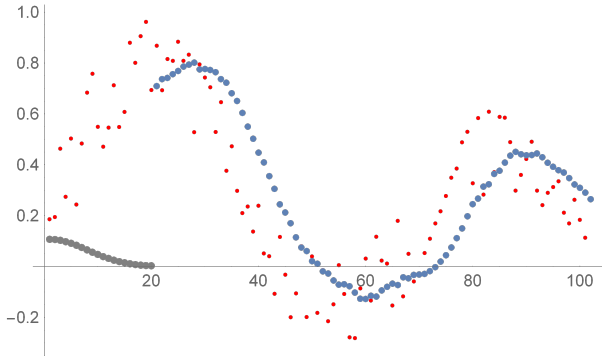


Figura 8.3: Discrete convolution example

- A similar operation which can replace convolution is **cross-correlation**

$$(x * w)(u, v) = \sum_m \sum_n x(u + m, v + n) w(m, n)$$

Example 8.0.3 (2-dimensional example)**.** In figure 8.4. As a matrix-vector multiplication:

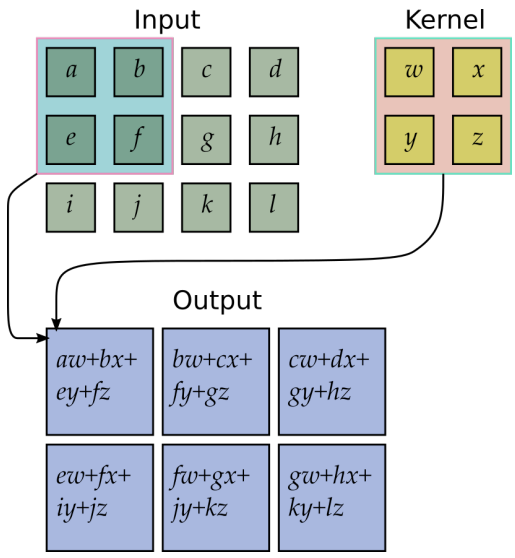
- If we represent the input as a vector, the cross-correlation above can be equivalently expressed as a matrix-vector multiplication, in which the matrix is very sparse (that is, it has many 0 entries), and has few distinct values
- The matrix is sparse because the kernel is small with respect to the input (fig 8.5)

Convolutional NN: Sparse interaction This "bring us" to the idea of sparse interaction (cit.)

Sparse interaction: If the kernel support is smaller than the input, only a few units contribute to the computation of a unit of the next layer: (fig 8.6)

- (a) **sparse**: 3 units affect y_3
- (b) **nonsparse**: *all* units affect y3
- A unit influences only few units of the next layer sparse: z3 affects 3 units
- nonsparse: z_3 affects all units
-

Although direct connections are sparse, in a deep network a unit is indirectly connected to all or most of the input layer



(a)

Figura 8.4: 2-dimensional example

$$\begin{bmatrix} w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w & x & 0 & 0 & y & z & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & w & x & 0 & 0 & y & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \end{bmatrix} = \begin{bmatrix} aw + bx + ey + fz \\ bw + cx + fy + gz \\ cw + dx + gy + hz \\ ew + fx + iy + jz \\ fw + gx + jy + kz \\ gw + hx + ky + lz \end{bmatrix}$$

(a)

Figura 8.5: Convolution matrix

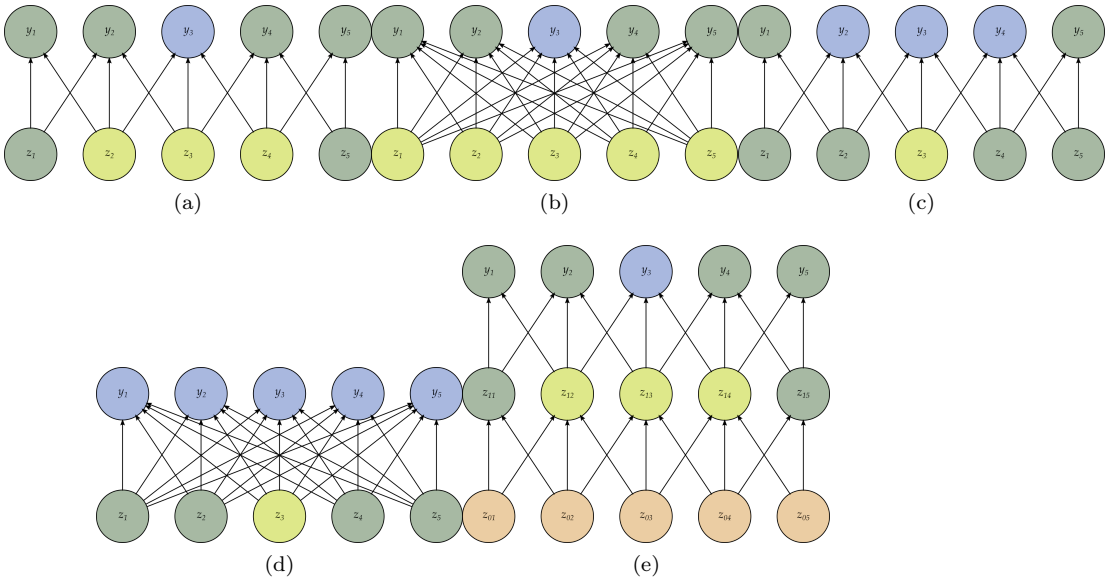


Figura 8.6: Sparse interaction

Parameter sharing **Parameter sharing** In the full product of the weight matrix $W^{(\ell)}$ by input $\mathbf{z}^{(\ell-1)}$, every entry is used exactly once. With convolution, every value of the kernel is used for all the values of $\mathbf{z}$:

- Thus, in a network with dense matrix multiplication, all $m \times n$ parameters which control the processing of the output of a n-unit layer by a m-unit layer must be learnt, whereas only k parameters have to be learnt with convolution
- Since $m \approx n$ and $k < n$, it follows that $k \ll m \times n$: Using convolution allows for a huge increase in statistical efficiency

Equivariant representations **Equivariant representations**: Equivariance of a function f to a function g means that applying g to the input before applying f yields the same output as applying g to the output of f, that is, $f(g(x)) = g(f(x))$:

- Convolution is equivariant to translation (but not to other transformations, like rotation or scaling)
- For example, shifting the intensity of all image pixels to the right by one unit and then applying convolution yields the same result as first applying convolution then shifting
- Similarly, in a time series, if an event is moved at a later time in the series, the corresponding output is also moved at a later time in the output. Therefore, convolution can produce a representation that shows the location of certain features in the input

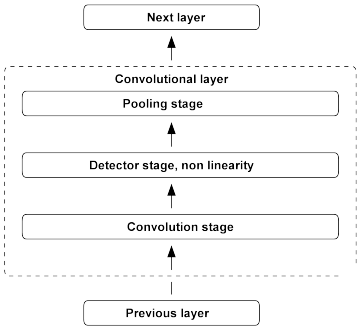


Figura 8.7: convolutional layer

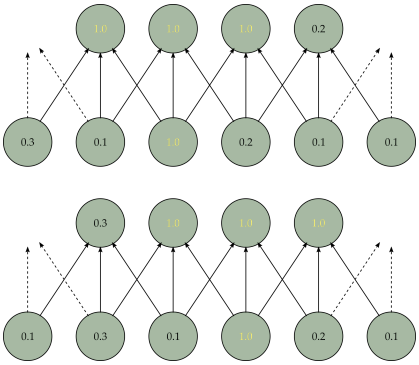


Figura 8.8: pooling and invariance

Convolutional layer Each layer of a Convolutional NN is actually internally dividede into three separate stages: **Convolution**, **Detector**, and **Pooling**

The detector stage applies the activation function, for example RELU. The pooling stage operates on the outputs: For each output it collects neighbouring outputs, it applies summary statistics, for example the maximum, and replaces the output with the new value

Boh fig 8.7.

Pooling and invariance Pooling improves invariance to small translations, in that a small shift of the input does not change most outputs. The property is useful to determine the presence of a feature in a region, independent of the exact location, for example an eye in a face

Example: the pooling stage employs the maximum function, with a pooling region of three units. The output values of the previous stage have been moved one unit to the right. Two pooling units change their maximum and two units retain it

Boh fig 8.8.

Pooling over separate convolutions Whereas pooling enhances invariance to translations, if a pooling unit receives input from separate convolutions, it

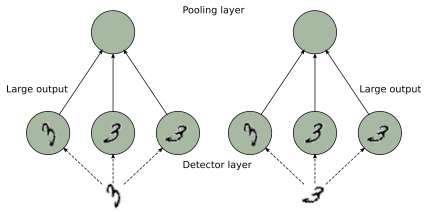


Figura 8.9: pooling over separate convolutions

can learn invariance to other transformation, such as rotations
Boh fig 8.9.

8.1 Python

Example 8.1.1 (CNN MNIST)**.** Take this shit muthafucka

```
#!/usr/bin/env python
# coding: utf-8
# Convolutional Neural Networks for Image Classification (CIFAR)
# In[1]:


# Import packages

import warnings
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import numpy as np
import random

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD


# ### Parameters of the CNN and number of classes

# In[2]:


batch_size = 100
num_classes = 10
epochs = 10000
```

```
# ### Load dataset and define the shape of each image

# In[3]:


# input image dimensions
img_rows, img_cols, rgb = 28, 28, 1

# load cifar dataset from keras
# the dataset is automatically loaded as train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()


# ### Sampling of the original training set

# In[4]:


# Sampling data to tun the experiments quickly
seed=123
np.random.seed(seed)

# n is the number of observations in each class
# the size of the training set will nxn° distinct classes
n = 1000
mask = np.hstack([np.random.choice(np.where(y_train == l)[0], n, replace=False)
                      for l in np.unique(y_train)])
random.shuffle(mask)
x_train, y_train = x_train[mask], y_train[mask]


# ### Define the function to visualize the images and the correspondent labels

# In[5]:


def plot_images(images, labels, nrow, ncol):
    # plot images
    fig, axes = plt.subplots(nrow, ncol, figsize=(1.5*ncol,2*nrow))
    for i in range(nrow*ncol):
        ax = axes[i//ncol, i%ncol]
        ax.imshow(images[i], cmap='gray')
        ax.set_title('Label: {}'.format(labels[i]))


# In[6]:


num_row = 4
```

```
num_col = 5

plot_images(x_train, y_train, num_row, num_col)


# ### Pre-processing

# In[7]:


# each image is represented by a multidimensional array (28x28x1)
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, rgb)


# In[8]:


# reshape the numpy array
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)


# ### Define the model

# In[9]:


model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

```
# ### Visualise the diagram of the model

# In[10]:


from keras.utils.vis_utils import plot_model
from IPython.display import Image

plot_model(model, show_shapes=True, show_layer_names=True, to_file='model.png')
Image('model.png')


# In[11]:


model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])


# ### Predict the labels for the test set

# In[12]:


y_hat = model.predict_classes(x_test)


# In[13]:


test_labels = np.argmax(y_test, axis=1)


# ### Plot images with predicted classes

# In[14]:
```

```
plot_images(x_test, y_hat, num_row, num_col)


# ### Plot images with true classes

# In[15]:


plot_images(x_test, test_labels, num_row, num_col)
```

Example 8.1.2

```
# Example 8.1.2. python
# coding: utf-8

# # Convolutional Neural Networks for Image Classification (CIFAR)

# In[1]:


# Import packages

import warnings
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import numpy as np
import random

import keras
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD


# ### Parameters of the CNN and number of classes

# In[2]:


batch_size = 100
num_classes = 10
epochs = 1000


# ### Load dataset and define the shape of each image

# In[3]:
```

```
# input image dimensions
img_rows, img_cols, rgb = 32, 32, 3

# load cifar dataset from keras
# the dataset is automatically loaded as train and test sets
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train.shape


# ### Sampling of the original training set

# # Sampling data to tun the experiments quickly
# seed=123
# np.random.seed(seed)
#
# # n is the number of observations in each class
# # the size of the training set will nxn° distinct classes
# n = 1000
# mask = np.hstack([np.random.choice(np.where(y_train == l)[0], n, replace=False)
#                       for l in np.unique(y_train)])
# random.shuffle(mask)
#
# x_train, y_train = x_train[mask], y_train[mask]
# x_train.shape

# ### Define the function to visualize the images and the correspondent labels

# In[4]:


def plot_images(images, labels, nrow, ncol):
    # plot images
    fig, axes = plt.subplots(nrow, ncol, figsize=(1.5*ncol,2*nrow))
    for i in range(nrow*ncol):
        ax = axes[i//ncol, i%ncol]
        ax.imshow(images[i], cmap='gray')
        ax.set_title('Label: {}'.format(labels[i]))


# In[5]:


num_row = 2
num_col = 3

plot_images(x_train, y_train, num_row, num_col)


# In[6]:
```

```
# 0: airplane
# 1: automobile
# 2: bird
# 3: cat
# 4: deer
# 5: dog
# 6: frog
# 7: horse
# 8: ship
# 9: truck


# In[7]:


# each image is represented by a multidimensional array (32x32x3)
input_shape = (img_rows, img_cols, rgb)


# ### Pre-processing

# In[8]:


# reshape the numpy array
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)


# ### Define the model

# In[9]:


# define the model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'
```

```
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padd
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', pad
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', pad
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
model.add(Dense(10, activation='softmax'))

# compile model
opt = SGD(lr=0.001, momentum=0.9)
model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])


# ### Visualise the diagram of the model

# In[10]:


from keras.utils.vis_utils import plot_model
from IPython.display import Image

plot_model(model, show_shapes=True, show_layer_names=True, to_file='model.png')
Image('model.png')


# In[11]:


model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))

score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])


# ### Predict the labels for the test set

# In[12]:


y_hat = model.predict(x_test)
```

```
labels_hat = np.argmax(y_hat, axis=1)

# In[13]:

y_hat

# In[14]:

labels_hat

# In[18]:

test_labels = np.argmax(y_test, axis=1)

# ### Plot images with predicted classes

# In[19]:

plot_images(x_test, labels_hat, num_row, num_col)

# ### Plot images with true classes

# In[20]:

plot_images(x_test, test_labels, num_row, num_col)

# In[ ]:
```

Capitolo 9

Autoencoders

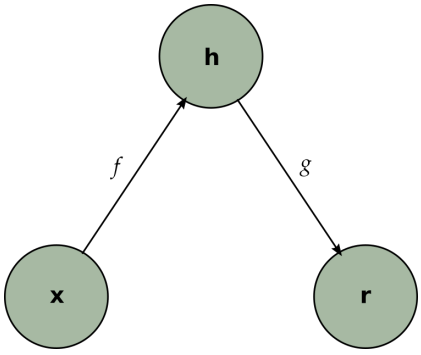
Autoencoder: Neural network that is trained to copy its input to its output:

- the architecture consists of an encoder $\mathbf{h} = f(\mathbf{x})$ which outputs a **code** vector $\mathbf{h}$, followed by a **decoder** which outputs a reconstruction $\mathbf{r} = g(\mathbf{h})$
- Autoencoders are designed to only *approximate* the input with the output, *rather than copying* exactly; in the approximation process, autoencoders learn interesting features of the data

Undercompletion for autoencoder we have

- when the dimension of the code $\mathbf{h}$ is smaller than the dimension of the input $\mathbf{x}$, the autoencoder is called **undercomplete** (we're summaring somehow)
- undercompletion is a deliberate choice, made in the hope to obtain useful properties of $\mathbf{h}$ after training
- the loss function is the error committed in reconstructing using the autoencoder

$$L(\mathbf{x}, g(f(\mathbf{x})))$$



(a)

Figura 9.1: autoencoder

for autoencoders is meant to penalise the difference between the input $\mathbf{x}$ and its reconstruction $\mathbf{r} = g(f(\mathbf{x}))$

- A linear decoder with a MSE function as L makes an undercomplete autoencoder span the PCA space
- Thus, non-linear activation allows for a generalisation of PCA, but also opens the possibility of overfitting, if capacity is too large

Overfitting/overcomplete encoders we have:

- An extreme case would be an autoencoder, with $\mathbf{h}$ of dimension 1, that is capable of representing the i-th input x_i of the training set as a distinct code number i
- Using such extremely detailed information, a suitable decoder could then map every i to the corresponding input x_i exactly, thereby learning the identity function
- The resulting autoencoder has minimal loss but clearly it is of no use at all
- When the dimension of $\mathbf{h}$ is at least the dimension of $\mathbf{x}$, the autoencoder can also be trained to simply copy input to output
- Autoencoders with the dimension of $\mathbf{h}$ greater than the dimension of $\mathbf{x}$ are called **overcomplete**

Regularisation asd

- Keeping the size of the code $\mathbf{h}$ and the capacity of the encoder and decoder small is not the only way to prevent overfitting
- A different approach is to *design the loss* function so that the autoencoder model has additional properties, such as sparsity or robustness to noise. Such autoencoders are called **regularised**
- Regularised autoencoders can be non-linear, overcomplete, and have a capacity high enough to learn the identity function, and still be capable of learning useful features from the data

Interlude: Capacity and regularisation foo

- The functions $f(\mathbf{x}; \boldsymbol{\theta})$ from which the learning algorithm draws the solution are called **hypotheses**, and their set is the **hypothesis space**
- A larger hypothesis space increases capacity (eg apt to be adapted to different situation)
- For example, adding higher order powers

$$\hat{y} = \sum_{i=0}^{n} w_i x^i$$

to a one-dimensional regression model increases its capacity

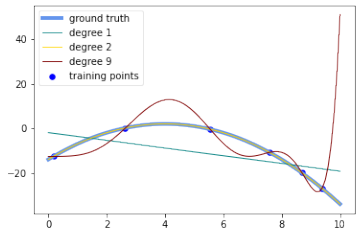


Figura 9.2: example deterministic

- Note that the normal equations are valid because the expression are linear in the parameters

Example 9.0.1. Example with deterministic data:

- Generate data from $-(x-4)^2+2$ at random xs, fit degree 1, 2, and 9 regression models
- Linear regression underfits
- Quadratic regression exactly captures the underlying generating function
- The function of degree 9 passes through the points but overfits: other points would not satisfy the fitted function

fig 9.2

Training and test errors waaa

- There are multiple formalisations of model capacity in **statistical learning theory**, with different properties
- Training error decreases as the number of examples increases, approaching the minimum value for the model
- The difference between the training and test errors has an upper bound that increases with capacity, but decreases as the number of examples increases
- Low capacity models tend to have a *smaller gap* between training and test error
- As a function of capacity, the test error decreases (**underfitting regime**) until it reaches a minimum, then increases again (**overfitting regime**). The capacity at the minimum is the **optimal capacity**

Example 9.0.2. Example with polynomials: fig 9.3

- Data: Degree 5 polynomial plus a uniform random term
- With few examples (< 30) degree 2 achieves the best test error, with more examples the optimal degree 5 model outperforms it
- The degree 9 model is overfitting until the number of examples is large enough to compensate for its excessive capacity (complexity, luca)

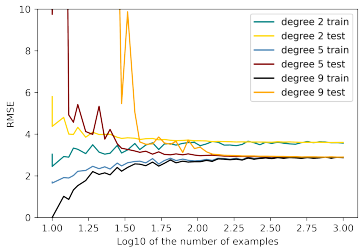


Figura 9.3: example polynomials

Equivalence of classifiers asd

- A result known as the **no free lunch theorem** states that all classificaton algorithms have the same error rate on previously unseen examples *when we average over every data generating distribution*
- The result implies that the goal of designing the best possible learning algorithm cannot be pursued, because a complex algorithm does not perform on average better than guessing
- However, since the result holds only if we consider all distributions, learning algorithms that generalise well can be designed if we can make assumptions on the distributions that generate the data of our case of interest

Regularisation **Regularisation**: Modification to a learning algorithm that aims at reducing its test error (but not its training error); this is reason why it's so important for high capacity models with higher risk of test error.

- Regularisation expresses *preferences* on the functions $f(\mathbf{x};\boldsymbol{\theta})$: when the learning algorithm finds two eligible functions in the hypothesis space for the solution, it chooses the preferred one. The other function is chosen only if its fitting to the training data is significantly better
- Usually, if a model learns a function $f(\mathbf{x};\boldsymbol{\theta})$, the model can be regularised by adding a penalty to the loss function of the model. The penalty term is called a **regulariser**

Example 9.0.3 (Ridge regression)**.** Ridge regression is an example of regularised model, in which the function to minimise is the mean square error plus the squared norm of the weights $\mathbf{w}^T\mathbf{w}$, multiplied by a parameter $\lambda \geq 0$

$$F(w,\lambda) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{w}^T x_i^T)^2 + \lambda \mathbf{w}^T\mathbf{w}$$

The penalty term expresses a preference for solutions with fewer nonzero weights, or a less steep linear model
λ is chosen before minimisation
The resulting model is also called linear regression with **weight decay**

Same idea is implmeented in sparse autoencoders

Sparse autoencoders

- In sparse autoencoders, a penalty term $\Omega(\mathbf{h})$ on the code $\mathbf{h}$ which promotes sparsity is added to the reconstruction error loss. The resulting loss is

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h})$$

- Usually, sparse autoencoders are employed to learn features for subsequent learning tasks, for example classification
- Analysis reveals that the interpretation of the sparsity term actually departs from the regularisation framework (cfr goodfellow bengio)
- Instead, it highlights the usefulness of autoencoders as learners of latent variables that explain the input

9.1 Python

Example 9.1.1

```
# -*- coding: utf-8 -*-
# Example.en.py
# coding: utf-8

# # Convolutional Neural Networks for Image Classification (CIFAR)

# In[1]:


# Import packages
import warnings
warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt
import numpy as np
import random

import keras
from keras.datasets import mnist
from keras.models import Sequential, Model
from keras.layers import Dense, Dropout, Flatten, LeakyReLU, Activation, Reshape, Input
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD


# ### Parameters of the CNN and number of classes

# In[2]:


batch_size = 250
num_classes = 10
epochs = 200
```

```
# ### Load dataset and define the shape of each image

# In[3]:


# input image dimensions
img_rows, img_cols = 28, 28

# load MINST dataset from keras
# the dataset is automatically loaded as train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()


# ### Sampling of the original training set

# In[4]:


# Sampling data to tun the experiments quickly
seed=123
np.random.seed(seed)

# n is the number of observations in each class
# the size of the training set will nxn° distinct classes
n = 100
mask = np.hstack([np.random.choice(np.where(y_train == l)[0], n, replace=False)
                      for l in np.unique(y_train)])
random.shuffle(mask)
x_train, y_train = x_train[mask], y_train[mask]


# ### Define the function to visualize the images and the correspondent labels

# In[5]:


def plot_images(images, labels, nrow, ncol):
    # plot images
    fig, axes = plt.subplots(nrow, ncol, figsize=(1.5*ncol,2*nrow))
    for i in range(nrow*ncol):
        ax = axes[i//ncol, i%ncol]
        ax.imshow(images[i], cmap='gray')
        ax.set_title('Label: {}'.format(labels[i]))


# In[6]:
```

```
num_row = 4
num_col = 5

plot_images(x_train, y_train, num_row, num_col)


# ### Pre-processing

# In[7]:


# reshape the numpy array
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')


# ### Define the model

# In[ ]:


import keras
from keras import layers

# This is the size of our encoded representations
encoding_dim = 32  # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

# This is our input image
input_img = keras.Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = layers.Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction
autoencoder = keras.Model(input_img, decoded)


# In[8]:


p = 32 # number of features in the latent space
# Encoder
encoder = Sequential()
encoder.add(Flatten(input_shape = (28, 28)))
```

```
encoder.add(Dense(512))
encoder.add(LeakyReLU())
encoder.add(Dense(p))
encoder.add(LeakyReLU())


# In[9]:


decoder = Sequential()

decoder.add(Dense(64, input_shape = (p,)))
decoder.add(LeakyReLU())
decoder.add(Dropout(0.5))
decoder.add(Dense(784))
decoder.add(Activation("sigmoid"))
decoder.add(Reshape((28, 28)))


# In[10]:


# input shape
img = Input(shape = (28, 28))


# In[11]:


# latent space
latent_vector = encoder(img)

# output shape
output = decoder(latent_vector)


# In[12]:


# modeling
model = Model(inputs = img, outputs = output)
model.compile("nadam", loss = "binary_crossentropy")


# ### Visualise the diagram of the model

# In[13]:


from keras.utils.vis_utils import plot_model
```

```
from IPython.display import Image

plot_model(model, show_shapes=True, show_layer_names=True, to_file='autoencoder_MINST.png')
Image('autoencoder_MINST.png')


# In[14]:


model.fit(x_train, x_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1)


# In[19]:


# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)


# In[25]:


import matplotlib.pyplot as plt
one_image = np.random.uniform(low=0.0, high=1.0, size=32)*5-2.5
encoded_img = one_image.reshape(1,32)
decoded_img = decoder.predict(encoded_img)
n = 1  # How many digits we will display
plt.figure(figsize=(1, 1))
for i in range(n):
    # Display original
    #ax = plt.subplot(2, n, i + 1)
    #plt.imshow(x_test[i].reshape(28, 28))
    #plt.gray()
    #ax.get_xaxis().set_visible(False)
    #ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_img[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

```
# ### Predict the labels for the test set

# In[22]:


import matplotlib.pyplot as plt

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()


# In[ ]:
```