

Introduction

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Course organization

The course will be held by

- Sandro Cumani (sandro.cumani@polito.it)
- Salvatore Sarni (salvatore.sarni@polito.it)

The course consists of 3 hours per week of lessons and 1,5 hours per week of laboratory

- Thursday, 14:30 – 17:30, Lessons — Sandro Cumani
- Friday, 11:30 – 13:00, Laboratory (in class, bring your own laptop) — Sandro Cumani, Salvatore Sarni

For any question, you can write e-mails to sandro.cumani@polito.it
The subject of the email **MUST** begin with the course code
01URTOV

Course organization

The lessons will mainly cover theoretical aspects of Machine Learning and Pattern Recognition

The laboratories will allow implementing techniques presented during lessons

Attendance to laboratories is strongly encouraged. The laboratories will often introduce concepts that may not have been presented during classes (and are part of the oral examination)

Course organization

The exam will consist of two parts:

- A project (18 points)
- An oral examination (12 points)

Course organization

The project will require solving a classification problem on a specific dataset.

You will be able to choose among a small set of tasks (the tasks will be presented in the following weeks)

The project may be done in groups (at most 2 people)

You should analyze which, among the methods that will be presented during classes, are suited to solve the problem

You should solve the problem using different approaches, and provide a critical analysis of the results obtained by the different techniques

You are free to experiment with methods that extend what we will see during classes

Course organization

You are required to submit a report, providing

- A description of the problem and of the dataset, together with an analysis of the dataset features
- An analysis of different methods that can solve the problem
- A comparison of the effectiveness of the different methods
- A critical analysis of the results
- The code that you used to implement the different algorithms (Python)

Even if some techniques do not work well, you are encouraged to add them, with a justification for their bad performance

Course organization

Since this course presents the basis of Machine Learning, **avoid** using ML libraries or ML toolboxes for the project (using toolboxes will result in lower marks — one of the goal of the course is that you learn how to implement the approaches)

The laboratories are already organized as to allow you to implement many of the techniques that we will discuss

You can, of course, re-use the code developed during the labs (including snippets provided by us)

If you are in doubt whether you can use some library or not, **ASK**

Course organization

If you want to participate to an exam session, you have to submit the report by the official exam date

Oral examinations will start 7 – 10 days after, the timetable will be provided through the teaching portal

The report must be submitted through the teaching portal, section “Work Submission” (Elaborati)

The format should be a .zip file, containing

- The report in pdf format
- The source code (python source files, no jupyter notebook or similars)

The file name should be <student id>_<exam-date>.zip

Course organization

For group projects, both authors should upload their own .zip file

The report must contain the names of **both** authors

Course organization

You can keep the same report for different oral sessions

The report mark will be disclosed during the oral exam

If you fail or withdraw from an oral session, you can also upload a new report if you wish to improve the report mark:

- If you withdraw before getting the report mark, you can resubmit the report on the same task
- After the report mark has been disclosed (to you or to your team mate), if you want to improve the mark you have to submit a report on a **different** task

The oral examination will include a discussion of the project, and questions that can cover the whole program

Course organization

In the teaching portal you will find the text of the laboratories and the slides used during classes. These will come in two versions:

- Slides projected during the classes
- Print-friendly version with less color

Reference books:

- [1] Christopher M. Bishop. 2006. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg.
- [2] Kevin P. Murphy. 2012. Machine Learning: A Probabilistic Perspective. The MIT Press.

Introduction to Python

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

High-level, interpreted language

Multiple programming paradigms

- Procedural, object oriented, functional

Multi-platform open-source implementations (Windows, Linux, Mac OS)

We will use Python 3

- Available for different platforms
- Python tutorial available at
<https://docs.python.org/3/tutorial/>

You can install Python and the required libraries in many different ways (install packages, distribution packages, ...)

If you have no prior experience, you can use Anaconda, a framework that already includes Python and the numerical libraries we will need:

<https://www.anaconda.com/products/individual>

Python

The Python interpreter can execute scripts (i.e. code from source files)

```
$> python myscript.py
```

The interpreter also allows for interactive sessions

```
$> python  
>>> print("Hello World")  
Hello World
```

The Python shell is quite basic. Alternative shells such as IPython offer many more functionalities

In IPython, you can check function / method documentation simply calling `func?` (question mark included)

Numeric Types

Python natively supports several data types

Numerical types (int, float):

```
>>> 2 + 2
```

```
4
```

```
>>> 3.0 * 5.5
```

```
16.5
```

Operator `**` allows computing powers

```
>>> 3 ** 4
```

```
81
```

Numeric Types

Division `/` always results in `float`, integer division is obtained through `//`:

```
>>> 6 / 4
```

```
1.5
```

```
>>> 6 // 4
```

```
1
```

The remainder is computed with the operator `%`:

```
>>> 6 % 4
```

```
2
```

Strings

Strings:

- Sequences of characters enclosed in single quotes ' . . .' or double quotes " . . . "

```
>>> "This is a string"  
'This is a string'  
  
>>> 'Also this is a string'  
'Also this is a string'
```

Strings

Strings:

- Immutable (cannot be modified, but new strings can be obtained from existing ones)

```
>>> "This" + " is " + 'a string'  
'This is a string'
```

Strings

Strings:

- Single quotes are allowed as characters in double-quoted strings and vice-versa. \ allows escaping quotes

```
>>> "A ' in a string"  
"A ' in a string"  
  
>>> "A \" in a string"  
'A " in a string'
```

Strings

Strings can be indexed (indexes start from 0 as in C)

```
>>> s = "Hello"
```

```
>>> s[0]  
'H'
```

NOTE: in C, an element of a string is of type char. In Python, indexing a string works differently — `s[0]` is still a string

Strings

Negative indices can be used to **index from the end** (but -0 is equal to 0 and is the index of the first element)

```
>>> s = "Hello"  
  
>>> s[ -1]  
'o'
```

Strings are **immutable** — we cannot assign a value to an element of a string

The built-in operator **len** allows computing the length of a string

```
>>> len(s)  
5
```

Strings

Strings can be sliced — we can take the substring starting in `s` and ending in `e` (the ending point is always excluded)

```
>>> s[0:3]  
'Hel'
```

The first and/or last elements can be omitted, and are implicitly assumed to be, respectively, 0 and the end of the string

```
>>> s[:3]  
'Hel'  
  
>>> s[3:]  
'lo'
```

Strings

Negative indices can be used also in slices

```
>>> s[-3:-1]  
'll'
```

Consecutive slices can be represented as `s[i:j]` and `s[j:k]`

Length of a slice:

```
>>> j-i
```

Accessing a slice of length `n`:

```
>>> s[i:i+n]
```

Objects variables and types

In Python everything is an object:

- int, float, str: built-in objects

We can **bind** values (objects) to **names**

A **variable** is a name bounded to a value

- In practice, the variable contains a reference (pointer) to the object

Strongly, dynamically typed language:

- Variables do not have types
- Values always have a type

Objects variables and types

No need to specify a variable type like in C

Variables can be bounded to values with different types

```
>>> a = 3  
  
>>> print(a)  
3  
  
>>> a = 'Hello World'  
  
>>> print(a)  
Hello World
```

Objects variables and types

No need to declare variables in advance – however variables should be assigned a value before being used

```
>>> print(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    NameError: name 'b' is not defined

>>> b = 3

>>> print(b)
3
```

Objects variables and types

Value types determine the operations that can be performed

Python allows for **duck-typing**

- Rather than checking the type of a value, we check whether it implements specific functionalities (i.e. methods) — more on this later

Data types

Python natively supports high-level data types, such as lists, tuples, hash tables and sets

A list can be built in Python by enclosing a sequence of comma-separated values in square brackets

```
>>> l = [1,2,3]
```

```
>>> l  
[1, 2, 3]
```

Data types

List elements can be heterogeneous

```
>>> l = [1, 'A', [4,5]]
```

```
>>> l  
[1, 'A', [4, 5]]
```

The length of a list can be obtained from function `len`

```
>>> l = [1,2,3]
```

```
>>> len(l)  
3
```

Data types

Lists can be indexed and sliced like strings

```
>>> l = [1,2,3,4,5]
```

```
>>> l[0]
```

```
1
```

```
>>> l[2:-1]
```

```
[3, 4]
```

Slicing a list returns a new list which contains references to the values of the original list (**shallow copy**)

More on slicing

The slicing operator allows also providing a step value

```
>>> l = [1,2,3,4,5]
>>> l[0:5:2]
[1,3,5]
```

Slice start and end can be omitted (implicitly assumed to be the beginning and end of the iterable)

The step can be negative – useful to traverse the iterable in reverse

```
>>> l = [1,2,3,4,5]
>>> l[::-1]
[5,4,3,2,1]
```

More on slicing

Pay attention when providing start or end with negative steps

```
>>> l = [1,2,3,4,5]
>>> l[0:5:-1]
[]
>>> l[5:0:-1]
[5, 4, 3, 2] # End point is always excluded
```

If we want to slice and reverse, better to do both steps separately

```
>>> l = [1,2,3,4,5]
>>> l[0:3][::-1]
[3,2,1]
```

Data types

A list can be modified assigning values to its elements (as if it were an array)

```
>>> l = [1,2,3,4,5]
```

```
>>> l[0] = 7
```

```
>>> l  
[7, 2, 3, 4, 5]
```

Slices can be assigned new values

- Assigned values should be iterable (e.g. lists)
- Assignment can both replace and remove values

Data types

Assigning values to a slice:

```
>>> l = [1,2,3,4,5]  
  
>>> l[1:4] = [11,12,13]  
  
>>> l  
[1, 11, 12, 13, 5]
```

Removing values ([] is an empty list):

```
>>> l[1:3] = []  
  
>>> l  
[1, 13, 5]
```

Data types

We can append an element to a list using its `append` method (more on methods later):

```
>>> l = [1,2,3]  
  
>>> l.append(4)  
  
>>> print(l)  
[1,2,3,4]
```

Data types

We can extend a list using its `extend` method:

```
>>> l = [1,2,3]  
  
>>> l.extend([4,5])  
  
>>> print(l)  
  
[1,2,3,4,5]
```

Data types

We can pop an element using its `pop` method (optional, pass the index of the element to pop, default is last element):

```
>>> l = [1,2,3]

>>> x = l.pop(1)

>>> print(l)

[1,3]

>>> print(x)

2
```

Data types

We can create a list from another iterable object (e.g. another list, a string, etc.) using `list`

```
>>> l = list([1,2,3])
```

```
>>> print(l)
```

```
[1,2,3]
```

```
>>> l=list('Hello')
```

```
>>> print(l)
```

```
['H', 'e', 'l', 'l', 'o']
```

Blocks, indentations and control structures

Conditional execution: `if`, `elif`, `else`

```
>>> x=3

>>> if x < 0:
...     print('Negative')
... elif x == 0:
...     print('Zero')
... else:
...     print('Positive')
```

Positive

Blocks, indentations and control structures

Python does not have explicit delimiters for blocks of code

Blocks are represented through **indentation**:

```
x = 3  
if x < 0:  
    x = x+1  
    x = x*2  
print(x)
```

Output: 3

```
x = 3  
if x < 0:  
    x = x+1  
    x = x*2  
print(x)
```

Output: 6

Blocks, indentations and control structures

Conditional execution: `if`, `elif`, `else`

The condition should be a Boolean (type `bool`) value

`True` and `False` built-in values (type `bool`)

Comparison operators return `bool` values

- `==`, `<`, `>`, `<=`, `>=`, `!=`

Blocks, indentations and control structures

Any object can be converted to bool (for control flow statements the conversion is automatic).

Almost everything is converted to True, except:

- The Boolean value `False`
- The numerical values `0`, `0.0`
- Empty collections (e.g. empty list `[]` or empty string `''`)
- The special value `None`

Note: explicit casting can be achieved invoking `bool()` over the value. The same applies for other types, such as `int()` or `float()`

Blocks, indentations and control structures

Logical operators can be used to form more complex logical expressions

- `and` , `or` , `not`

```
>>> (2 == 3) or (1 < 3)
```

```
True
```

```
>>> not True
```

```
False
```

Blocks, indentations and control structures

Boolean operators may be applied to both Boolean and non-boolean values (and may return non boolean values)

The semantic is:

- `not x`: evaluate x and yield True if x is false, False otherwise
- `x and y`: evaluate x; if x is false, return its value, else evaluate y and return its value
- `x or y`: evaluate x; if x is true, return its value, else evaluate y and return its value

This allows for conditional evaluation of y (similar to C)

Blocks, indentations and control structures

Conditional execution: `if`, `elif`, `else`

Multiple `elif` conditions can be included — they act as a compact way to express else – if statements and avoid excessive code nesting

```
if x < 3:  
    y = 4  
else:  
    if x > 10:  
        y = 5  
    else:  
        y = 2
```

```
if x < 3:  
    y = 4  
elif x > 10:  
    y = 5  
else:  
    y = 2
```

Blocks, indentations and control structures

Conditional execution: `if`, `elif`, `else`

The `else` block can be omitted

Blocks cannot be empty – the `pass` instruction can be used to do nothing

```
if x < 3:  
    y = 4  
print(y)
```

```
if x < 3:  
    y = 4  
else:  
    pass  
print(y)
```

Blocks, indentations and control structures

Python allows for conditional expressions, similar to C construct

```
cond ? exp1 : exp2
```

The syntax in Python is

```
exp1 if cond else exp2
```

```
>>> a = True
>>> x = 1 if a else 2
>>> print(x)
1
>>> y = 1 if not a else 2
>>> print(y)
2
```

Blocks, indentations and control structures

Loops: `while` and `for`

The `while` construct behaves like in C, repeating a block of code as long as the condition is true

```
>>> x = 3

>>> while x < 6:
...     print(x)
...     x = x+1
...
3
4
5
```

Blocks, indentations and control structures

The `for` construct behaves differently — it allows iterating over the elements of a sequence (e.g. string, list, etc.), following the order they have in the sequence

To avoid unintended behavior, it is better to avoid modifying the sequence while iterating

```
>>> for x in [1,2,3,'hello']:  
...     print(x)  
...  
1  
2  
3  
hello
```

Blocks, indentations and control structures

The `range` function can be used to generate a sequence of numbers on the fly (note: range does not create a list, but rather an iterator object that can be iterated)

```
>>> for x in range(3):
...     print(x)
...
0
1
2
```

Blocks, indentations and control structures

The `range` function allows defining also an initial value and a step (both are optional; if two values are passed they are assumed to be first and ending element of the range — remember that the ending is always excluded)

```
>>> for x in range(1, 6, 2):
...     print(x)
...
1
3
5

>>> print(list(range(1,6)))

[1,2,3,4,5]
```

Blocks, indentations and control structures

`break` and `continue` statements can be used to exit from the innermost cycle or to skip to the next iteration. The semantic is the same as in C

```
>>> for x in range(10):
...     print(x)
...     if x >= 3:
...         break
...
0
1
2
3
```

Blocks, indentations and control structures

`break` and `continue` statements can be used to exit from the innermost cycle or to skip to the next iteration. The semantic is the same as in C

```
>>> for x in range(10):
...     if x >= 3 and x <= 8:
...         continue
...     print(x)
...
0
1
2
9
10
```

Blocks, indentations and control structures

Loop statements allow for an `else` clause as well

We won't make use of it — if you're interested you can check the Python documentation

Functions

Python allows defining functions with the `def` keyword, followed by the function name and the list of formal parameters (in parenthesis)

```
>>> def f(x):
...     return x + 1
...
>>> f(2)
3
```

A `return` statement is used to make the function return a value

Functions with no `return` statement still return a default value when they reach the end of the function block. The value returned is the special value `None`

Functions

When a function is executed, a local symbol table is created to store local variables

Variables that are assigned inside the function are stored in the local symbol table

Referenced variables are looked-up in the local symbol table, then in the local tables of enclosing functions, and finally in the global symbol table

In general, it's not possible to directly assign to a global variable inside a function (there are exceptions, but we will avoid using them to keep things simpler)

Functions

Function arguments are passed by value

```
>>> a = 3
>>> def f(x):
...     x = 4
...
>>> print(a)
3
>>> f(a)
>>> print(a)
3
```

Functions

However, the value that is copied is the **object reference** (similar to passing pointers in C):

```
>>> l = [2,3]
>>> def f(x):
...     x.append(4)
...
>>> print(l)
[2,3]
>>> f(l)
>>> print(l)
[2,3,4]
```

Here, we used method `append` to modify the list object whose reference is the value of `l`. Inside the function, `x` is a copy of the list reference. However, it refers to the same list.

Functions

Python functions allow for both positional and keyword arguments (arguments with default values):

```
>>> def f(x=4):  
...     print(x)  
...  
>>> f()  
4  
>>> f(2)  
2  
>>> f(x=3)  
3
```

Functions

Positional arguments must be specified when invoking the function, keyword arguments can be omitted (and will use the default value)

```
>>> def f(a, b=0):
...     print(a+b)
...
>>> f(1)
1
>>> f(1, b=4)
5
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required positional argument
: 'a'
```

Functions

When invoking the function, positional arguments may be passed via keywords, and optional arguments may be passed as positional (following the order in which they are defined)

```
>>> f(1,2)
3
>>> f(a=1,b=2)
3
```

Positional arguments must be placed before keyword ones

```
>>> f(1,2)
3
>>> f(b=2,1)
  File "<stdin>", line 1
    SyntaxError: non-keyword arg after keyword arg
```

Functions

Arguments can be restricted to be positional only or keyword only — if you're interested, check the tutorial

Python also allows for arbitrary argument lists and arbitrary keyword only argument dictionaries

```
>>> def f(*a):
...     print(a)
...
>>> f(1,2,3)
(1,2,3)
```

In this case, the arguments are packed in a [tuple](#) (see below) and passed as value for the parameter `a`

Functions

Arguments can be restricted to be positional only or keyword only — if you're interested, check the tutorial

Python also allows for arbitrary argument lists and arbitrary keyword only argument dictionaries

```
>>> def f(*a, **kw):
...     print(kw)
...
>>> f(a=1, b=2)
{'b': 2, 'a': 1}
```

In this case, the keyword arguments are packed in a [dictionary](#) (see below) and passed as value for the parameter `kw`

Functions

In python functions are objects as well

They can be assigned and passed to functions

```
>>> def fpr(x):
...     print(x)
...
>>> def g(f, a):
...     f(a+1)
...
>>> g(fpr, 1)
2
```

Functions

Anonymous functions can be created on the fly through the keyword `lambda`

The body of the function can only be an expression, and is the value returned by the function

```
>>> (lambda x: x+1)(3)  
4
```

```
>>> f = lambda x: x+1  
>>> f(0)  
1
```

Functions

Functions can be defined in nested scopes

The nested function can access, but not modify (unless explicitly indicated) the variables of the enclosing function

```
>>> def f(x):
...     def g(y):
...         return x+y
...     print g(1)
...
>>> f(3)
4
```

Functions

This can be useful to create functions with “pre-set” parameters.
In the example, we define a generic “add_n” function:

```
>>> def add_n(n):
...     def add(x):
...         return x+n
...     return add
...
>>> add_3 = add_n(3)
>>> add_3(2)
5
```

Python implements high-level data types

We have already introduced lists – you can refer to the Python manuals for the complete set of list methods

Python allows for concise syntax to create new lists: [list comprehension](#)

This construct allows creating a list by applying some operations on the elements of another list or iterable object

```
>>> l = [i**2 for i in range(4)]  
>>> print(l)  
[0, 1, 4, 9]
```

List comprehension

Filters can be applied to select only a set of elements

```
>>> [i**2 for i in range(10) if i % 2 == 0]  
[0, 4, 16, 36, 64]
```

No `else` branch should be provided, only elements for which the condition is True will be selected

Note the difference with conditional expressions

```
>>> [i**2 if i % 2 else 0 for i in range(10)]  
[0, 1, 0, 9, 0, 25, 0, 49, 0, 81]
```

Tuples

Besides lists, Python supports another sequence type: tuples

A tuple is a sequence of values separated by commas

```
>>> t = 1,2,3, 'A'  
>>> print(t)  
(1, 2, 3, 'A')
```

Although in some case it's not necessary, to ensure the proper operation order, it's better to enclose the values in parentheses

```
>>> t = (1,2,3, 'A')  
>>> print(t)  
(1, 2, 3, 'A')
```

Tuples

Tuples are **immutable** (their values cannot be changed), but can be sliced, indexed and iterated in the same way as lists

The empty tuple is represented by a set of empty brackets `()`

Pay attention: `x = 1` is different from `x = 1,`

Python allows also for **tuple unpacking** — the elements of a tuple can be assigned to a sequence of variables directly

```
>>> x, y, z = (1,2,3)
>>> print(x)
1
```

Tuples

Tuples packing and unpacking is useful, for example, to return more than a single value from a function

```
>>> def f():
...     return 1, 2
...
>>> a, b = f()
```

Tuples

Another interesting use case is iteration of tuples of values

```
>>> l = [(1,'a'), (2, 'b'), (3, 'c')]  
>>> for val, name in l:  
...     print(val, name)  
  
...  
1 a  
2 b  
3 c
```

Tuples

The `zip` function can be used to create tuples of values from a set of lists (`zip` actually returns an *iterator*, i.e. an object that we can iterate to get values from, but can be iterated only once — we use the `list` function to create a real list in the example)

```
>>> l1 = [1,2,3]
>>> l2 = ['a', 'b', 'c']
>>> l = list(zip(l1, l2))
>>> print(l)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Tuples

The `zip` function can be used to create tuples of values from a set of lists (`zip` actually returns an iterator, i.e. an object that we can iterate to get values from)

```
>>> l1 = [1,2,3]
>>> l2 = ['a', 'b', 'c']
>>> for val, name in zip(l1, l2):
...     print(val, name)
...
1 a
2 b
3 c
```

Tuples

The function `enumerate` returns an iterator that provides pairs of `(index, value)` from a given iterable

```
>>> l2 = ['a', 'b', 'c']
>>> for val, name in enumerate(l2):
...     print(val, name)
...
0 a
1 b
2 c
```

NOTE: Iterators like those returned by `zip`, `range` and `enumerate` are consumed once they are used, if we need to iterate the result multiple times use `list` to convert the result to a list

Sets

Sets are unordered collections of items without duplicates

Sets are usually employed for testing membership and removing duplicate entries from a collection

Python allows creating sets using a specific notation or the built-in class `set`, which receives an iterable object:

```
>>> a = {1,2,3}
>>> print(a)
set([1, 2, 3])

>>> b = set([1,2,3])
>>> print(b)
set([1, 2, 3])
```

Sets

Note: the object `{}` is not the empty set, but the empty dictionary. To create an empty set, use `set()`

To test set membership, we can use the operator `in`:

```
>>> a = {1,2,3}  
>>> 1 in a  
True
```

The `in` operator can be used also for lists and tuples, but the look-up is much slower

Sets

Python sets implement standard set operations (union, intersection, ...)

```
>>> a = {1,2,3}  
>>> b = {2,3,4}  
>>> a.intersection(b)  
set([2, 3])
```

As with lists, we have set comprehensions as well:

```
>>> {i//2 for i in range(10)}  
set([0, 1, 2, 3, 4])
```

Dictionaries

Python includes a built-in **dictionary** (`dict`) type

Dictionaries are objects that allow associating **keys** to **values**

Think of dicts as sets of pairs *key:value*, where keys are unique

Keys are required to be immutable (and hashable)

Dictionaries

We can define dictionaries explicitly from key:value pairs

```
>>> a = {'A': 1, 'B': 2, 'C': 3}  
>>> print(a)  
{'A': 1, 'C': 3, 'B': 2}
```

We can look-up values in the dict:

```
>>> a = {'A': 1, 'B': 2, 'C': 3}  
>>> print(a['A'])  
1
```

Dictionaries

The empty dict is defined as `{}`

We can assign values to keys (overwriting if the key was present):

```
>>> a = {}
>>> a['A'] = 1
>>> a['B'] = 2
>>> print(a)
{'A': 1, 'B': 2}

>>> a['A'] = 3
>>> print(a)
{'A': 3, 'B': 2}
```

Dictionaries

The dictionary can be iterated, the iteration runs over the dictionary keys

No guarantees on the order in which keys are visited (this was recently changed in Python 3.7) — sort the keys (builtin-in function `sorted` or use an `OrderedDict` from the `collection` module)

```
>>> a = {'A': 1, 'B': 2, 'C': 3}
>>> print list(a)
['A', 'C', 'B']

>>> [a[i] for i in sorted(a)]
[1,2,3]
```

Dictionaries

The collection of keys can be recovered also using the `.keys` method

The collection of values can be recovered using the `.values` method

Elements can be removed using the `del` keyword:

```
>>> a = {'A': 1, 'B': 2, 'C': 3}
>>> del a['A']
>>> print(a)
{'C': 3, 'B': 2}
```

Keyword `del` can also be used to remove elements from lists, or to delete variables

Dictionaries

Dictionaries can be created using dict comprehensions:

```
>>> {i: i**2 for i in range(5)}  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

The `dict` construct allows creating dictionaries from sequences of key-value pairs

```
>>> dict([('A', 1), ('B', 2)])  
{'A': 1, 'B': 2}
```

Importing modules

Python allows using functionalities defined in different files

Each file is called a **module**, and modules can be **imported** in other modules, or in an interactive session

Importing a module allows accessing functions and variables defined in the file

File fact.py

```
def fact_func(n):
    if n == 0:
        return 1
    return n*fact_func(n-1)
```

File script.py

```
import fact
print(fact.fact_func(4))
```

```
$> python script.py
```

24

Importing modules

The `import` statement allows importing a module

Remember that the imported file name extension should be `.py`, but the extension should not be specified when importing the module

The symbols defined in the module are placed in a private symbol table associated to the module, and can be accessed through the dot `.` operator

Importing modules

When a module is imported, **it is actually executed** (multiple imports are allowed, but the module is executed only the first time)

File mod.py

```
a = 3  
b = a + 4  
print ('Hello world')
```

File script.py

```
import mod  
print(mod.b)
```

```
$> python script.py  
Hello world
```

7

Importing modules

Inside a module, its own name is accessible through the variable

`__name__`

The main module (the module we pass to the interpreter at launch) has the special name `'__main__'`

This can be used to execute parts of a script only when it's the main module but not when it's imported

Importing modules

File mod.py

```
a = 3
b = a + 4
if __name__ == '__main__':
    print('Hello world')
```

File script.py

```
import mod
if __name__ == '__main__':
    print(mod.b)
```

```
$> python script.py
```

```
7
```

```
$> python mod.py
```

```
Hello world
```

Importing modules

Assigning a different name to a module: use the `as` keyword

```
>>> import mod as mymodule  
>>> print(mymodule.b)  
7
```

We can import module symbols in the current local symbol table:

```
>>> from mod import b  
>>> print(b)  
7
```

We can import all module symbols in the local symbol table with `import *` — Pay attention that you do not accidentally overwrite a local variable or function (suggestion: avoid `import *`)

Importing modules

Libraries can be treated as modules (they are in general packages, but we won't go into details)

Library modules sometimes have sub-modules. In some cases these need to be imported as well before their content is accessible (but this depends on the library implementation)

We will see some examples when introducing the numerical libraries

Input / output

Files can be accessed using the function `open`

```
>>> f = open('filename', 'r')
```

The first argument is the filename. The second argument is the *mode*. Text modes are similar to C: `'r'`, `'w'`, `'a'`, `'r+'`.

Binary encoding is represented by `'b'` appended to the mode. In this case, I/O operations do not return strings, but byte sequences (we won't use explicitly the binary encoding)

Input / output

To avoid data loss on writing, files should be always closed when I/O is finished

We can let the file object handle this automatically using the `with` keyword

```
>>> with open('filename', 'r') as f:  
...     #do something with f here
```

The `with` statement takes care of closing the file when the block is finished (this is done by automatically calling some methods of the file object when the `with` statement starts and ends)

Input / output

To read an entire file, we can use the method `read(size)`

`size` is an optional parameter specifying the maximum number of characters or bytes to read.

file tmp.txt

This is a file
with multiple lines

script.py

```
f=open('tmp.txt')  
print(f.read())
```

```
$> python script.py
```

This is a file
with multiple lines

Input / output

We can read a file line by line using the `readline` method. An empty string denotes that the end of file has been reached

```
>>> f = open('tmp.txt', 'r')
>>> f.readline()
'This is a file\n'
>>> f.readline()
'with multiple lines\n'
>>> f.readline()
''
```

Input / output

A file object can be iterated, and can thus be used in a `for` loop. The file is read line by line (remember, lines can end with `'\n'`)

```
>>> f = open('tmp.txt', 'r')
>>> for line in f:
...     print(line.rstrip())
...
This is a file
with multiple lines
```

Input / output

To write to a file we can use the method `write`

```
>>> f = open('out.txt', 'w')
>>> f.write('Hello')
>>> f.close()
```

We can also use `print`, providing an argument for parameter `file`

```
>>> f = open('out.txt', 'w')
>>> print('Hello', file=f)
>>> f.close()
```

Note: `print` adds a newline at the end of the string, unless we pass `end=''`

Serialization

Complex data can be serialized (i.e. transformed to a string that can be written to a file) in different ways

Using `json` (more portable, less powerful):

```
>>> import json
>>> l=[1, ('Hello', 'world'), 3]
>>> f=open('out', 'w')
>>> json.dump(l, f)
>>> f.close()

>>> f = open('out', 'r')
>>> json.load(f)
[1, ['Hello', 'world'], 3]
```

Serialization

Complex data can be serialized (i.e. transformed to a string that can be written to a file) in different ways

Using Python `pickle` (less portable, supports more complex data):

```
>>> import pickle  
>>> l=[1, ('Hello', 'world'), 3]  
>>> f=open('out', 'wb') # Binary mode  
>>> pickle.dump(l, f)  
>>> f.close()  
  
>>> f = open('out', 'rb')  
>>> pickle.load(f)  
[1, ('Hello', 'world'), 3]
```

String formatting

Python supports C-style string interpolation. The syntax is
`'string' % values`, where values is either a single value or a tuple of values

```
>>> '%d is %s than %d' % (2, 'lower', 3)
'2 is lower than 3'
```

Format specifiers are similar to those of C `printf`

Python supports more advanced string formatting, you can refer to the tutorial

Some other useful string methods (they return new strings, check the documentation for the explanations of their parameters):

- `lower`, `upper` : convert a string to lower or upper case
- `lstrip`, `rstrip`, `strip` : remove leading / trailing / both whitespaces from a string
- `replace` : replace occurrences of text
- `split` : returns a list of words of a given string, separated using a second string as delimiter

Exceptions

Python allows handling errors at execution time (exceptions)

```
>>> try:  
...     y = 'a' + 3  
... except:  
...     y = 4  
...  
>>> print(y)  
4
```

Exceptions

We can specify which exception should be handled by each `except` block

```
>>> try:  
...     y = 'a' + 3  
... except ValueError:  
...     y = 4  
... except TypeError:  
...     y = 5  
...  
>>> print(y)  
5
```

Exceptions

We can specify code to be executed regardless of whether the `try` block incurs in an exception

```
>>> try:  
...     y = 'a' + 3  
... except:  
...     pass  
... finally:  
...     y = 3  
...  
>>> print(y)  
3
```



```
>>> try:  
...     y = 2  
... except:  
...     pass  
... finally:  
...     y = 3  
...  
>>> print(y)  
3
```

Exceptions

We can raise an exception through the `raise` keyword

```
>>> raise TypeError  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError
```

More on namespaces

We have seen the concept of namespaces when discussing modules and nested functions

In Python, a *namespace* is a mapping from names (variables, functions, ...) to objects

Examples of namespace include

- The set of built-in names (`sum`, `print`, ...)
- The global variables of a module
- The local variables of a function

Names in different namespace are independent

More on namespaces

Scopes define which names can be directly accessed

Scopes are searched in the following order

- Innermost scope, containing the local names
- Scopes of enclosing functions (starting from the innermost)
- The current module, containing the module global names
- The outermost scope, containing the built-in names

More on namespaces

We can access names in a different namespace through the dot
operator

We have already seen this when accessing methods of built-in types, or variables in a module

In general, we refer to the element after the dot as an [attribute](#) of the object on the left

Attributes can be read-only or writable

```
>>> import mymodule
>>> mymodule.newvar = 1
>>> print(mymodule.newvar)
1
```

Classes and objects

Python classes and objects allow writing code in an object-oriented way

To define a class, we use the `class` keyword, followed by a block of statements

These statements are executed in a local `namespace` associated to the class

Note: classes are themselves objects

We can access class attributes with the dot `.` operator

Classes and objects

script.py

```
class AClass:  
    x = 3  
    def f(self):  
        print('Hello world')  
  
if __name__ == '__main__':  
    print(AClass.x)  
    print(AClass.f)
```

```
$> python script.py  
3  
<function AClass.f at 0x7f2e4eab3ea0>
```

Classes and objects

Class instances (called objects) can be created by “calling” the class object

script2.py

```
from script import AClass
obj = AClass()
print(obj.x)
print(obj.f)
obj.f()
```

```
$> python script2.py
```

```
3
<bound method AClass.f of <script.AClass object at 0
    x7f0db0bb7ef0>>
Hello world
```

Classes and objects

The instantiated object allows for **attribute** look-up. Attributes can be either **data** attributes (similar to variables) or **methods**

Similar to variables, data attributes are not declared, but are created on assignment

script3.py

```
from script import AClass
obj = AClass()
obj.z = 1
print(obj.z)
```

\$> python script3.py

1

Classes and objects

script4.py

```
from script import AClass
obj = AClass()
obj.f()
AClass.f(obj)
```

```
$> python script4.py
```

```
Hello world
Hello world
```

Classes and objects

When a class is instantiated, its special method `__init__` is invoked (if not present, it's inherited from the built-in 'object' class)

We can use the method to initialize the object attributes

```
class Point:  
    name = '2D Point'  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

Note: `x` and `y` are *object* attributes that are created on the new object when it's instantiated

Class attributes (e.g. `name`) are **shared** by all instances

Classes and objects

Class attributes are used in place of object attributes whenever a given attribute is not found on an object

```
class A:  
    x = 3  
    def __init__(self, y):  
        self.y = y  
  
>>> a = A(1)  
>>> print(a.x)  
3
```

NOTE: class attributes are shared among all instances of the class, but can be masked by object attributes

Classes and objects

```
class A:  
    x = 3  
  
>>> a = A()  
>>> b = A()  
>>> print(a.x, b.x)  
3 3  
>>> a.x = 4  
>>> print(a.x, b.x)  
4 3
```

In this case, `a.x = 4` creates a *new* attribute called `x` on the object `a`

Classes and objects

```
class A:  
    x = [3]  
  
>>> a = A()  
>>> b = A()  
>>> print(a.x, b.x)  
[3] [3]  
>>> a.x.append(4)  
>>> print(a.x, b.x)  
[3, 4] [3, 4]
```

In this case, `a.x` is not assigned, but is accessed. Since `a` has no attribute `x`, class `A` is searched, and its `x` attribute is used. We then call a method of the object `A.x`. Its modifications are visible to all objects that do not specify a local attribute `x`

Classes and objects

Python classes can define *special* (or *magic*) methods

These methods, whose names start and end with a double underscore, allow defining how the object should behave in particular cases, such as when it appears in an expression

For example, the method `__add__` allows defining how to compute the result of a sum (`+`) of two objects

Classes and objects

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def __add__(self, other):  
        return Point(self.x + other.x, self.y +  
                     other.y)  
  
>>> p1 = Point(1,2)  
>>> p2 = Point(5,5)  
>>> p3 = p1 + p2  
>>> print(p3.x, p3.y)  
6, 7
```

Classes and objects

Other methods allow **overriding** different operators (arithmetic, logical, slicing, getting and setting items, ...)

For example, the `__str__` method defines how an object is converted to a string

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def __str__(self):  
        return 'Point(% .2f, % .2f)' % (self.x, self.y)  
  
>>> p = Point(1,2)  
>>> print(p)  
Point(1.00,2.00)
```

Classes and objects

Python allows a class to inherit from a different one

The syntax requires specifying a comma-separated sequence of base classes

```
class DerivedClass(Base1, Base2, Base3):  
    <statements>
```

Attribute look-up becomes more complex

For this course, we will mainly use classes and objects to simulate C-style structures, i.e. as containers of data

If you're interested, you can learn more on Python object-oriented programming on the Python tutorial and on the Python documentation

Command line arguments

Command line arguments can be accessed from the attribute `argv` of module `sys`

`argv` is a list of strings, each element corresponds to one argument

Command line arguments

script.py

```
import sys  
print(len(sys.argv))  
for arg in sys.argv:  
    print(arg)
```

```
$> python myscript.py 1 aaa 42
```

```
4  
script.py  
1  
aaa  
42
```

Numerical Libraries

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Numerical libraries

We will make heavy use of numerical libraries

The main ones are numpy and scipy — for more information:

<https://numpy.org/doc/stable/user/quickstart.html>

<https://numpy.org/doc/stable/index.html>

- Multi-dimensional arrays
- Linear algebra
- Numerical optimization
- Many ML algorithms (in particular with sklearn – **we won't use pre-built ML tools in most cases**)

Data visualization will rely on matplotlib

Numpy

Numpy main objects are multidimensional arrays (`ndarray` class)

We will mainly use one-dimensional (vector) and two-dimensional (matrices) arrays (not to be confused with the (matrix) class)

The dimensions are also called axes

- One-dimensional arrays have one axis, two-dimensional arrays have two axes, and so on

```
>>> numpy.array([1,2,3])
array([1,2,3])
>>> numpy.array([[1,2,3], [4,5,6]])
array([[1, 2, 3],
       [4, 5, 6]])
```

Arrays have attributes that describe the array itself

- `ndarray.size`: Total number of elements
- `ndarray.shape`: Tuple with the number of elements for each axis — A $m \times n$ matrix will have a shape (m , n)
- `ndarray.ndim`: Number of axes
- `ndarray.dtype`: The data type (for example, `numpy.int32`, `numpy.float32`, `numpy.float64`)

Numpy

```
>>> x = numpy.array([1,2,3])
>>> x.size
3
>>> x.shape
(3,)
>>> y = numpy.array([[1,2,3], [4,5,6]])
>>> y.shape
(2, 3)
>>> y.size
6
>>> y.ndim
2
```

Arrays can be created in many ways

- From Python lists or tuples. To create multi-dimensional arrays, use nested lists (see previous examples). We can specify the data type passing the argument `dtype`:

```
>>> numpy.array([1,2,3], dtype=numpy.float64)
array([ 1.,  2.,  3.])
```

- As a copy of another array

```
>>> y = numpy.array(x)
```

Numpy

Arrays can be created in many ways

- Using functions that create predefined arrays: zero, ones, arange, eye, ...

```
>>> numpy.zeros((2, 3), dtype=numpy.float32)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)

>>> numpy.ones(5)
array([ 1.,  1.,  1.,  1.,  1.])

>>> numpy.arange(4)
array([0, 1, 2, 3])

>>> numpy.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Numpy

The `arange` function is similar to the `range` function, and allows specifying a step size

```
>>> numpy.arange(0, 6, 2)  
array([0, 2, 4])
```

If we want to create an array of evenly spaced values in a range, we can use `linspace`

```
>>> numpy.linspace(0, 5, 4)  
array([ 0. , 1.66666667, 3.33333333, 5. ])
```

Arithmetic operators on numpy array operate *element-wise*

They require arrays with matching shapes

```
>>> x = numpy.array([[1,2,3], [4,5,6]])
>>> y = numpy.array([[2,2,2], [3,3,3]])
>>> x + y
array([[3, 4, 5],
       [7, 8, 9]])
>>> x * y
array([[ 2,  4,  6],
       [12, 15, 18]])
```

Arithmetic operators in general create new arrays. To modify an existing array, use in-place operators such as *=, +=, ...

Numpy

Matrix product can be performed using the dot function, or the @ operator (requires Python >= 3.5)

```
>>> x = numpy.array([[1,2], [3,4], [5,6]])
>>> y = numpy.array([[1,2,3], [4,5,6]])
>>> numpy.dot(x, y)
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51]])
```

Again, dimensions should match (in this case, the number of columns of x should be equal to the number of rows of y)

Numpy

Numpy arrays can be reshaped

```
>>> x = numpy.array([[1,2], [3,4], [5,6]])
>>> x
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> y = x.reshape((2,3))
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

The order of the data is preserved, but the shape is changed

Numpy

We will mainly use this to create row and column vectors:

```
>>> x = numpy.arange(3)
>>> x
array([0, 1, 2])

>>> x.reshape((1, x.size))
array([[0, 1, 2]])

>>> x.reshape((x.size, 1))
array([[0],
       [1],
       [2]])
```

Numpy

Method `ndarray.ravel()` allows reshaping the array to a 1-dimensional vector

```
>>> x = numpy.arange(12).reshape((2,2,3))

>>> x
array([[[ 0,  1,  2],
       [ 3,  4,  5]],

       [[ 6,  7,  8],
       [ 9, 10, 11]])]

>>> x.ravel()
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
       10, 11])
```

NOTE: Pay attention that 1-dimensional arrays (shape (n,)) are NOT row vectors (shape (1, n))

We will represent data mainly as **column** vectors, and data matrices will consist of horizontally stacked column vectors

Many ML libraries adopt the opposite convention — data points are represented as row vectors, and data matrices are made up of vertically stacked column vectors

Column vectors:

- Natural correspondence with mathematical notation: $y = Ax$
- Some operations are slightly slower, unless we use a Fortran-style (column major) ordering

Row vectors:

- The elements of a single data-point are consecutive (C-style, row major ordering) — some operations can be slightly faster
- A data matrix can be iterated using a for loop (however, the iterated elements will be 1-dimensional arrays)

Numpy

Arrays can be transposed using the .T attribute

```
>>> x = numpy.arange(3).reshape((3,1))
>>> x
array([[0],
       [1],
       [2]])

>>> x.T
array([[0, 1, 2]])
```

Transposition can be applied also to n-dimensional array — if interested check the documentation for the semantic

Numpy

Numpy also provides functions (and methods) to perform reduction operations (e.g. sum or product of all elements, maximum, minimum)

These functions operate over the whole array as if it were one-dimensional

```
>>> x = numpy.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.sum()
21
>>> x.max()
6
```

If we want to perform the operation over a specific axis, we can specify the axis parameter

```
>>> x = numpy.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.sum(axis=0) # sum of rows
array([5, 7, 9])
>>> x.sum(axis=1) # sum of columns
array([ 6, 15])
```

Numpy

Numpy also provides a set of element-wise functions that can be applied to all elements of an array, creating a new array

```
>>> x = numpy.array([[1,2,3],[4,5,6]])
>>> numpy.exp(x)
array([[ 2.71828183,    7.3890561 ,
        20.08553692],
       [ 54.59815003,  148.4131591 ,
        403.42879349]])
>>> numpy.log(x)
array([[ 0.          ,  0.69314718,  1.09861229],
       [ 1.38629436,  1.60943791,  1.79175947]])
```

NOTE: Whenever possible, it's best to avoid iterating explicitly over array elements

The Python interpreter is slow, and loops are expensive

Numpy functions are implemented directly in C, and therefore loops are executed much faster

Numpy

script1.py

```
import numpy
s = 0
N = 100000000
x = numpy.arange(N)
for i in x:
    s += i
```

script2.py

```
import numpy
N = 100000000
x = numpy.arange(N)
s = x.sum()
```

```
$> time python script1.py
real 0m3.347s
```

```
$> time python script2.py
real 0m0.127s
```

Arrays can be sliced

For 1-dimensional arrays, this is similar to Python lists

```
>>> x = numpy.arange(5)
```

```
>>> x[1:3]
```

```
array([1, 2])
```

```
>>> x[::-2]
```

```
array([0, 2, 4])
```

```
>>> x[3]
```

```
3
```

Numpy

Multidimensional arrays allow specifying a slice for each axis

Note: if we specify a single value for an axis, we get an array with one dimension less

```
>>> x = numpy.arange(15).reshape(3,5)

>>> x
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

>>> x[1, 0:3]
array([5, 6, 7])
```

If we want a 1-row matrix, we have to specify also the end of the slice:

```
>>> x[1:2, 0:3]
array([[5, 6, 7]])
```

Numpy allows for advanced indexing

In addition to slices, we can use numpy integer arrays and boolean arrays for indexing

Using complex indexing is not always straightforward, we will mainly use 1-dimensional indices — You can check on the documentation for further information on advanced indexing

Numpy

We can provide an index array in place of a slice:

```
>>> idx = numpy.array([0, 0, 2])
```

```
>>> x[idx, :]
array([[ 0,  1,  2,  3,  4],
       [ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])
```

```
>>> x[:, idx]
array([[ 0,  0,  2],
       [ 5,  5,  7],
       [10, 10, 12]])
```

Numpy

NOTE: `x[idx, jdx]` will keep the elements whose indices are the pairs of values (`idx[i], jdx[i]`)

If we want to extract all rows in `idx` and all columns in `jdx`, we can either use a two-step approach, use the `ix_` function, or reshape the index arrays

```
>>> idx = numpy.array([0, 2])
>>> jdx = numpy.array([1, 3])

>>> x[idx, jdx]
array([ 1, 13])

>>> x[idx, :][:, jdx]
array([[ 1,  3],
       [11, 13]])
```

Numpy

NOTE: Since `idx` and `jdx` are 1-dimensional, `x[idx, jdx]` will keep the elements whose indices are the pairs of values (`idx[i], jdx[i]`)

If we want to extract all rows in `idx` and all columns in `jdx`, we can use a two-step approach, use the `ix_` function, or reshape the index arrays

```
>>> x[numpy.ix_(idx, jdx)]
array([[ 1,  3],
       [11, 13]])

>>> x[idx.reshape((2,1)), jdx.reshape((1,2))]
array([[ 1,  3],
       [11, 13]])
```

Indexing can also use boolean arrays

```
>>> xMask = x > 5
>>> xMask
array([[False, False, False, False, False],
       [False, True, True, True, True],
       [ True, True, True, True, True]], dtype=bool)

>>> x[xMask]
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14])

>>> z=numpy.array([1,0,1], dtype=numpy.bool)
>>> z
array([ True, False,  True], dtype=bool)

>>> x[z]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])
```

Slices and indexing can be also used to assign values to an array

```
>>> x = numpy.zeros(6)
>>> x[::2] = 3
>>> x
array([ 3.,  0.,  3.,  0.,  3.,  0.])

>>> x[numpy.array([True, False, True, True, True, False],
      dtype=numpy.bool)] = numpy.arange(4)
>>> x
array([ 0.,  0.,  1.,  2.,  3.,  0.])
```

NOTE: $x[:] = 0$ modifies the elements of the array, $x=0$ binds name x to value 0

In general, slicing creates array **views**

A view is an array that shares its data with a different one

We can also create explicit views

Modification to a view modifies the original array

```
>>> x = numpy.arange(5)
>>> x
array([0, 1, 2, 3, 4])

>>> y = x[0:3]
>>> y[:] = 3
>>> x
array([3, 3, 3, 3, 4])
```

Advanced indexing creates copies of an array

We can also explicitly create a copy using the method `copy`, or creating a new array through `numpy.array`

If in doubt, we can check whether an array owns its data:

```
>>> x = numpy.arange(5)
>>> x.flags.owndata
True
```

```
>>> y = x[0:3]
>>> y.flags.owndata
False
```

An important feature of numpy array is **broadcasting**

Broadcasting allows applying elementwise operations, such as addition and multiplication, to arrays with different shapes

Whenever arrays have different shapes:

- 1's will be prepended to the shapes of smaller arrays until all arrays have the same number of dimensions
- Axes with shape 1 are treated as if they had the same dimension as the array with largest size along the axis, and all elements were the same along the axis

Of course, numpy arrays should have the same dimensions after broadcasting

Broadcasting examples: adding the same values to each row of a 2-D array

```
>>> x = numpy.zeros((3,4))
>>> m = numpy.arange(4)
>>> x + m
array([[ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.]])
```

Array m is broadcasted to a shape $(1, 4)$, and then replicated along axis 0

Broadcasting examples: adding the same values to each row of a 2-D array - alternative

```
>>> x = numpy.zeros((3,4))
>>> m = numpy.arange(4).reshape((1,4))
>>> x + m
array([[ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.]])
```

Array m already has the same number of dimensions as x , and gets replicated along axis 0

Broadcasting examples: adding the same values to each column of a 2-D array — note that, in this case, m has to be a column vector

```
>>> x = numpy.zeros((3,4))
>>> m = numpy.arange(3).reshape((3,1))
>>> x + m
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.]])
```

Array m already has the same number of dimensions as x , and gets replicated along axis 1

Broadcasting is widely used, so get familiar with it

You can find the details at

<https://numpy.org/doc/stable/user/basics.broadcasting.html>

Numpy

Numpy arrays can be concatenated

We can use hstack and vstack to stack vectors horizontally or vertically

For 2-D arrays:

```
>>> x1 = numpy.array([[1,2,3]])
>>> x2 = numpy.array([[4,5,6]])

>>> numpy.hstack([x1, x2])
array([[1, 2, 3, 4, 5, 6]])

>>> numpy.vstack([x1, x2])
array([[1, 2, 3],
       [4, 5, 6]])
```

hstack and vstack can be used for N-dimensional arrays

hstack concatenates arrays along axis 1

vstack concatenates arrays along axis 0

We can also use the function concatenate — we need to specify the axis

- `numpy.hstack(l)` is equivalent to `numpy.concatenate(l, axis = 1)`
- `numpy.vstack(l)` is equivalent to `numpy.concatenate(l, axis = 0)`

Numpy

Numpy also provides several linear algebra functions

These can be found inside `numpy.linalg`

For example, we can compute the eigenvalue decomposition of a 2-D array:

```
>>> x = numpy.arange(9).reshape(3,3)
>>> numpy.linalg.eig(x)
(array([ 1.33484692e+01, -1.34846923e+00,
        -2.48477279e-16]),
 array([[ 0.16476382,  0.79969966,  0.40824829],
        [ 0.50577448,  0.10420579, -0.81649658],
        [ 0.84678513, -0.59128809,  0.40824829]]))
```

We will discuss the different functionalities as needed

In many cases it will be useful to plot functions, data points and so on

Several libraries are available, we will use `matplotlib`

The library has a huge number of functionalities, we won't go into details much

The main module we will consider is `pyplot`

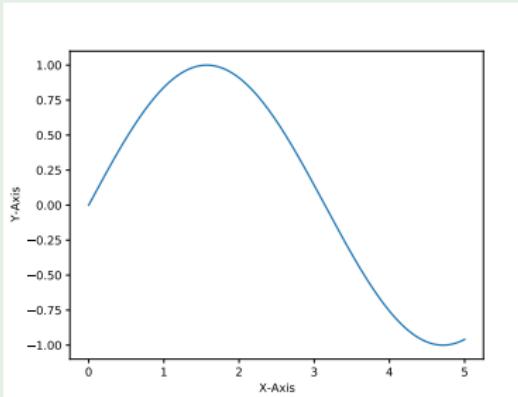
Tutorials:

- <https://matplotlib.org/3.3.3/tutorials/index.html>
- <https://matplotlib.org/3.3.3/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py>

Matplotlib

Plotting values can be achieved through the plot function

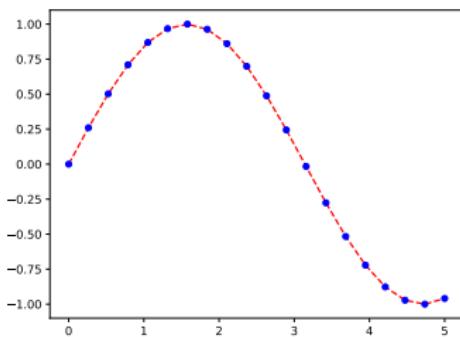
```
import matplotlib.pyplot as plt
x = numpy.linspace(0, 5, 1000)
plt.plot(x, numpy.sin(x))
plt.xlabel('X-Axis')
plt.ylabel('Y-Axis')
plt.show()
```



Matplotlib

We can specify color, line style (solid, dashed, points, ...), style attributes (e.g. line width) and so on

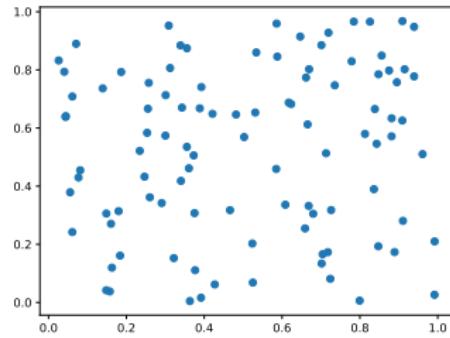
```
import matplotlib.pyplot as plt
x = numpy.linspace(0, 5, 20)
plt.plot(x, numpy.sin(x), color='r', linestyle='--')
plt.plot(x, numpy.sin(x), 'bo', markersize = 5)
plt.show()
```



Matplotlib

We can visualize 2-D data using scatter plots

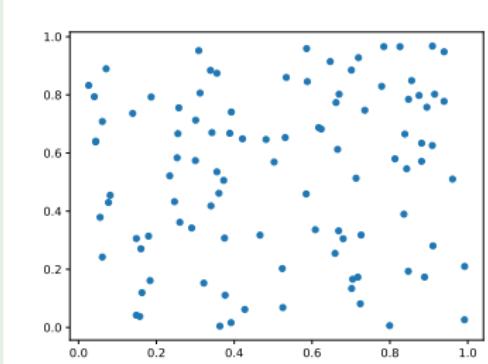
```
import matplotlib.pyplot as plt  
D = numpy.random.random((2, 100))  
plt.scatter(D[0], D[1])  
plt.show()
```



Matplotlib

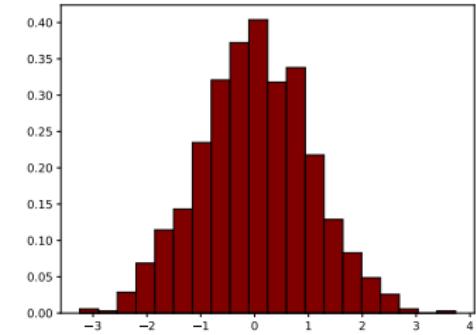
Using plot:

```
import matplotlib.pyplot as plt
D = numpy.random.random((2, 100))
plt.plot(D[0], D[1], linestyle='', marker='.',
          markersize=10)
plt.show()
```



Visualizing data distributions through histograms:

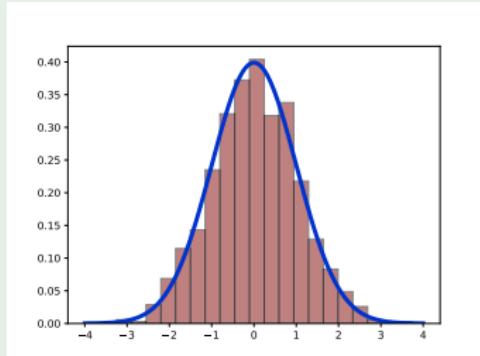
```
import matplotlib.pyplot as plt  
D = numpy.random.normal(size=1000)  
plt.hist(D, bins = 20, density=True, ec='black',  
         color ='#800000')  
plt.show()
```



Matplotlib

We can add multiple plots to the same figure. `show` is used to show a picture after all elements have been added

```
import matplotlib.pyplot as plt
D = numpy.random.normal(size=1000)
plt.hist(D, bins = 20, density=True, ec='black', color='#800000',
         alpha = 0.5)
x = numpy.linspace(-4, 4, 1000)
y = 1.0/(2*numpy.pi)**0.5 * numpy.exp(-0.5 * x**2)
plt.plot(x, y, color=(0.0, 0.2, 0.8), linewidth=4)
plt.show()
```



We will discuss additional functionalities as needed

Lots of examples and material are also available on the web

Machine Learning and Pattern Recognition

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Pattern Recognition:

Automatic discovery of regularities in data through the use of computer algorithms (...) to take actions such as classifying the data into different categories.¹

Machine 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 .²

¹C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

²T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

Define models that are able to capture the regularities in our data and allow performing inference about the properties we are interested in.

The models should not simply specify a set of human-defined rules, but should be able to learn from data.

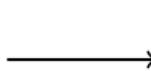
The learning stage should leverage observed data to improve the quality of the inference.

Example: Classification

Given a set of objects, assign them to discrete categories

Find a mapping from the space of input vectors representing the objects to a discrete set of labels (output values)

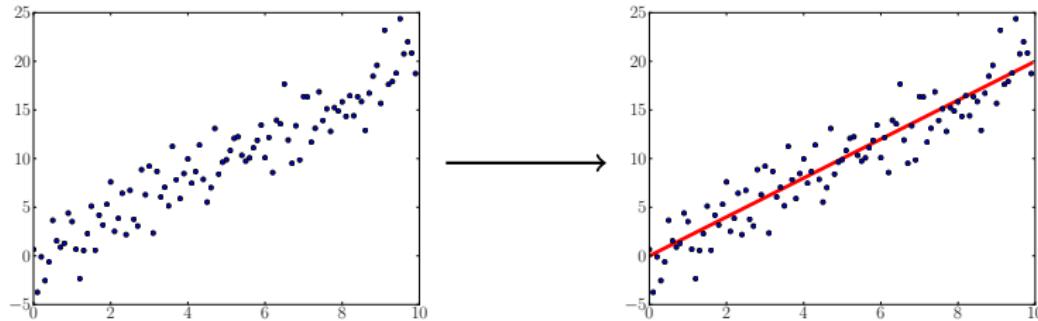
Possible applications include image and face recognition and speaker identification



Example: Regression

Similar to classification, but output values are single or multiple continuous variables

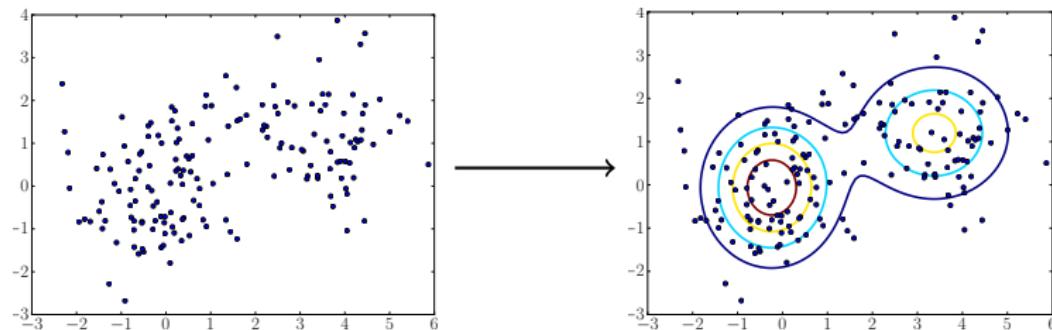
Possible applications include modeling of physical phenomena or prediction of continuous-valued functions (e.g. stock market prediction)



Example: Density Estimation

Estimation of the distribution of input vectors

Often used to explore data characteristics and to build inference models



Depending on the task, we identify two main branches

Supervised learning: Along with the *input* data the system is provided with *output* data (e.g. class labels, function values, ...). The goal is estimating a mapping between input and output data.

Common supervised tasks:

- Classification
- Regression

Depending on the task, we identify two main branches

Unsupervised learning: No feedback is provided to the system, whose goal is to identify some (useful) structure of the data.

Examples of unsupervised tasks:

- Clustering
- Density estimation
- Dimensionality reduction

Unsupervised methods are often used as pre-processing steps for supervised learning tasks (e.g. unsupervised dimensionality reduction).

Furthermore, some techniques are common to both supervised and unsupervised tasks (e.g. density estimation for generative classification models)

In this course we will focus on classification and density estimation tasks.

- Assign a **pattern** to a **class**
- **Classes** represent characteristics of the objects that we are considering

For example

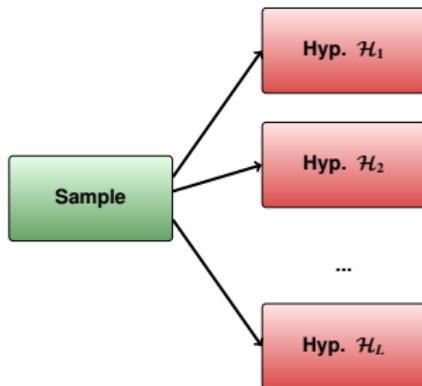
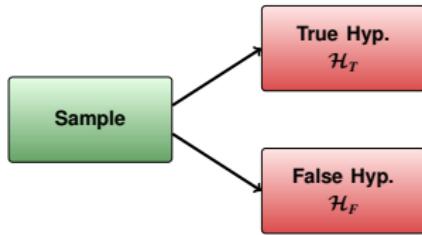
- What object is contained in an image
- The language of a text
- Tomorrow's weather
- ...

Pattern classification

- **Binary** classification
- **Multiclass** classification

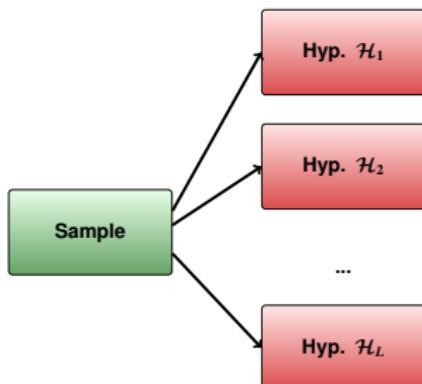
Pattern classification

- **Binary** classification
 - Identity verification
 - Intrusion detection
- **Multiclass** classification
 - Object categorization
 - Speech decoding

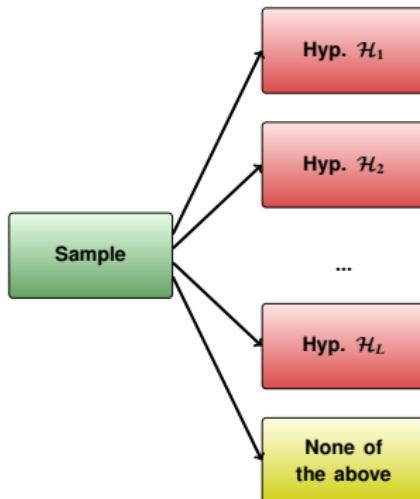


Pattern classification

- Closed-set classification



- Open-set classification



Pattern Classification

It is convenient to divide the problem in three different stages



In practice, these components often interact

- A neural network model can perform all the stages simultaneously
- Dimensionality reduction is often considered part of the feature extraction process (e.g. PCA)
- Dimensionality reduction techniques can be used to compute classification boundaries (e.g. LDA)

Feature Extraction



We represent an object in terms of numerical attributes, usually arranged as vectors or matrices



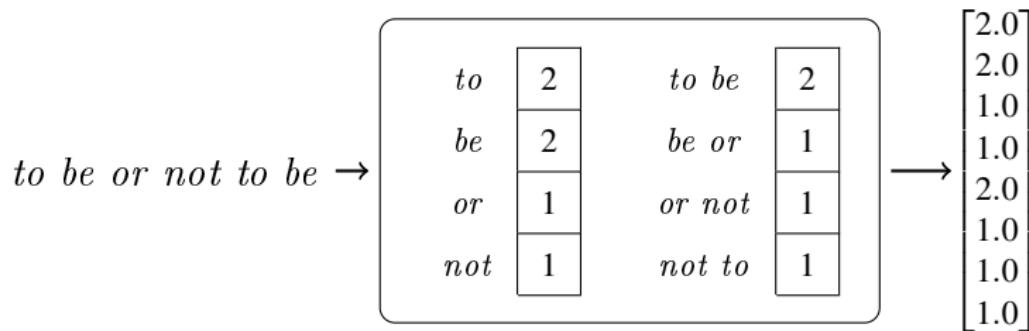
→

$$\begin{bmatrix} 116 & 133 & 149 & \dots & 117 & 128 & 136 \\ 86 & 107 & 126 & \dots & 116 & 132 & 144 \\ 70 & 91 & 112 & \dots & 117 & 133 & 144 \\ \vdots & & \ddots & & \vdots \\ 96 & 101 & 115 & \dots & 139 & 138 & 139 \\ 92 & 102 & 118 & \dots & 139 & 138 & 138 \\ 93 & 104 & 118 & \dots & 138 & 137 & 136 \end{bmatrix}$$

Feature Extraction



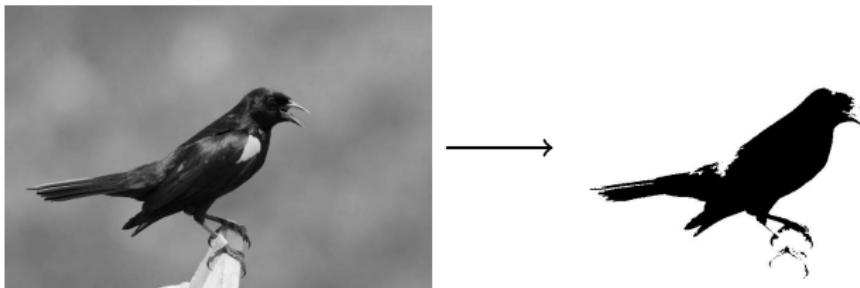
We represent an object in terms of numerical attributes, usually arranged as vectors or matrices



Feature Extraction



Often it involves (complex) manipulations of the object to extract useful representations



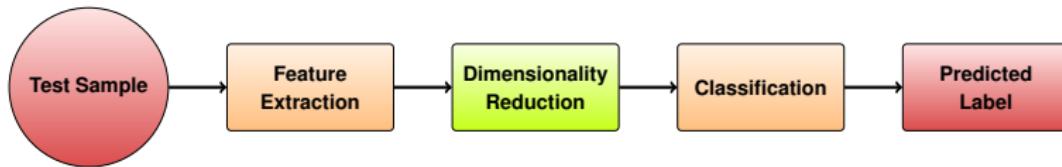
Dimensionality Reduction



The feature space is often very large and contains a large amount of unwanted and potentially harmful information.

Dimensionality reduction techniques compute a mapping from the n -dimensional feature space to a m -dimensional space, with $m \ll n$.

Dimensionality Reduction



- Compress information
- Remove unwanted variability (noise)
- Data visualization

Dimensionality Reduction

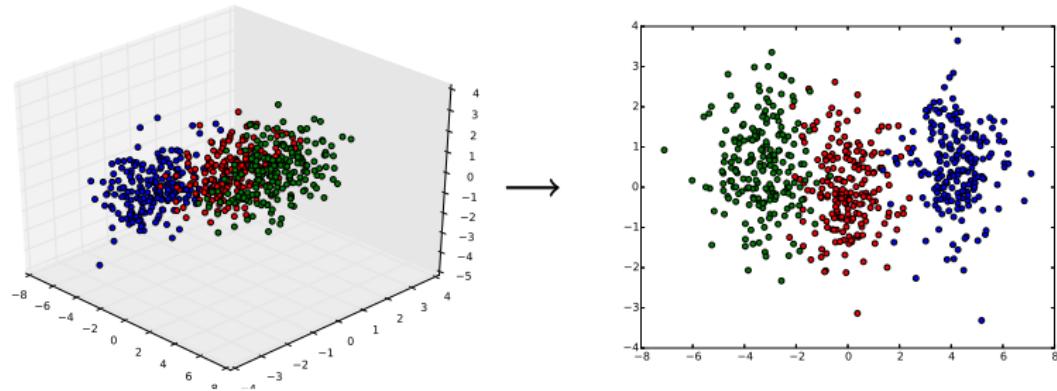


- Simplify classification (reduce curse of dimensionality, reduce overfitting)

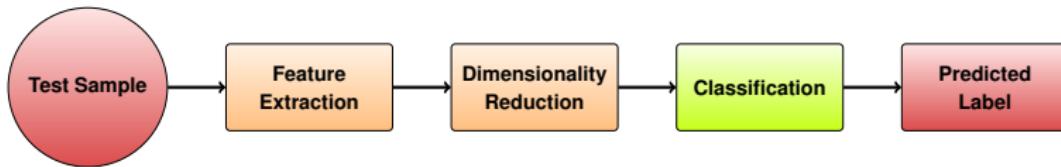
Overfitting: An over-complex model fits very accurately the observed data (very small training error), but is not able to provide accurate predictions for unseen data

Curse of dimensionality: Volumes in high-dimensional spaces grow very fast, and data becomes very sparse

Dimensionality Reduction Classification



Classification



Given the (possibly reduced) feature vector representing an object, associate a label to the object based on the properties of the representation

Perform a mapping from the m -dimensional feature space to the space of labels

The mapping is also called decision function

Classification



Our goal is to design suitable decision functions

- ³**Discriminant model**: directly construct a function $f(x)$ mapping feature vector x directly to a label
- **Discriminative non-probabilistic model**: construct a function $f(x)$ mapping feature vector x to a set of “scores”

³C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

Classification



Our goal is to design suitable decision functions

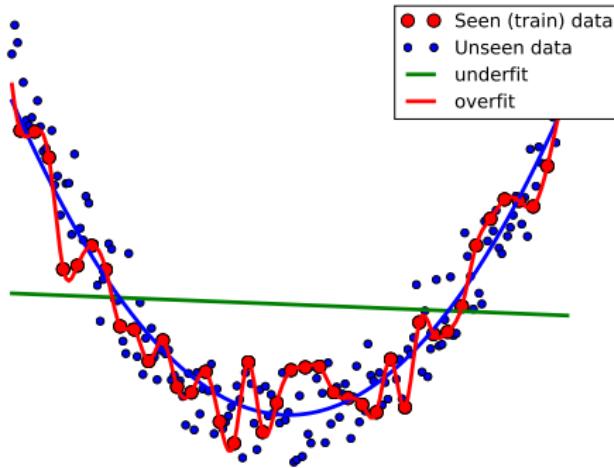
- **Discriminative probabilistic model:** model class **posterior probabilities** $P(\mathcal{C}_k|\mathbf{x})$ and assign labels according to posterior probabilities
- **Generative probabilistic model:** model the **joint distribution** of features and labels $P(\mathbf{x}, \mathcal{C}_k) = P(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k)$ and apply Bayes theorem to compute posterior probabilities $P(\mathcal{C}_k|\mathbf{x})$

Classification

The decision functions should provide good **generalization** error

We want models that are able to provide good predictions on **unseen** data (generalization)

Models describe data — too simple models can **underfit** our data, too complex models can **overfit** our data



Generative probabilistic model:

- The model describes the data generation process in terms of **class-conditional** distributions $P(\mathbf{x}|\mathcal{C}_k)$
- The model can incorporate **application-dependent** prior class information $P(\mathcal{C}_k)$
- Inference is based on Bayes theorem

$$P(\mathcal{C}_k|\mathbf{x}) = \frac{P(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k)}{P(\mathbf{x})}$$

- Class assignment is based on highest posterior probability $P(\mathcal{C}_k|\mathbf{x})$
- Probabilistic interpretation, optimal decisions depending only on prior information

Generative probabilistic model:

- Most demanding — it requires modeling per-class densities of our data
- Most informative — it allows us to *generate* new samples

Discriminative probabilistic model:

- Directly model class posterior probability $P(\mathcal{C}_k | \mathbf{x}_t)$
- Class assignment is again based on highest posterior probability
- Probabilistic interpretation
- Cannot directly incorporate application-dependent information – class prior probabilities are embedded in the model (though in many cases they can be compensated)

Discriminative non-probabilistic model:

- The output is a score s
- The score can be taken as a measure of strength of the hypotheses (possible class assignments) under test
- Class assignment in general is based on best (usually highest) score
- Cannot directly account for prior information (in some cases it's possible to recover probabilistic interpretations)

Discriminant function:

- Directly outputs the class label for pattern x
- Does not allow measuring uncertainty for our predictions
- Cannot be adapted to take into account application costs and prior information

We will focus (mainly) on probabilistic models

Solving the inference problem

Create a **model** \mathcal{M} describing relationships between features and labels

- **Parametric** models: The model depends on a set of parameters θ whose size does not depend on the available data (we will focus on these models)
- **Non-parametric** models: The model complexity grows with the sample size

Create a (labeled) **training** set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$

- The set contains the observed samples that will be used for the *learning* stage

Solving the inference problem

Learn the model parameters

- Frequentist / non-probabilistic approaches: estimate optimal values for the model parameters using the training set:

$$\mathcal{D}, \mathcal{M} \rightarrow \boldsymbol{\theta}^*$$

- Usually based on the optimization of a suitable **objective** function
- Bayesian probabilistic approach: update the prior belief over the model parameters using the observed data

$$\mathcal{D}, \mathcal{M}, P(\boldsymbol{\theta}|\mathcal{M}) \rightarrow P(\boldsymbol{\theta}|\mathcal{D}, \mathcal{M})$$

Solving the inference problem

Predict the class label y_t for a (previously unseen) test sample x_t

- Generative model: compute the class-conditional probabilities for the sample $P(x_t|\mathcal{C}_k, \mathcal{D}, \mathcal{M})$
 - Frequentist approach:

$$P(x_t|\mathcal{C}_k, \mathcal{D}, \mathcal{M}) \approx P(x_t|\mathcal{C}_k, \boldsymbol{\theta}^*, \mathcal{M})$$

- Bayesian approach:

$$P(x_t|\mathcal{C}_k, \mathcal{D}) = \int P(x_t|\mathcal{C}_k, \boldsymbol{\theta}, \mathcal{M})P(\boldsymbol{\theta}|\mathcal{D}, \mathcal{M})d\boldsymbol{\theta}$$

Solving the inference problem

Predict the class label y_t for a (previously unseen) text sample x_t

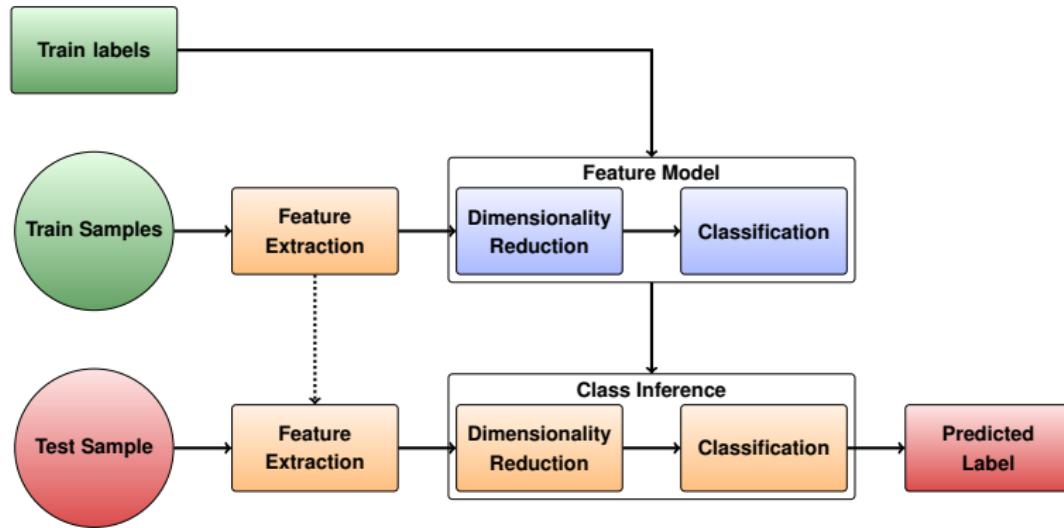
- Discriminative model: compute posterior class probabilities for the sample $P(\mathcal{C}_k|x_t, \mathcal{D}, \mathcal{M})$
 - Frequentist approach:

$$P(\mathcal{C}_k|x_t, \mathcal{D}, \mathcal{M}) \approx P(\mathcal{C}_k|x_t, \boldsymbol{\theta}^*, \mathcal{M})$$

- Bayesian approach:

$$P(\mathcal{C}_k|x_t, \mathcal{D}) = \int P(\mathcal{C}_k|x_t, \boldsymbol{\theta}, \mathcal{M})P(\boldsymbol{\theta}|\mathcal{D}, \mathcal{M})d\boldsymbol{\theta}$$

Machine learning for pattern classification



Machine learning for pattern classification

We also need to be able to assess the quality of our predictions

We are interested in performance on **unseen** data

Since we cannot know the system performance on unseen data,
we **simulate** its behavior on *known* data

We use a (labeled) **evaluation** or **test** set

Machine learning for pattern classification

The test set should mimic real use cases

- It should contain data similar to that of our use case
- It should not contain data that was used to train the model
- Ideally, it should not be used to make any decision on the system

In practice, test sets are usually employed to compare the performance of different systems

Although unavoidable for practical reasons, with time this may lead to biased conclusions

Machine learning for pattern classification

Models and / or training procedures often contain *hyperparameters*, for example

- Number of dimensions of reduced features
- Step size in gradient-descent based optimization
- Regularization coefficients

The optimal value of these hyperparameters usually cannot be estimated using the same criterion employed for the model parameters

In other cases, we may have different competing models, and we would like to select the one that provides the best predictions

Machine learning for pattern classification

We need a way to perform **model selection**

However, we cannot use test data to select good hyperparameters values, as this would bias our evaluation

We make use of an additional training set, called **validation** set.

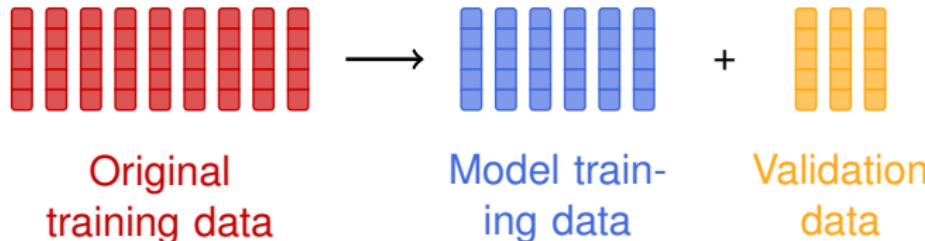
The validation set is used to asses the influence of model hyperparameters on prediction accuracy in order to select good hyperparameter values.

Machine learning for pattern classification

We can build a validation set by extracting some data from the training set

Part of the training set will be used for estimating our models

The remaining part (validation set) will simulate the evaluation data, and will be used to infer hyperparameters and for model selection



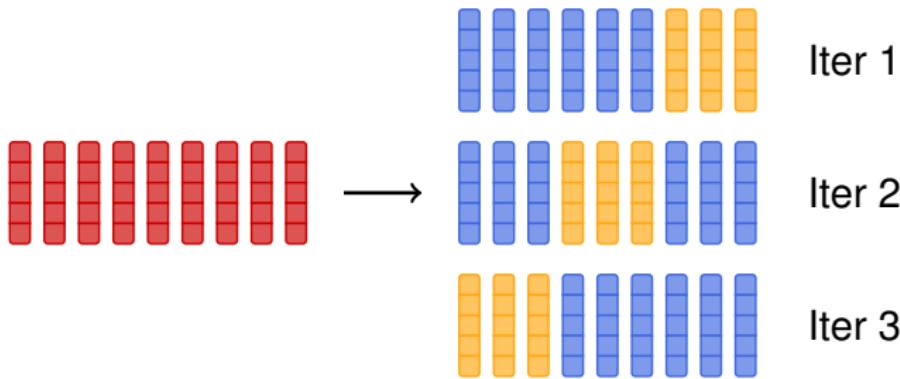
When training data is scarce, **cross-validation** is usually employed

K-fold cross-validation:

- Subdivide the training data in K folds
- Repeatedly use $K - 1$ folds as training data, the remaining fold as validation set
- Combine the validation results on the K folds and select optimal values for hyperparameters
- Retrain the model using the selected hyperparameters but using all data

Machine learning for pattern classification

K-Fold: Repeatedly use $K - 1$ folds as training data, the remaining fold as validation set



Small values of K require less computational resources (fewer models to train), but each model is trained with less data

Large values of K require training many models, but each model will be much more similar to the one trained over all data

Limiting case: leave-one-out approach

- Similar to K-fold, but each evaluation partition contains a single sample

In some experimental protocols, validation sets are extracted from the evaluation set

- Higher similarity with evaluation data
- We have to guarantee that the data does not overlap
- Validation data becomes, in all aspects, training material
- In real use cases, evaluation data may not be available, so the protocol may not reflect actual usage of the models. This may lead to biased results

For our **projects**, validation data can be extracted from the training set. **Evaluation data** should **not be used** to estimate **anything**

Dimensionality Reduction

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Dimensionality Reduction

Dimensionality reduction techniques compute a mapping from the n -dimensional feature space to a m -dimensional space, with $m \ll n$

- Compress information
- Remove unwanted variability (noise)
- Simplify classification (reduced effects due to high dimensionality, reduced risk of overfitting)
- Data visualization

Dimensionality Reduction

Different goals may require different approaches

- Compress information: we want to retain the **maximum amount of information** for a given output size
- Improve classification: we want to retain **discriminant information**
- Data visualization: we want 2-D or 3-D representations that preserve as much as possible relationships between different samples

Dimensionality Reduction

We will focus on two linear methods:

- Unsupervised: Principal Component Analysis (PCA)
- Supervised: Linear Discriminant Analysis (LDA)

In both cases, we want to find a subspace of the feature space that preserves most of the “useful” information

Notes on linear algebra

We recall some linear algebra notions for real matrices that will be used in the following

For a quick reference about matrix properties, you can also refer to

Petersen, Pedersen, "The Matrix Cookbook",

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.113.6244>

Notes on linear algebra

Let A be a square symmetric $n \times n$ matrix $A \in \mathbb{R}^{n \times n}$

A admits an eigen-decomposition

$$A = V\Sigma V^{-1} = V\Sigma V^T$$

- V is an orthogonal $n \times n$ matrix whose columns are the (right) eigenvectors of A
- Σ is a diagonal $n \times n$ matrix whose elements are the eigenvalues of A

Since V is orthogonal, $V^T V = VV^T = I$ and $V^{-1} = V^T$

Notes on linear algebra

A generic rectangular matrix $A \in \mathbb{R}^{n \times m}$ always admits a **Singular Value Decomposition** (SVD) of the form:

$$A = U\Sigma V^T$$

where

- U is an orthogonal $n \times n$ matrix of eigenvectors of AA^T
- V is an orthogonal $m \times m$ matrix of eigenvectors of A^TA
- Σ is a diagonal rectangular matrix containing the **singular values** of A (in decreasing order)

Notes on linear algebra

$$A = U \Sigma V$$

$$\Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_m \\ & 0 & & \end{bmatrix}$$

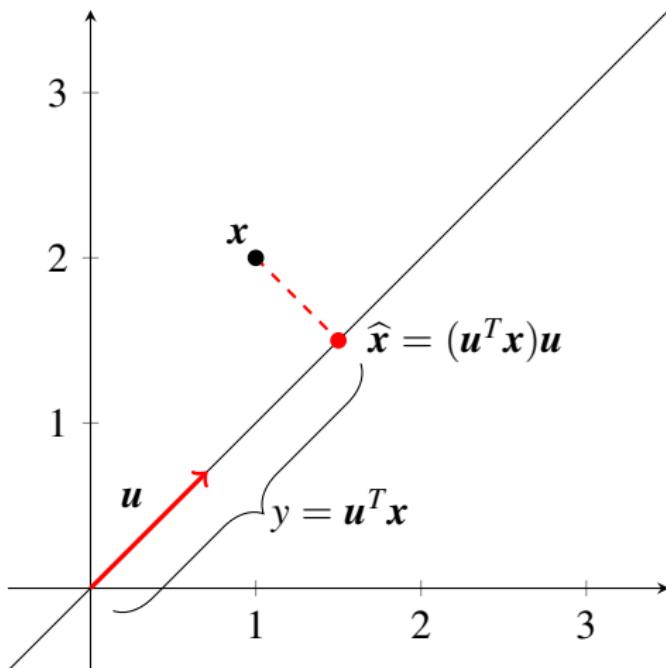
Notes on linear algebra

$$\boxed{A} = \boxed{U} \boxed{\Sigma} \boxed{V}$$

$$\Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_m \end{bmatrix} \mathbf{0}$$

Notes on linear algebra

Projecting a point over a direction u :



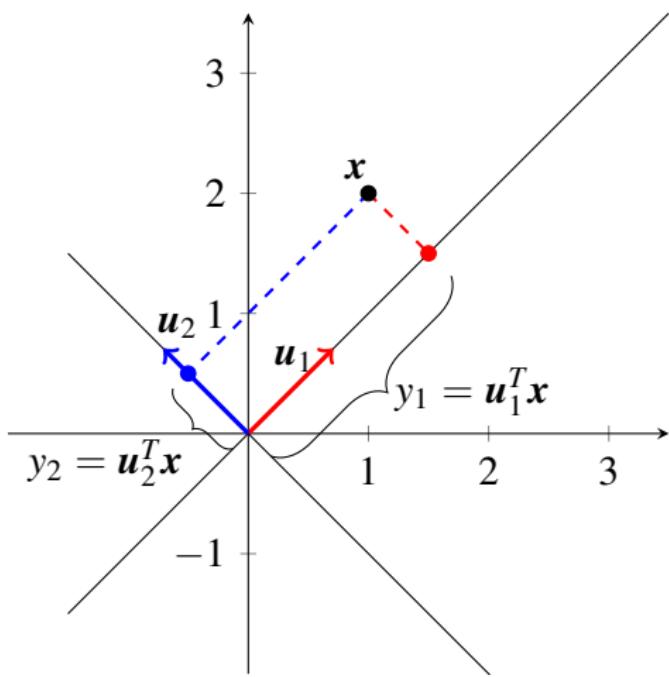
u is a **unit** vector representing a direction

$y = u^T x$ is the **projection** of x over u

$\hat{x} = yu = (u^T x)u$ is the representation of the projected point in the original space (reconstruction)

Notes on linear algebra

Projecting a point in an m -dimensional (sub)space $\mathbf{U} = [\mathbf{u}_1 \dots \mathbf{u}_m]$ ($m = 2$ in the example):



The columns of $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2]$ form a **basis** of \mathbb{R}^2

$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1^T \mathbf{x} \\ \mathbf{u}_2^T \mathbf{x} \end{bmatrix} = \mathbf{U}^T \mathbf{x}$ is the **projection** of \mathbf{x} over the column space of \mathbf{U}

The representation (reconstruction) of \mathbf{y} in the original space can be obtained as

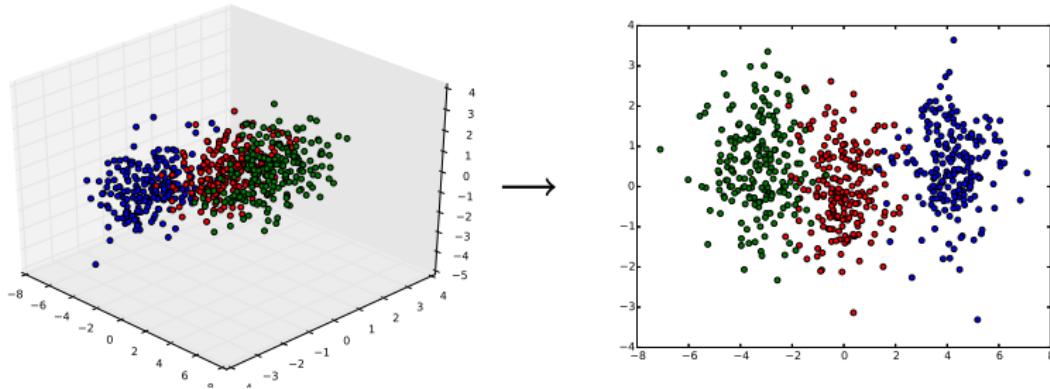
$$\hat{\mathbf{x}} = y_1 \mathbf{u}_1 + y_2 \mathbf{u}_2 = \mathbf{U}\mathbf{y} = \mathbf{U}\mathbf{U}^T \mathbf{x}$$

In this example, since \mathbf{U} is full rank, $\hat{\mathbf{x}} = \mathbf{x}$

Principal Component Analysis

We are given a zero-mean dataset $X = \{x_1, \dots, x_K\}$, with $x_i \in \mathbb{R}^n$

We want to find the subspace of \mathbb{R}^n that allows preserving *most* of the information



Principal Component Analysis

A subspace can be represented as a matrix $P \in \mathbb{R}^{n \times m}$ whose columns are **orthonormal**

The columns of P form a basis of a subspace of \mathbb{R}^n with dimension m

The projection of x over the subspace is given by $y = P^T x$:

$$y = \begin{bmatrix} p_1^T x \\ p_2^T x \\ \vdots \\ p_m^T x \end{bmatrix}$$

where $p_1 \dots p_m$ are the columns of P

We can compute the coordinates of the projected point y in the original space as $\hat{x} = Py$

Principal Component Analysis

We need to define a criterion for estimating P

A reasonable criterion may be the minimization of the **average reconstruction error** (K is the number of samples)

$$\frac{1}{K} \sum_{i=1}^K \|x_i - \hat{x}_i\|^2 = \frac{1}{K} \sum_{i=1}^K \|x_i - Py_i\|^2 = \frac{1}{K} \sum_{i=1}^K \|x_i - PP^T x_i\|^2$$

We therefore want to solve

$$P^* = \arg \min_P \frac{1}{K} \sum_{i=1}^K \|x_i - \hat{x}_i\|^2 = \arg \min_P \frac{1}{K} \sum_{i=1}^K \|x_i - PP^T x_i\|^2$$

Principal Component Analysis

We can rewrite the objective function as

$$\begin{aligned}\mathcal{L}(\mathbf{P}) &= \frac{1}{K} \sum_{i=1}^K \|\mathbf{x}_i - \mathbf{P}\mathbf{P}^T\mathbf{x}_i\|^2 = \frac{1}{K} \sum_{i=1}^K (\mathbf{x}_i^T \mathbf{x}_i - 2\mathbf{x}_i^T \mathbf{P}\mathbf{P}^T \mathbf{x}_i + \mathbf{x}_i^T \mathbf{P}\mathbf{P}^T \mathbf{P}\mathbf{P}^T \mathbf{x}_i) \\ &= \frac{1}{K} \sum_{i=1}^K (\mathbf{x}_i^T \mathbf{x}_i - 2\mathbf{x}_i^T \mathbf{P}\mathbf{P}^T \mathbf{x}_i + \mathbf{x}_i^T \mathbf{P}\mathbf{P}^T \mathbf{x}_i) \\ &= \frac{1}{K} \sum_{i=1}^K (\mathbf{x}_i^T \mathbf{x}_i - \text{Tr}(\mathbf{P}^T \mathbf{x}_i \mathbf{x}_i^T \mathbf{P}))\end{aligned}$$

where Tr represents the trace of a matrix, and we used the fact that $\mathbf{P}^T \mathbf{P} = \mathbf{I}$

Here we used the fact that, for any two vectors \mathbf{v} and \mathbf{w}

$$\mathbf{v}^T \mathbf{w} = \text{Tr}(\mathbf{v}^T \mathbf{w})$$

and the property that

$$\text{Tr}(\mathbf{AB}) = \text{Tr}(\mathbf{BA})$$

Principal Component Analysis

Since Tr is a linear operator, minimizing \mathcal{L} is equivalent to maximizing

$$\widehat{\mathcal{L}}(\mathbf{P}) = \text{Tr} \left(\mathbf{P}^T \left[\frac{1}{K} \sum_{i=1}^K \mathbf{x}_i \mathbf{x}_i^T \right] \mathbf{P} \right)$$

We require that \mathbf{P} is an orthogonal matrix

It can be shown that the optimal solution is then given by the matrix \mathbf{P} whose columns are the m eigenvectors of $\frac{1}{K} \sum_{i=1}^K \mathbf{x}_i \mathbf{x}_i^T$ corresponding to the m largest eigenvalues

Principal Component Analysis

Let

$$\frac{1}{K} \sum_{i=1}^K \mathbf{x}_i \mathbf{x}_i^T = \mathbf{U} \boldsymbol{\Sigma} \mathbf{U}^T$$

be an eigen-decomposition of the matrix, with $\boldsymbol{\Sigma}$ containing the eigen-values in **descending** order

$$\boldsymbol{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix}, \quad \sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n$$

and

$$\mathbf{U} = [\mathbf{u}_1 \dots \mathbf{u}_m, \mathbf{u}_{m+1} \dots \mathbf{u}_n]$$

Then

$$\mathbf{P}^* = [\mathbf{u}_1 \dots \mathbf{u}_m]$$

i.e., \mathbf{P}^* corresponds to the first m columns of \mathbf{U}

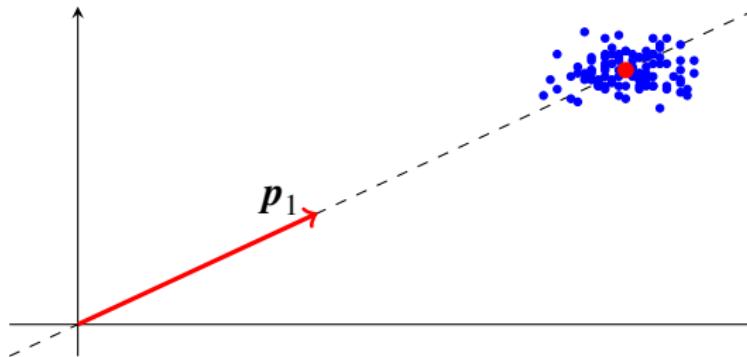
Principal Component Analysis

What happens if the dataset is not zero-mean?

P represents a subspace, whose axes pass through the origin

If the dataset is far from the origin, the first PCA direction will approximately connect the origin and the dataset mean

This direction (in most cases) is not very interesting



Principal Component Analysis

We can recast the problem as looking for the projection surface (line, plane, ...) that minimizes the reconstruction error

In practice, we can cast the problem as jointly looking for a dataset shift \mathbf{m} and a subspace \mathbf{U} over which we can project the shifted data

An optimal solution for the shift is given by the dataset mean

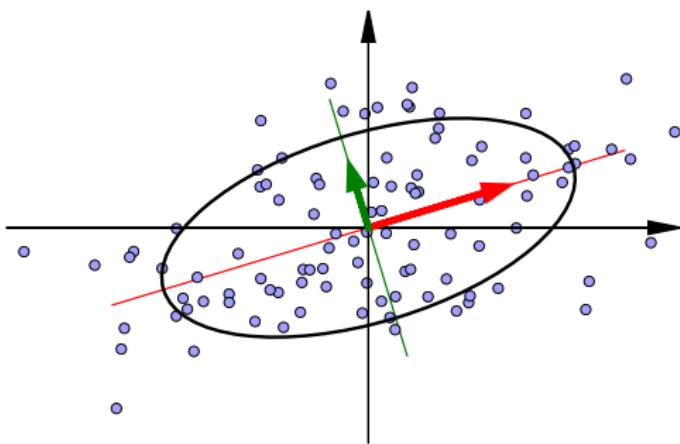
In practical terms, we remove the dataset mean before computing PCA

If the dataset mean is $\bar{\mathbf{x}}$, the PCA subspace is computed from the eigenvectors of the empirical covariance matrix

$$\frac{1}{K} \sum_i (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T$$

Principal Component Analysis

PCA can be interpreted as the linear mapping that preserves the directions with highest variance



The axes of the ellipse correspond to the principal directions

Principal Component Analysis

Consider the covariance matrix

$$\mathbf{C} = \frac{1}{K} \sum_i (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T$$

We can compute the eigen-decomposition of \mathbf{C}

$$\mathbf{C} = \mathbf{U}\Sigma\mathbf{U}^T$$

\mathbf{U} : Eigenvectors matrix

Σ : Diagonal eigenvalues matrix

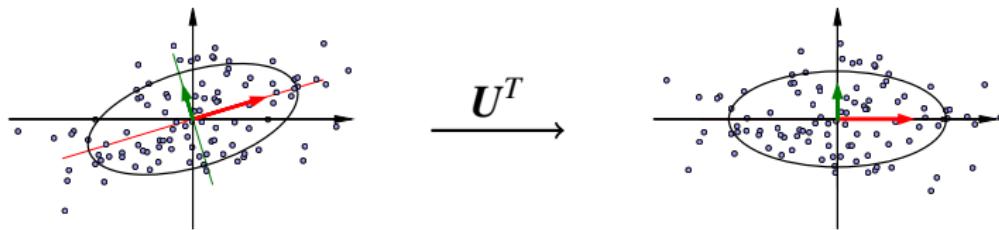
Remember that $\mathbf{U}^{-1} = \mathbf{U}^T$, thus $\mathbf{U}^T\mathbf{U} = \mathbf{I}$

Principal Component Analysis

Projecting our data centered points over \mathbf{U}^T we obtain

$$\mathbf{C}' = \frac{1}{K} \sum_i \mathbf{U}^T (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^T \mathbf{U} = \mathbf{U}^T (\mathbf{U} \Sigma \mathbf{U}^T) \mathbf{U} = \Sigma$$

Projection over \mathbf{U}^T transforms our data so that the different directions are **uncorrelated**



To keep only the m directions with highest variance we keep only the first m transformed directions

Principal Component Analysis

Usually PCA is applied directly to **centered** data

- Compute sample mean \bar{x}
- Center data: $z_i = x_i - \bar{x}$
- Compute the sample covariance matrix

$$\mathbf{C} = \frac{1}{K} \sum_i (x_i - \bar{x})(x_i - \bar{x})^T = \frac{1}{k} \sum_i z_i z_i^T$$

- Compute the eigen-decomposition of $\mathbf{C} = \mathbf{U}\Sigma\mathbf{U}^T$
- Project the data in the subspace spanned by the m columns of \mathbf{U} corresponding to the m highest eigenvalues (matrix \mathbf{P}):
 $y_i = \mathbf{P}^T z_i = \mathbf{P}^T (x_i - \bar{x})$
- Reconstruction requires inverting the process: $\hat{x}_i = \bar{x} + \mathbf{P}y_i$

Principal Component Analysis

Selection of optimal m can be done by cross-validation using a validation set

We can also select m as to retain a given percentage t (e.g. 95%) of the variance of the data

Remember that each eigenvalue corresponds to the variance along the corresponding axis

We choose m as the lowest number for which the sum of the first m eigenvalues divided by the sum of all eigenvalues is larger than t

$$\min_m m \quad \text{s.t.} \quad \frac{\sum_{i=1}^m \sigma_i}{\sum_{i=1}^n \sigma_i} \geq t$$

where σ_i is the i -th largest eigenvalue (diagonal element of Σ)

Principal Component Analysis

Computing the sample covariance can be difficult when the feature space is very large

Different solutions

- Truncated Singular Value Decomposition
- Probabilistic PCA

For smaller dataset the standard approach is sufficient

MNIST Dataset

To compare different methods we consider the MNIST handwritten digits dataset

Task: classification of handwritten digits

- Classes are digits $0, 1, \dots, 9$
- Images have already been normalized (centered and scaled to a square 28×28 shape)
- Gray-scale images (originally binary)
- 60 000 training images (6000 for each digit)
- 10 000 test images

MNIST Dataset

Train samples:

0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9

Test samples:

0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9

In general, we will consider both multi-class and pair-wise binary classification tasks

In the following we show the results of PCA applied to the images

Principal Component Analysis

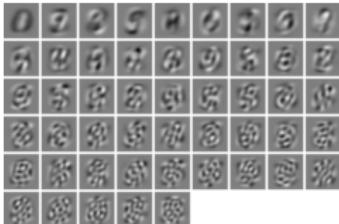
50 dimensional PCA

0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 8



0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 8

$U:$



Principal Component Analysis

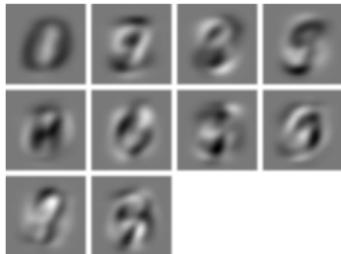
10 dimensional PCA

0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 8



0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 8

$U:$



Dimensionality Reduction

To analyze the results of PCA as pre-processing for classification we consider a very simple classifier based on Euclidean distance

For each class, we compute the **mean** vector $\mu_c = \frac{1}{n_c} \sum_{i=1}^{n_c} \mathbf{x}_{c,i}$

For a test sample x_t we predict its label as the label of the class whose mean is closest to the test sample itself:

$$c_t = \arg \min_c \|(\mathbf{x}_t - \mu_c)\|^2$$

Dimensionality Reduction

We measure the error rate as the number of incorrect classified samples over the total number of samples (we will see better measures in the next weeks)

	m^1				
w/o PCA	100	50	9	5	
18.0%	18.1%	18.2%	25.5%	35.9%	

Most of the dimensions can be safely removed — the results with 50 and 100 dimensions are very close to those of the full image

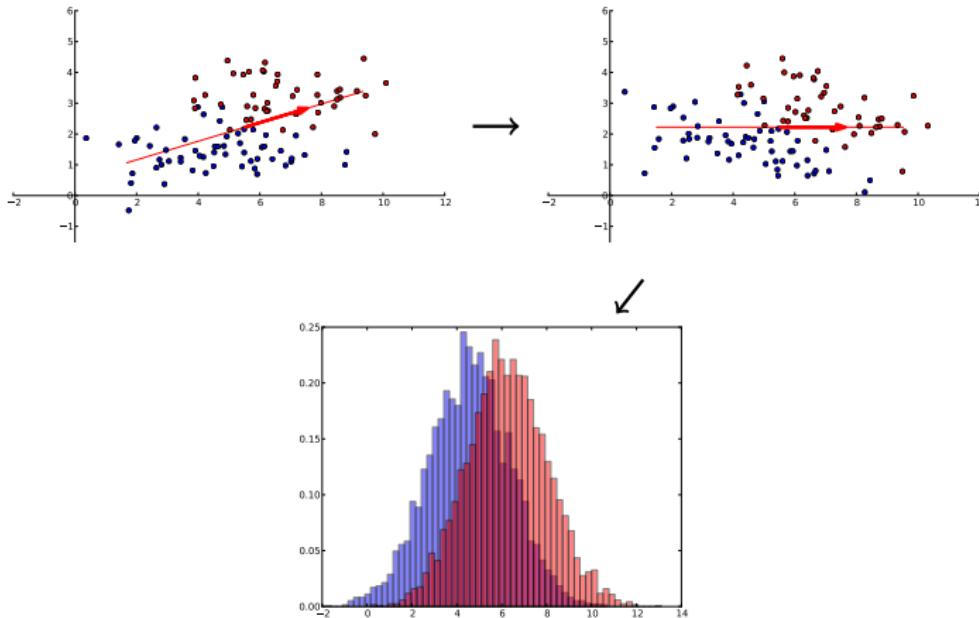
If we remove too much information, classification accuracy drops significantly

For the Euclidean classifier on MNIST, PCA is not helpful in removing unwanted variability

¹ Remember that the optimal dimensionality m should be selected according to results on a validation set!

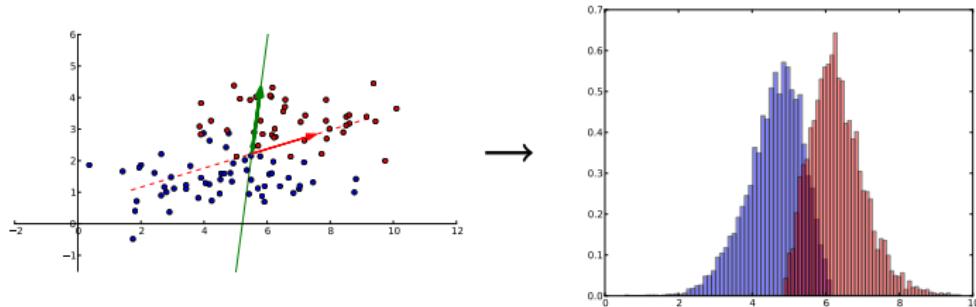
Linear Discriminant Analysis

PCA is unsupervised: no guarantee of obtaining **discriminant** directions



Linear Discriminant Analysis

We want a transformation that allows us to better separate the classes



We represent a direction as a unit vector w

The projected point is $y = w^T x$ (a scalar)

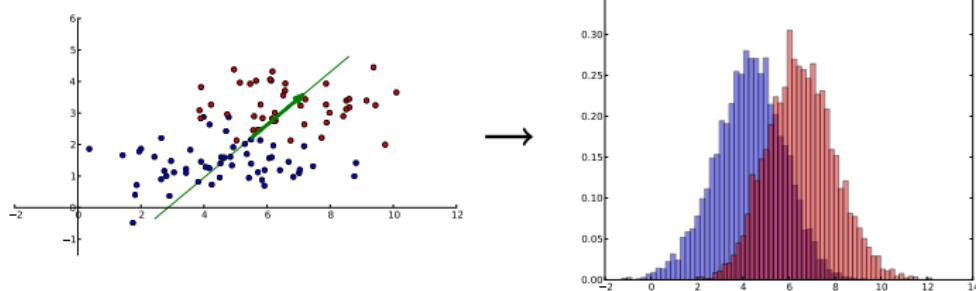
Red line: PCA

Green line: Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis

We could consider the line connecting the class means

$$w \propto \mu_2 - \mu_1$$



Problem: if the data points of each class are scattered along the same directions of the class mean, we still cannot properly separate the classes

Linear Discriminant Analysis

Fisher Linear Discriminant Analysis: find a direction that has a large separation between the classes and small spread inside each class

We measure spread in terms of class **covariance**

LDA: maximize the **between-class** variability over **within-class** variability ratio for the transformed samples:

$$\max_w \frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}}$$

Linear Discriminant Analysis

The between and within class variability matrices are defined as

$$S_B \triangleq \frac{1}{N} \sum_{c=1}^K n_c (\boldsymbol{\mu}_c - \boldsymbol{\mu}) (\boldsymbol{\mu}_c - \boldsymbol{\mu})^T$$

$$S_W \triangleq \frac{1}{N} \sum_{c=1}^K \sum_{i=1}^{n_c} (\mathbf{x}_{c,i} - \boldsymbol{\mu}_c) (\mathbf{x}_{c,i} - \boldsymbol{\mu}_c)^T$$

where $\mathbf{x}_{c,i}$ is the i -th sample of class c , n_c is the number of samples of class c , K is the total number of classes, N is the total number of samples $N = \sum_{c=1}^K n_c$, and

- $\boldsymbol{\mu}$ is the dataset mean $\boldsymbol{\mu} = \frac{1}{N} \sum_{c=1}^K \sum_i \mathbf{x}_{c,i}$
- $\boldsymbol{\mu}_c$ is the mean of class c $\boldsymbol{\mu}_c = \frac{1}{n_c} \sum_{i=1}^{n_c} \mathbf{x}_{c,i}$

Linear Discriminant Analysis

The between class covariance matrix can be interpreted as a covariance matrix for the class means, where each class is weighted by the corresponding sample size n_c

The within class covariance matrix can be seen as a (also weighted) average of the covariance matrix of each class

We can observe that

$$\mathbf{S}_B + \mathbf{S}_W = \frac{1}{N} \sum_{c=1}^K \sum_{i=1}^{n_c} (\mathbf{x}_{c,i} - \boldsymbol{\mu}) (\mathbf{x}_{c,i} - \boldsymbol{\mu})^T$$

i.e., the covariance matrix of the dataset as a whole

Linear Discriminant Analysis

Since we are looking for a discriminant direction w , we now consider the between and within class variance of the projected samples $w^T x$

The global mean and class means in the direction w are simply

$$m = w^T \mu, \quad m_c = w^T \mu_c$$

The between and within class variance over the direction w can also be computed as:

$$s_B = \frac{1}{N} \sum_{c=1}^K n_c (w^T \mu_c - w^T \mu) (w^T \mu_c - w^T \mu)^T = w^T S_B w$$

$$s_W = \frac{1}{N} \sum_{c=1}^K \sum_{i=1}^{n_c} n_c (w^T x_{c,i} - w^T \mu_c) (w^T x_{c,i} - w^T \mu_c)^T = w^T S_W w$$

Linear Discriminant Analysis

Fisher discriminant analysis defines as criterion of optimality the maximization of the **ratio of between and within class variance** for the projected points

We assume that S_W is **full rank**, thus the **objective function** is

$$\mathcal{L}(\mathbf{w}) = \frac{s_B}{s_W} = \frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}}$$

Note that the criterion does not depend on the scale of \mathbf{w} , i.e. if \mathbf{w} is a maximizer of \mathcal{L} , then $\alpha\mathbf{w}$ is also a maximizer of \mathcal{L} . We can therefore select a maximizer with **unit norm**.

Linear Discriminant Analysis

We can find an optimum by solving $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = 0$, where $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$ is the gradient of \mathcal{L} :

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = 2 \frac{\mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}} - 2 \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w} \mathbf{S}_W \mathbf{w}}{(\mathbf{w}^T \mathbf{S}_W \mathbf{w})^2} = 0$$

The optimum is obtained for

$$(\mathbf{w}^T \mathbf{S}_W \mathbf{w}) \mathbf{S}_B \mathbf{w} = (\mathbf{w}^T \mathbf{S}_B \mathbf{w}) \mathbf{S}_W \mathbf{w}$$

i.e.

$$\mathbf{S}_W^{-1} \mathbf{S}_B \mathbf{w} = \lambda(\mathbf{w}) \mathbf{w}$$

where

$$\lambda(\mathbf{w}) = \mathcal{L}(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

Linear Discriminant Analysis

We can observe that

- The optimal solution is an eigenvector of $S_W^{-1}S_B$
- The eigenvalue corresponding to solution w is $\lambda(w) = \mathcal{L}(w)$, i.e. the value of the ratio we want to maximize

The maximum of \mathcal{L} is thus the **eigenvector** of $S_W^{-1}S_B$ corresponding to the **largest eigenvalue**

Linear Discriminant Analysis

The method was originally introduced to solve binary problems

Indeed, once we have estimated w , we can project our test samples over w , and assign the class according to whether the projected value (score) is larger or lower than a given threshold²:

$$C(\mathbf{x}_t) = \begin{cases} C_1 & \text{if } \mathbf{w}^T \mathbf{x}_t \geq t \\ C_2 & \text{if } \mathbf{w}^T \mathbf{x}_t < t \end{cases}$$

²How to select a good threshold will be a topic for next classes

Linear Discriminant Analysis

For the binary problem, we can express the **between-class** covariance matrix as

$$\mathbf{S}_B = k(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T$$

where k is a constant that depends on the number of samples of each class but is irrelevant to find the optimal direction.

In this case, we can observe that $\text{rank}(\mathbf{S}_B) = 1$, thus $\mathbf{S}_W^{-1}\mathbf{S}_B$ has a single non-zero eigenvalue

It can be verified that the eigenvector of $\mathbf{S}_W^{-1}\mathbf{S}_B$ associated to the non-zero eigenvalue is

$$\mathbf{w} \propto \mathbf{S}_W^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)$$

Linear Discriminant Analysis

Despite being originally introduced to solve binary classification problems, LDA has found large success as a dimensionality reduction technique

In this case, we are interested in looking for the m most discriminant directions

We represent these directions as a matrix W , whose columns contain the directions we want to find

Notice that we do not require that W is orthogonal

If we want, we can nevertheless find a basis for the subspace spanned by the columns of W

Linear Discriminant Analysis

The projected points are computed as $\hat{x} = W^T x$

We can express the projected between and within class covariance matrices as

$$\widehat{S}_B = W^T S_B W$$

$$\widehat{S}_W = W^T S_W W$$

Different criteria can be used to generalize the 1-dimensional case

A common one looks for the maximizer of

$$\mathcal{L} = \text{Tr} \left(\widehat{S}_W^{-1} \widehat{S}_B \right)$$

Linear Discriminant Analysis

It can be shown that the solution is given by the m (right) eigenvectors corresponding to the m largest eigenvalues of $S_W^{-1}S_B$

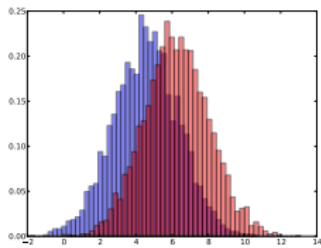
We can also compute the solution by solving the **generalized eigenvalue** problem $S_B\mathbf{w} = \lambda S_W\mathbf{w}$

Notice that, from the definition of S_B , the number of non-zero eigenvalues is at most $C - 1$

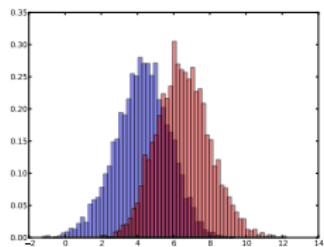
Therefore, LDA allows estimating **at most $C - 1$ directions**

Linear Discriminant Analysis

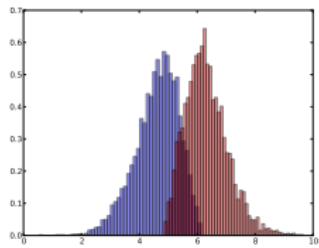
PCA



Eigenvectors of S_B



LDA



Linear Discriminant Analysis

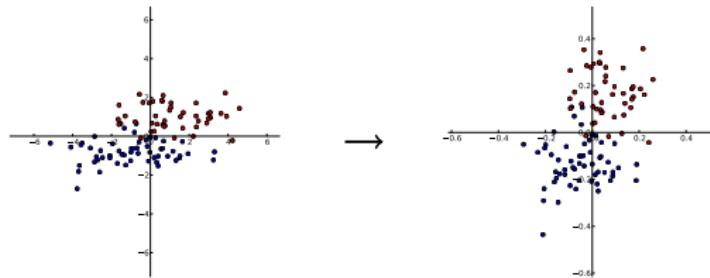
A way to solve the generalized eigenvalue problem, and thus find the LDA matrix W , consists in the joint diagonalization of S_W and S_B that makes S_W become the identity matrix and S_B become a diagonal matrix

Remember that applying the linear transformation $\tilde{x} = Ax$ to our dataset, covariance matrices transform as $\tilde{\Sigma} = A\Sigma A^T$

Linear Discriminant Analysis

Whiten S_W

- Compute the eigen–value decomposition $S_W = U_W \Sigma_W U_W^T$
- Apply the whitening transformation³ described by
$$P_W = U_W \Sigma_W^{-\frac{1}{2}} U_W^T$$
- $S_W \rightarrow I, S_B \rightarrow P S_B P^T$

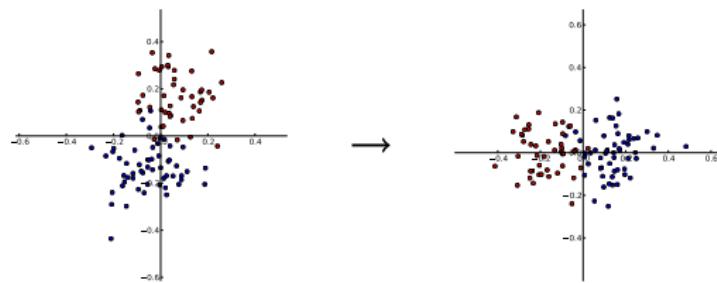


³There are different methods to compute the whitening transformation. Indeed, matrix P is defined up to a unitary transformation.

Linear Discriminant Analysis

Diagonalize the transformed S_B

- Compute the eigenvalue decomposition $\mathbf{P}S_B\mathbf{P}^T = \mathbf{U}_B\Sigma_B\mathbf{U}_B^T$
- Diagonalize by projecting over \mathbf{U}_B^T
- $S_W \rightarrow \mathbf{I}$, $S_B \rightarrow \Sigma_B$



The first m directions of the transformed samples correspond to the LDA subspace. This method will be further discussed in Laboratory 3.

Linear Discriminant Analysis

LDA assumes Gaussian-distributed noise

When the number of directions is large S_W can be singular or close to singular

It is often helpful to pre-process our data using PCA before applying LDA

Linear Discriminant Analysis

In MNIST we have 9 classes, so we can have at most 9 directions

0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9



0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9

PCA+LDA



0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9

PCA

Dimensionality Reduction

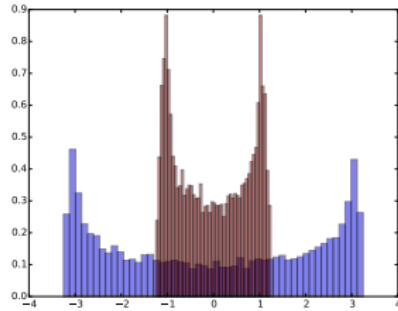
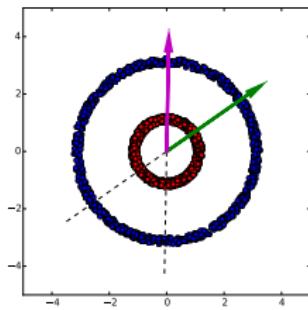
MNIST — Error rates for euclidean distance classifier⁴

m	PCA	PCA+LDA
100	18.1%	—
50	18.2%	—
9	25.5%	12.2%
5	35.9%	17.9%

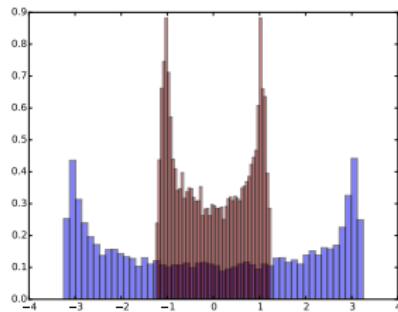
⁴Remember that the method and dimensionality m should be selected according to results on the validation set!

Non-linear Dimensionality Reduction

Linear transformations are not always suited for our data



PCA



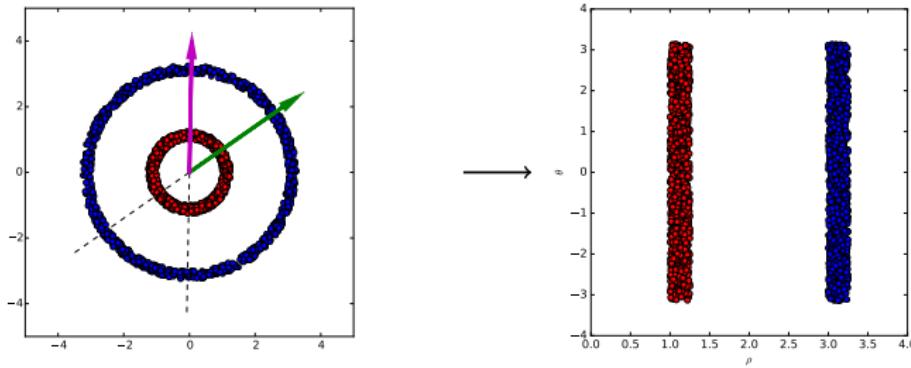
LDA

Non-linear Dimensionality Reduction

We can transform the features so that linear methods are suited for the transformed data

For example, we can represent the 2-dimensional data in the figure through polar coordinates :

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \longrightarrow \mathbf{y} = f(\mathbf{x}) \begin{bmatrix} \rho(\mathbf{x}) \\ \theta(\mathbf{x}) \end{bmatrix}$$



Alternative non-linear methods will be presented in other courses

Probability and density estimation

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Random events and probability

Our goal: making **predictions** that allow us to take **actions**

Complex phenomena: too many factors that can influence the outcomes to account for all of them

Solution: model phenomena in terms of **random events**

- Deterministic event: its occurrence can be predicted exactly
- Random event: its occurrence is uncertain

Random events and probability

We describe random events in terms of their **probability**

- Classical interpretation: probability as fraction of favorable outcomes
- Frequentist interpretation: probability as frequency of an outcome under a large (infinite) number of repeated trials
- Bayesian interpretation: probability as (subjective) measure of belief that an event will occur

The axiomatic treatment defines the quantity that we will use

Interpretations provide guidance on how to employ the axiomatic theory to describe reality

Random events and probability

The next set of slides (5 to 51) recalls basic notions of probability

- Probability spaces
- Random variables (discrete and continuous) and random vectors
- Cumulative distribution functions
- Probability density functions
- Expectations

These are provided as a quick reference. Specific notions will be discussed when needed.

Axioms

Let

- i) Ω be a set (possible outcomes)
- ii) \mathcal{A} be σ -field over Ω
 - a) \mathcal{A} is a collection of subsets of Ω (events)
 - b) $\Omega \in \mathcal{A}$
 - c) If $A \in \mathcal{A}$ then $A^C \in \mathcal{A}$
 - d) If $A_1, A_2, \dots \in \mathcal{A}$ then $\bigcup_{n=1}^{\infty} A_n \in \mathcal{A}$

Note that b), c) and d) imply that

- e) $\emptyset \in \mathcal{A}$
- f) $\bigcap_{n=1}^{\infty} A_n \in \mathcal{A}$

Axioms

We define probability as a function P

$$P : \mathcal{A} \longrightarrow \mathbb{R}^+$$

which has the properties:

- 1) $P(\Omega) = 1$, and
- 2) (*countable additivity*) Given a sequence A_1, A_2, \dots of mutually exclusive elements of \mathcal{A} (i.e., $i \neq j \implies A_i \cap A_j = \emptyset$),

$$P\left(\bigcup_{n=1}^{\infty} A_n\right) = \sum_{i=n}^{\infty} P(A_n)$$

The triplet (Ω, \mathcal{A}, P) is called a **probability space**

Properties

Some properties of probability spaces:

- $P(\emptyset) = 0$
- $A \subset B \implies P(A) \leq P(B)$
- $P(A) \leq 1$
- $P(A^C) = 1 - P(A)$
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- $P(\bigcup_n A_n) = 1 - P(\bigcap_n A_n^C)$
- If $A_1 \subset A_2 \subset A_3 \subset \dots$ is an increasing sequence of sets with $A = \bigcup_n^\infty A_n$ (with $A = \bigcap_n^\infty A_n$). Then $P(A) = \lim_{n \rightarrow \infty} P(A_n)$ (the same applies for a decreasing sequence of sets, with $A = \bigcap_n A_n$)

An example

Consider rolling a six-sided die

The set of possible outcomes is the set of possible values that we can roll

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

Events can consist of single outcomes, e.g. “rolling a 6”

- $A = \{6\}$

Events can consider more outcomes, e.g. “rolling an even number”

- $A = \{2, 4, 6\}$

An example

We can assign a probability to the different events corresponding to the outcomes

If we assume that these events have the same probability p :

$$P(\{1\}) = P(\{2\}) = P(\{3\}) = P(\{4\}) = P(\{5\}) = P(\{6\}) = p$$

Since the events are mutually exclusives and their union is Ω , it follows that the value of p should be $p = \frac{1}{6}$

We can then compute the probability of rolling an even number:

$$P(\{2, 4, 6\}) = P(\{2\}) + P(\{4\}) + P(\{6\}) = \frac{1}{2}$$

Conditional probability

Let $A, B \in \mathcal{A}$, with $P(B) > 0$

We define the **conditional probability** of A given B as

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

The conditioned probability represents the probability of A , when we know that B has happened

Note: $A|B$ is not an event per se, we are rather defining a new probability measure $P(\cdot|B)$

Conditional probability

Consider again rolling a six-sided die

We don't know the result, but somebody tells us that it's not 1

We can compute the probability of rolling a number lower than 4 ($A = \{1, 2, 3\}$) conditioned on the roll being larger than one ($B = \{2, 3, 4, 5, 6\}$):

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(\{2, 3\})}{P(\{2, 3, 4, 5, 6\})} = \frac{2}{5}$$

Conditional probability

Bayes formula allows expressing the conditional probability $P(B|A)$

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

We say that two events are independent if and only if

$$P(A \cap B) = P(A)P(B)$$

In the following we will often denote the *joint* probability of two events $P(A \cap B)$ simply as $P(A, B)$

Conditional probability

Considering again the previous example, we can compute the probability that the roll is greater than one (event B) given that the roll is lower than 4 (event A):

$$P(A|B) = \frac{2}{5}, \quad P(A) = \frac{1}{2}, \quad P(B) = \frac{5}{6}$$

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} = \frac{2}{3}$$

We can also verify that

$$P(B|A) = \frac{P(A \cap B)}{P(A)} = \frac{P(\{2, 3\})}{P(\{1, 2, 3\})} = \frac{2}{3}$$

Random variables

Random variables allow extending probabilistic reasoning to quantities that depend on events

A random variable X is defined as a function

$$X : \Omega \longrightarrow \mathbb{R}$$

such that, for any real value $x \in \mathbb{R}$ the event

$$\{\omega \in \Omega : X(\omega) \leq x\}$$

belongs to \mathcal{A} .

Essentially X is a function of the outcomes ω , for which we can compute the probability that it takes values no greater than x :

$$P(X \leq x) = P(\{\omega \in \Omega : X(\omega) \leq x\})$$

Random variables

Consider rolling two six-sided dices

The outcome space Ω consists of pairs of values

$$\Omega = \{(1, 1), (1, 2), \dots, (6, 6)\}$$

We are interested in computing the probability that the sum of the two dice is lower or equal to some value

Let an outcome be denoted as $\omega = (r_1, r_2)$. We define the R.V. X that maps each outcome to the sum of its components:

$$X(\omega) = r_1 + r_2$$

Random variables

We can then compute the probability that $X \leq x$:

$$P(X \leq x) = P(\{(r_1, r_2) \in \Omega : r_1 + r_2 \leq x\})$$

For example, if we set $x = 4$, then we have

$$\begin{aligned} P(X \leq 4) &= P(\{(r_1, r_2) \in \Omega : r_1 + r_2 \leq 4\}) \\ &= P(\{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (3, 1)\}) = \frac{1}{6} \end{aligned}$$

Random variables

A R.V. induces a probability space over the real line

In general, we will be interested in assigning probabilities to subset of the real line A in the form $P(X \in A)$

Note that not all subsets A of the real line may be assigned a probability, as the set $\{\omega \in \Omega : X(\omega) \in A\}$ may not be an event

The subset of the real line for which we can compute the probability $P(X \in A)$ is a *Borel* σ -field \mathcal{B}

The probability space induced by X is $(\mathbb{R}, \mathcal{B}, P)$

Note: the same symbol P is used for both for the original and the induced probability space, its meaning will be clear from the context

Random variables

For example, the definition of a R.V. implies that $\{\omega : X(\omega) \leq b\}$ is an event

It follows that

$$\{\omega : X(\omega) > a\}$$

is also an event. It follows that

$$\{\omega : a < X(\omega) \leq b\} = \{\omega : X(\omega) > a\} \cap \{\omega : X(\omega) \leq b\}$$

and

$$\{\omega : X(\omega) = x\} = \bigcup_n \{\omega : x - \frac{1}{n} < X(\omega) \leq x\}$$

are also an event

We can thus compute the probability that $X \in (a, b]$, i.e. $P(a < X \leq b)$, or the probability that $X = x$, i.e. $P(X = x)$

Random variables

Given R.V. X , we define its **cumulative distribution function** (c.d.f.) as:

$$F_X(x) = P(X \leq x)$$

The c.d.f. has the following properties:

- 1) $0 \leq F_X(x) \leq 1 \quad \forall x \in \mathbb{R}$
- 2) F_X is non-decreasing, i.e. $x_1 < x_2 \implies F_X(x_1) \leq F_X(x_2)$
- 3) $\lim_{x \rightarrow -\infty} F_X(x) = 0, \lim_{x \rightarrow \infty} F_X(x) = 1$
- 4) F_X is right-continuous: $\lim_{x \downarrow x_0} F_X(x) = F_X(x_0)$

It follows that

- i) $P(a < X \leq b) = F_X(b) - F_X(a)$
- ii) $P(X = x_0) = F_X(x_0) - \lim_{x \uparrow x_0} F_X(x)$

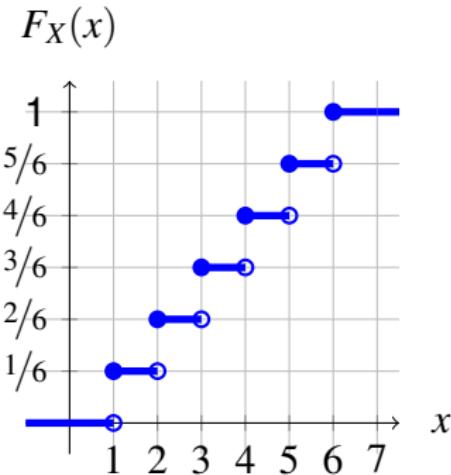
Random variables

Consider again rolling a six-sided die

Consider the R.V. that gives the rolled value $X(\omega) = \omega$

The c.d.f. is given by

$$F_X(x) = \begin{cases} 0 & \text{if } x < 1 \\ 1/6 & \text{if } 1 \leq x < 2 \\ 2/6 & \text{if } 2 \leq x < 3 \\ 3/6 & \text{if } 3 \leq x < 4 \\ 4/6 & \text{if } 4 \leq x < 5 \\ 5/6 & \text{if } 5 \leq x < 6 \\ 1 & \text{if } x \geq 6 \end{cases} \quad (1)$$



Discrete random variables

A R.V. is said to be **discrete** if it takes a **finite** or a **countably infinite** number of values

For a discrete R.V. X we can define the **probability mass function** (p.m.f.) or discrete **density**:

$$f_X(x) = P(X = x)$$

Discrete random variables

The discrete density has the following properties:

- 1) $f_X(x) \geq 0$
- 2) $f_X(x) = 0$ for all x except at most a countably infinite number of values (those taken by X)
- 3) $\sum_{x \in \mathcal{S}} f_X(x) = 1$

where \mathcal{S} is the **support** of X , i.e. the set of values for which $f_X(x) > 0$

If we know the density, we can compute the c.d.f.

$$F_X(x) = \sum_{t \leq x} f_X(t)$$

Discrete random variables

The examples that we have considered so far involve discrete R.V. with a finite support

- Rolled value for a six-sided die:

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

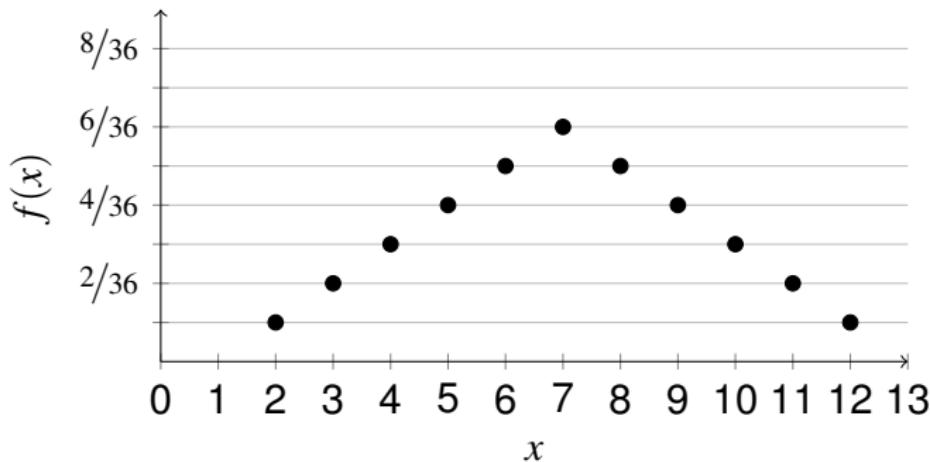
$$f_X(x) = \frac{1}{6} \quad \forall x \in \mathcal{S}$$

Discrete random variables

The examples that we have considered so far involve discrete R.V. with a finite support

- Sum of the values rolled by two dice:

$$\mathcal{S} = 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$$



Discrete random variables

We consider now an example where the support is not finite

We start considering the problem of tossing a coin K times

For example, with $K = 3$ the outcomes are the possible sequences

$$\Omega = (HHH), (HHT), (HTH), (HTT), (THH), (THT), (TTH), (TTT)$$

In general, if the coin is fair, we can assign to each sequence the same probability $p = \frac{1}{2^K}$

Discrete random variables

We now consider the problem of tossing a coin until it comes up head

The outcome consists of a countable number of elements (sequences of tails followed by one head)

$$\Omega = \{(H), (TH), (TTH), (TTTH), \dots\}$$

If the coin is fair, it's reasonable to assume that the events "the first toss is a head", $A = \{(H)\}$, and its complement "the first toss is a tail" have both probability $P(A) = P(A^C) = 1/2$

Assuming tosses are independent, we can now consider the events "the second toss is a head", $B = \{(TH)\}$, and its complement "the second toss is a tail" (both imply that the first toss was tail as well)

Discrete random variables

We note that $B \cup B^C = A^C$, so that $P(B \cup B^C) = 1/2$

Again, if the coin is fair, it's reasonable to assume that $P(B) = P(B^C) = 1/4$

By induction, we can assign to any event ω a probability $P(\{\omega\}) = 1/2^K$, where K is the length of the sequence ω .

In general, if the coin is biased and the probability of head is p , then

$$P(\{\omega\}) = p(1-p)^{K-1}$$

Discrete random variables

We now consider the R.V. X that maps an outcome to the number of *tails* required to get a head (note that this is the sequence length *minus 1*)

$$X((H)) = 0, \quad X((TH)) = 1, \quad X((TTH)) = 2, \dots$$

The support of X is the set of positive integers \mathbb{N}^+

The density (or distribution) of X is the **geometric distribution**:

$$f_X(x) = \begin{cases} p(1-p)^x & x \in \mathbb{N}^+ \\ 0 & \text{otherwise} \end{cases}$$

Continuous random variables

In many cases, a R.V. can take any value in \mathbb{R} (or in a subset of \mathbb{R})

Nevertheless, we have shown that also in these cases $\{X \leq x\}$ are events, and we can define the c.d.f. of X as $F_X(x) = P(X \leq x)$

A R.V. X for which F_X is a continuous function will be referred to as **continuous** Random Variable

If F_X is continuous, then $\lim_{x \uparrow x_0} F_X(x) = \lim_{x \downarrow x_0} F_X(x)$, thus

$$P(X = x) = 0$$

It follows that

$$P(a < X \leq b) = P(a \leq X \leq b) = P(a < X < b) = P(a \leq X < b)$$

Continuous random variables

We have seen that for discrete R.V.s we can express the cumulative function in terms of probability mass functions

Similarly, for a continuous R.V. we introduce the concept of **probability density**

We say that a function $f : \mathbb{R} \rightarrow \mathbb{R}$ is a density if and only if

1) $f(x) \geq 0 \quad \forall x \in \mathbb{R}$

2) f is integrable over \mathbb{R}

3) $\int_{-\infty}^{+\infty} f(x)dx = 1$

Continuous random variables

Let F_X be the c.d.f. of continuous R.V. X , and let there exist a density $f_X(x)$ such that

$$F_X(x) = \int_{-\infty}^x f_X(t)dt$$

We will say that f_X is a **probability density function** (p.d.f.) of X

Note that f_X is not unique (e.g. if f_X and g_X differ on a non-measurable subset of \mathbb{R} , they are both densities for X)

If F_X is differentiable, then

$$f_X(x) = \frac{d}{dx} F_X(x)$$

is a p.d.f. of X

Continuous random variables

In contrast with the density of discrete R.V.s, the p.d.f. of a continuous R.V. can, in general, take values greater than 1

The p.d.f. allows computing the probability that X is in an interval $[a, b]$:

$$P(a \leq X \leq b) = \int_a^b f_X(x)dx$$

More in general, if $X \in A$ is an event, we can compute its probability as

$$P(X \in A) = \int_A f_X(x)dx$$

Random vectors

We define a m -dimensional **Random Vector** as a vector whose components $X_1 \dots X_m$ of X are Random Variables

Let $\mathbf{x} = (x_1, \dots, x_m)$.

The cumulative distribution function of X is defined as:

$$F_X(\mathbf{x}) = P(X_1 \leq x_1, \dots, X_m \leq x_m)$$

i.e., the **joint** cumulative distribution for $X_1 \dots X_m$

The distributions of each component (or subset of components) are called **marginal** distributions

Random vectors

Discrete R.V.:

$$\begin{aligned}f_X(\mathbf{x}) &= P(X = \mathbf{x}) \\&= P(X_1 = x_1, \dots, X_m = x_m)\end{aligned}$$

$$F_X(\mathbf{x}) = \sum_{y_1 \leq x_1} \cdots \sum_{y_m \leq x_m} f_X(\mathbf{y})$$

Continuous R.V.:

$$\left| \begin{aligned}f_X(\mathbf{x}) &= \frac{\partial^m}{\partial x_1 \cdots \partial x_m} F_X(\mathbf{x}) \\F_X(\mathbf{x}) &= \int_{-\infty}^{x_1} \cdots \int_{-\infty}^{x_m} f_X(\mathbf{y}) dy_1 \cdots dy_m\end{aligned}\right.$$

Marginals

$$\left| \begin{aligned}f_{X_1}(x_1) &= \sum_{y_2 \in \mathcal{S}(X_2)} \cdots \sum_{y_m \in \mathcal{S}(X_m)} f_X(x_1, y_2, \dots, y_m) \\f_{X_1}(x_1) &= \int_{\mathbb{R}} \cdots \int_{\mathbb{R}} f_X(x_1, y_2, \dots, y_m) dy_2 \cdots dy_m\end{aligned}\right.$$

Statistical independence: $X \perp\!\!\!\perp Y \iff F_{X,Y}(x,y) = F_X(x)F_Y(y)$

$$f_{X,Y}(x,y) = f_X(x)f_Y(y) \iff X \perp\!\!\!\perp Y \quad \left| \quad f_{X,Y}(x,y) = f_X(x)f_Y(y) \implies X \perp\!\!\!\perp Y\right.$$

Random variables

Given two R.V.s X and Y , we define the **conditional density** of X given $Y = y$:

$$f_{X|Y}(x|y) = \frac{f_{X,Y}(x,y)}{f_Y(y)}, \quad \forall y \in \mathcal{S}(Y)$$

where $\mathcal{S}(Y)$ is the support of Y

Bayes rule:

$$f_{Y|X}(y|x) = \frac{f_{X|Y}(x|y)f_Y(y)}{f_X(x)}$$

Also,

$$f_{Y|X}(y|x) = \frac{f_{X,Y}(x,y)}{\int f_{X,Y}(x,y)dy} = \frac{f_{X|Y}(x|y)f_Y(y)}{\int f_{X|Y}(x|y)f_Y(y)dy}$$

Coinditional independence: X and Y are independent given Z ,

$$(X \perp\!\!\!\perp Y)|Z \iff F_{X,Y|Z}(x,y|z) = F_{X|Z}(x|z)F_{Y|Z}(y|z)$$

Transformations of random variables

In many cases we are interested in the distribution of *functions* of our R.V.s

For example, we may want to know the distribution of $Y = X^2$, or, more in general, of $Y = g(X)$

For discrete R.V.s we can simply compute

$$f_Y(y) = \sum_{x|f(x)=y} f_X(x)$$

Transformations of random variables

For continuous R.V.s, we can compute the c.d.f. of Y as

$$F_Y(y) = P(g(X) \leq y) = P(X \in \{x | g(x) \leq y\})$$

If g is monotonic, differentiable, and with differentiable inverse, we can write

$$F_Y(y) = P(g(X) \leq y) = P(X \leq g^{-1}(y)) = F_X(g^{-1}(y))$$

Taking the derivative w.r.t. y :

$$\begin{aligned}f_Y(y) &= \frac{d}{dy} F_Y(y) = \frac{d}{dy} F_X(g^{-1}(y)) = \frac{d}{dg^{-1}(y)} F_X(g^{-1}(y)) \cdot \frac{d}{dy} g^{-1}(y) \\&= f_X(g^{-1}(y)) \cdot \frac{d}{dy} g^{-1}(y)\end{aligned}$$

Transformations of random variables

In general, let $g : \mathcal{S}_X \longrightarrow \mathcal{S}_Y$ be invertible and differentiable, with differentiable inverse. Then

$$f_Y(y) = f_X(g^{-1}(y)) \left| \frac{d}{dy} g^{-1}(y) \right|$$

For example, let X be a R.V. with support \mathbb{R}^+ , and $Y = X^3$. We have

$$g(x) = x^3, \quad g^{-1}(y) = \sqrt[3]{y} \quad \frac{d}{dy} g^{-1}(y) = \frac{1}{3} y^{-2/3}$$

The density of Y is thus

$$f_Y(y) = \frac{1}{3} f_X(\sqrt[3]{y}) y^{-2/3}$$

Transformations of random variables

We can extend the change of variables rule to continuous Random Vectors

Let $\mathbf{Y} = g(\mathbf{X})$, invertible, differentiable and with differentiable inverse

Then

$$f_{\mathbf{Y}}(\mathbf{y}) = f_{\mathbf{X}}(g^{-1}(\mathbf{y})) |\det \mathbf{D}g^{-1}(\mathbf{y})|$$

$\mathbf{D}g^{-1}(\mathbf{y})$ is the **Jacobian** matrix of g^{-1} (i.e. matrix of partial derivatives $\frac{\partial}{\partial y_j} [g^{-1}]_i(\mathbf{y})$)

Transformations of random variables

In some cases we will consider the sum or R.V.s $Z = X + Y$

In this case, the c.d.f. is given by

$$F_Z(z) = P(Z \leq z) = P(X + Y \leq z) = \int_{\{(x,y)|x+y \leq z\}} f_{X,Y}(x,y) dx dy$$

and corresponds to the density

$$f_Z(z) = \int f_{X,Y}(x, z-x) dx = \int f_{X,Y}(z-y, y) dy$$

Transformations of random variables

The same result can be obtained from the conditional density of $Z|Y$:

$$(Z|Y = y) = (X|Y = y) + y$$

Applying the change of variable

$$f_{Z|Y}(z|y) = f_{X|Y}(z - y|y)$$

and finally

$$f_Z(z) = \int f_{Z|Y}(z|y)f_Y(y)dy = \int f_{X|Y}(z-y|y)f_Y(y)dy = \int f_{X,Y}(z-y, y)dy$$

Expectations

We define the **mean** or **expected value** of a R.V. as:¹

- Discrete R.V.:

$$\mathbb{E}_X[X] = \sum_{x \in \mathcal{S}} x f_X(x)$$

- Continuous R.V.:

$$\mathbb{E}_X[X] = \int_{\mathcal{S}} x f_X(x) dx$$

¹Here and in the following definitions we assume that the integrals exist and are finite, otherwise the corresponding quantities are undefined. We will discuss the cases where this happens when needed.

Expectations

The **variance** of a R.V. is given by

$$\text{var}(X) = \mathbb{E}_X \left[(X - \mathbb{E}_X [X])^2 \right] = \mathbb{E}_X [X^2] - \mathbb{E}_X [X]^2$$

where $\mathbb{E}_X [X^2]$ is given by:

- Discrete R.V.:

$$\mathbb{E}_X [X^2] = \sum_{x \in \mathcal{S}} x^2 f_X(x)$$

- Continuous R.V.:

$$\mathbb{E}_X [X^2] = \int_{\mathcal{S}} x^2 f_X(x) dx$$

The variance is a measure of “spread” of the distribution around the mean

We define the **standard deviation** as $\text{std}(X) = \sqrt{\text{var}(X)}$

Expectations

We define the *k-th moment* of X as $\mathbb{E}_X [X^k]$

- Discrete R.V.:

$$\mathbb{E}_X [X^k] = \sum_{x \in \mathcal{S}} x^k f_X(x)$$

- Continuous R.V.:

$$\mathbb{E}_X [X^k] = \int_{\mathcal{S}} x^k f_X(x) dx$$

The *central k-th moment* is defined as $\mathbb{E}_X [(X - \mathbb{E}_X [X])^k]$.

Expectations

In general, we define the **expectation** of function $g(X)$ as:

- Discrete R.V.:

$$\mathbb{E}_X [g(X)] = \sum_{x \in \mathcal{S}} g(x) f_X(x)$$

- Continuous R.V.:

$$\mathbb{E}_X [g(X)] = \int_{\mathcal{S}} g(x) f_X(x) dx$$

Expectations

Since summation and integration are linear operators, also the expectation operator $\mathbb{E} [\cdot]$ is linear:

$$\mathbb{E}_X [aX] = a\mathbb{E}_X [X]$$

$$\mathbb{E}_{X,Y} [X + Y] = \mathbb{E}_X [X] + \mathbb{E}_Y [Y]$$

For example, let $\mathbf{Y} = \mathbf{AX} + \mathbf{b}$

Then

$$\mathbb{E}_Y [\mathbf{Y}] = \mathbb{E}_X [\mathbf{AX} + \mathbf{b}] = \mathbf{A}\mathbb{E}_X [\mathbf{X}] + \mathbf{b}$$

In general, we will write just \mathbb{E} , unless the considered distribution is not clear.

Expectations

Let X, Y be two R.V.s

We define the **covariance** of X and Y as

$$\text{cov}(X, Y) = \mathbb{E}_{X,Y} [XY] - \mathbb{E}_X [X]\mathbb{E}_Y [Y] = \text{cov}(Y, X)$$

If $\text{cov}(X, Y) = 0$ we say that X and Y are uncorrelated

$X \perp\!\!\!\perp Y \implies \text{cov}(X, Y) = 0$ (but the opposite is not true in general!)

Expectations

We also define the Pearson **correlation coefficient** between X and Y as

$$\text{corr}(X, Y) = \frac{\text{cov}(X, Y)}{\sqrt{\text{var}(X) \text{ var}(Y)}}$$

We have $-1 \leq \text{corr}(X, Y) \leq 1$

Also, $\text{corr}(X, Y) = 1 \iff Y = aX + b$ for some $a \neq 0, b \in \mathbb{R}$

Expectations

For a Random Vector we define the **covariance matrix**

$$\begin{aligned}\boldsymbol{\Sigma} = \text{cov}(\boldsymbol{X}) &= \mathbb{E} \left[(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])^T \right] \\ &= \begin{bmatrix} \text{var}(X_1) & \text{cov}(X_1, X_2) & \cdots & \text{cov}(X_1, X_m) \\ \text{cov}(X_1, X_2) & \text{var}(X_2) & \cdots & \text{cov}(X_2, X_m) \\ \vdots & \vdots & \ddots & \vdots \\ \text{cov}(X_m, X_1) & \text{cov}(X_m, X_2) & \cdots & \text{var}(X_m) \end{bmatrix}\end{aligned}$$

Expectations

As for the variance, we can express the covariance matrix as

$$\Sigma = \text{cov}(X) = \mathbb{E} [XX^T] - \mathbb{E}[X]\mathbb{E}[X]^T$$

Σ is symmetric

Σ is positive semi-definite: all eigenvalues are $\lambda_i \geq 0$, and $\forall v \in \mathbb{R}^m, v^T \Sigma v \geq 0$

If $Y = AX + b$, then

$$\text{cov}(Y) = A \text{cov}(X)A^T$$

For 1-dimensional R.V. the covariance matrix becomes a scalar, and corresponds to the variance

Notation

In the following we will denote the density of a continuous R.V. as $f_X(x)$

For discrete R.V.s, we will also use $P_X(x)$ or $P(X = x)$

For conditional continuous p.d.f., we will use $f_{X|Y}(x|y)$, $f_{X|Y=y}(x)$, or $f_{X|y}(x)$ if there is on ambiguity on the conditioning R.V.

When densities depend on unknown values y , we use the notation $f_{X|y}(x|y)$ or $P_X(x|y)$ for those cases where y is not treated as a random value (note that there is no R.V. Y involved in the notation)

For conditional discrete densities, we may alternatively use $P_{X|Y}(x|y)$, $P_{X|Y=y}(x)$, $P(X = x|Y = y)$, or, if there is on ambiguity on the conditioning R.V., simply $P_{X|y}(x)$ and $P(X = x|y)$

When referring to known distributions, e.g. the normal distribution \mathcal{N} , we will use the same symbol both for the distribution and its density: $\mathcal{N}(\mu, \sigma^2)$ is a normal distribution, with density given by $\mathcal{N}(x|\mu, \sigma^2)$

Discrete R.V.: Bernoulli distribution

The **Bernoulli** distribution can be used to model the outcome of a binary event, e.g. tossing a single (potentially biased) coin that can result in a (H)ead or a (T)ail

Let $X \in \{0, 1\}$ denote the *R.V.* that assigns value 1 (also called **success**) to head

The distribution of X is

$$X \sim \text{Ber}(p)$$

$$\begin{aligned} P_X(x) = \text{Ber}(x|p) &= \begin{cases} p & \text{if } x = 1 \\ 1 - p & \text{if } x = 0 \end{cases} \\ &= p^x(1 - p)^{1-x} \end{aligned}$$

In many cases we will simply write that $P(H) = p$, implicitly assuming that symbol H has been mapped to 1

Discrete R.V.: Binomial distribution

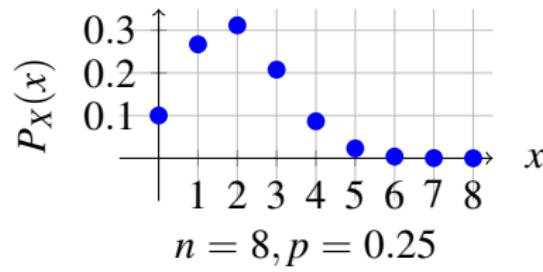
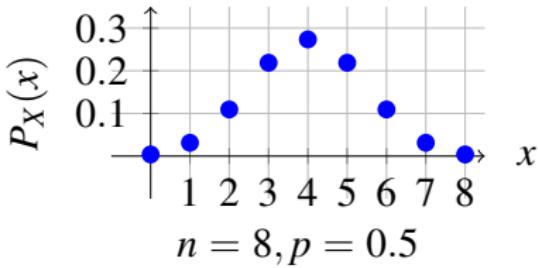
Suppose we want to count the number of successes in n repeated trials (e.g. the number of heads over n coin flips)

Let p denote the probability of success for a single trial

The distribution of X is the **Binomial** distribution

$$X \sim \text{Bin}(n, p)$$

$$P_X(x) = \binom{n}{x} p^x (1-p)^{n-x}$$



Discrete R.V.: Binomial distribution

- $\text{Ber}(p) \sim \text{Bin}(1, p)$
- $X_1 \perp\!\!\!\perp \dots \perp\!\!\!\perp X_n \sim \text{Ber}(p) \implies Y = \sum_i X_i \sim \text{Bin}(n, p)$
- $X \sim \text{Ber}(p) \implies \mathbb{E}[X] = p, \text{var}(X) = p(1 - p)$
- $X \sim \text{Bin}(n, p) \implies \mathbb{E}[X] = np, \text{var}(X) = np(1 - p)$

Discrete R.V.: Categorical and multinomial distribution

The Bernoulli and Binomial distributions can be extended to events that have K possible outcomes (e.g. rolling a die)

Categorical distribution: $X \in \{1, 2, \dots, K\}^2$

$$X \sim \text{Cat}(\mathbf{p})$$

$$f_X(x) = P(X = x) = p_x = \prod_i p_i^{\mathbb{I}[x=i]}$$

where $\mathbf{p} = (p_1, \dots, p_K)$, with $\sum_{i=1}^K p_i = 1$. p_i is the probability of outcome i , and \mathbb{I} is the indicator function

$$\mathbb{I}[C] = \begin{cases} 1 & \text{if } C \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

²The actual labels are in many practical cases irrelevant. For example, if we want to use the same encoding as for the Bernoulli distribution, we can assume $X \in \{0, \dots, K-1\}$

Discrete R.V.: Categorical and multinomial distribution

In many cases it's convenient to represent outcomes with a 1-of-K encoding (vector):

$$X = 1 \implies \mathbf{X} = (1, 0, \dots, 0)$$

$$X = 2 \implies \mathbf{X} = (0, 1, \dots, 0)$$

...

$$X = K \implies \mathbf{X} = (0, 0, \dots, 1)$$

The density can then be expressed as

$$f_{\mathbf{X}}(\mathbf{x}) = \prod_i p_i^{x_i}$$

In the following we will adopt this approach

Discrete R.V.: Categorical and multinomial distribution

We can consider a set of n trials, encoded as $\mathbf{x} = (x_1 \dots x_K)$, where x_i denotes the number of occurrences of outcome i , and $n = \sum_{i=1}^m x_i$.

Let $\mathbf{p} = (p_1, \dots, p_K)$ be the vector of probabilities for a single trial
 X follows a **Multinomial** distribution

$$\mathbf{X} \sim \text{Mul}(n, \mathbf{p})$$

$$f_X(\mathbf{x}) = \frac{n!}{x_1! \cdots x_K!} \prod_{i=1}^K p_i^{x_i}$$

- $\text{Mul}(1, \mathbf{p}) \sim \text{Cat}(\mathbf{p})$
- $X_1 \perp\!\!\!\perp \dots \perp\!\!\!\perp X_n \sim \text{Cat}(\mathbf{p}) \implies \mathbf{Y} = \sum_{i=1}^n X_i \sim \text{Mul}(n, \mathbf{p})$

Continuous R.V.: Gaussian distribution

The **Gaussian** or **normal** distribution is probably the most employed example of continuous distributions

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

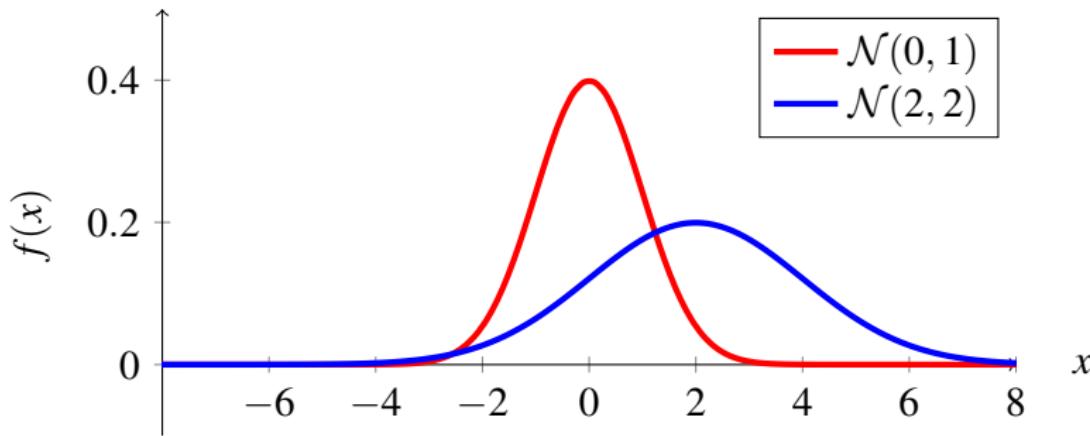
- μ is the mean, and as the name suggests $\mathbb{E}[X] = \mu$
- σ is called standard deviation, and $\text{var}(X) = \sigma^2$
- The inverse of the variance $\lambda = \frac{1}{\sigma^2}$ is called **precision**

if $X \sim \mathcal{N}(0, 1)$ we say that X follows a **standard normal distribution**

Continuous R.V.: Gaussian distribution

The distribution is symmetric, centered around μ

Higher variance corresponds to a “flatter” density



Continuous R.V.: Gaussian distribution

Theorem (Central limit)

Let X_1, \dots, X_N be a sequence of independent and identically distributed (i.i.d.) random variables, with mean μ and variance $\sigma^2 > 0$. Let $S_N = \sum_{i=1}^N X_i$. Then

$$\frac{S_N - N\mu}{\sigma\sqrt{N}}$$

converges in distribution to $\mathcal{N}(0, 1)$

Convergence in distribution means that

$$\lim_{n \rightarrow \infty} F_{S_N}(x) = F_{\mathcal{N}(0,1)}(x)$$

for each x

Multivariate Gaussian Distribution

We can extend the Gaussian distribution to random vectors

Let X be a random vector $X = [X_1, \dots, X_N]^T$ where X_i are i.i.d., standard normal distributed R.V.s $X_i \sim \mathcal{N}(0, 1)$

The distribution of X is given by the joint distribution of X_1, \dots, X_N

$$f_X(\mathbf{x}) = \prod_{i=1}^N f_{X_i}(x_i)$$

or, equivalently,

$$f_X(\mathbf{x}) = (2\pi)^{-\frac{N}{2}} e^{-\frac{1}{2}\mathbf{x}^T \mathbf{x}}$$

We say that X follows a **standard multivariate normal** distribution

$$\mathbf{X} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

Multivariate Gaussian Distribution

X follows a **multivariate Gaussian** (MVG) distribution with mean μ and covariance matrix Σ if X can be written as a linear transformation of a standard multivariate normal distributed random vector Y :

$$X = AY + \mu$$

where $Y \sim \mathcal{N}(\mathbf{0}, I)$ and $\Sigma = AA^T$

We will write $X \sim \mathcal{N}(\mu, \Sigma)$

The p.d.f. of X is given by

$$f_X(x) = (2\pi)^{-\frac{N}{2}} |\Sigma|^{-\frac{1}{2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

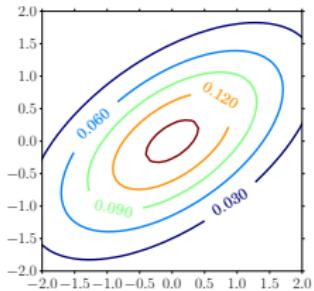
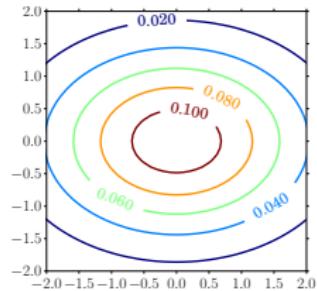
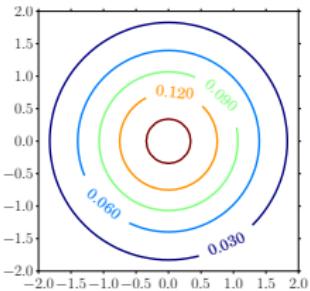
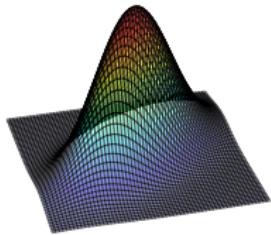
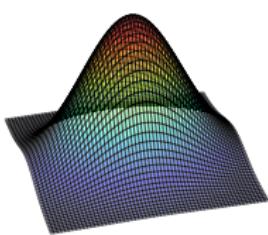
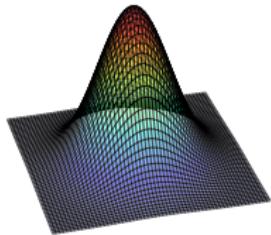
Often, rather than working with a covariance matrix, it's easier to work with its inverse, the precision matrix $\Lambda = \Sigma^{-1}$

Multivariate Gaussian Distribution

$$\mu = \mathbf{0}, \Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mu = \mathbf{0}, \Sigma = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mu = \mathbf{0}, \Sigma = \begin{bmatrix} 1.5 & 0.7 \\ 0.7 & 1 \end{bmatrix}$$



Density estimation

We start considering a simple example

We would like to predict whether a coin flip will result in a head (H) or tail (T)

We do not know whether the coin is biased or not

However, we have observed a number of tosses n

Density estimation

Let's denote the observed results as $(x_1 \dots x_n)$

For a head toss, we have $x_i = 1$, while we represent a tail as $x_i = 0$

These can be considered outcomes of R.V.s $(X_1 \dots X_n)$

Our goal is to predict the probability that a new toss X_t will result in head

We want to model the distribution $X_t | X_1 = x_1 \dots X_n = x_n$

Density estimation

Let's, for a moment, assume that we knew the probability that the coin lands a head, $P(H) = \pi$

We also assume that the coin tosses are independent and identically distributed

Thus we can model the R.V.s X_i as Bernoulli R.V.s with parameter π :

$$X_i \sim X \sim \text{Ber}(\pi)$$

Since we know the probability of head π , the R.V. X_t is independent of $X_1 \dots X_n$ and is also Bernoulli distributed:

$$X_t \sim X \sim \text{Ber}(\pi)$$

Density estimation

Unfortunately, we do not know π

The first approach that we follow, the frequentist approach, assumes that there exists a “true” value π_T that explains the observed data and can be used to predict new values

Since we do not know this true value, we have to **estimate** it

A possible way to estimate a “good” value for π consists in looking for the π that best explains the observed tosses $x_1 \dots x_n$

We thus define the **likelihood** function

$$\mathcal{L}(\pi) = f_{X_1 \dots X_n | \pi}(x_1 \dots x_n | \pi) = P(X_1 = x_1 \dots X_n = x_n | \pi)$$

Note that π is not treated as a random value, but only as an unknown value

Density estimation

We assume again that, regardless of the value of π , the tosses are independent:

$$P(X_1 = x_1 \dots X_n = x_n | \pi) = \prod_{i=1}^n P(X_i = x_i | \pi)$$

and since

$$P(X_i = x_i | \pi) = \text{Ber}(x_i | \pi) = \pi^{x_i} (1 - \pi)^{1-x_i}$$

the likelihood becomes

$$\mathcal{L}(\pi) = \prod_{i=1}^n \pi^{x_i} (1 - \pi)^{1-x_i}$$

Density estimation

We can compute the **Maximum Likelihood** estimate (ML) for π as the value that maximizes the likelihood:

$$\pi_{ML}^* = \arg \max_{\pi} \mathcal{L}(\pi) = \arg \max_{\pi} \prod_{i=1}^n \pi^{x_i} (1 - \pi)^{1-x_i}$$

π^* is the parameter π which maximizes the probability of the observed sequence of toss

To solve for π , we consider the logarithm of the likelihood³

$$\ell(\pi) = \log \mathcal{L}(\pi)$$

Since the logarithm is monotonically increasing, maximizing ℓ is equivalent to maximizing \mathcal{L}

³We will often work with logarithms of densities, for two reasons: (i) many expressions will be simpler in the log domain, (ii) as we will see in the laboratories, working directly with densities often leads to numerical issues

Density estimation

The expression for $\ell(\pi)$ is

$$\ell(\pi) = \log \mathcal{L}(\pi) = \sum_{i=1}^n x_i \log \pi + (1 - x_i) \log(1 - \pi)$$

We set the derivative of $\ell(\pi)$ equal to zero:

$$\frac{d\ell}{d\pi} = \sum_{i=1}^n \frac{x_i}{\pi} - \frac{(1 - x_i)}{1 - \pi} = 0$$

and we obtain

$$\pi_{ML}^* = \frac{1}{n} \sum_{i=1}^n x_i$$

Density estimation

We can finally predict the outcome for a new toss:

$$P(X_t = 1 | X_1 = x_1 \dots X_n = x_n) = \pi_{ML}^*$$

We have “condensed” the knowledge of the observed outcomes in the estimate π_{ML}^* (which is a function of $x_1 \dots x_n$)

Given enough observations, our estimation will often be “good”.

Consider, however, a simple scenario where our observations are just 3 heads. The ML estimate would be $\pi_{ML}^* = 1$, i.e., we predict that the coin will never land a tail

The issue raises from the fact that we use a single value (the estimate π_{ML}^* of π_T), and we do not consider other possible, even if less likely, values — we are neglecting our **uncertainty** over the estimated value

Density estimation

The Bayesian approach addresses this issue considering the unknown parameters as **random** values, that can be described in terms of **probability distributions**

In contrast with the frequentist approach, we do not require the notion of a “true” value for π

What we need is to specify a **distribution** for π

The Bayesian approach thus requires us to specify our knowledge about the possible values that π may assume in terms of a **prior** distribution $f_{\Pi}(\pi)$

The prior distribution reflects our “knowledge of the world” before we observed the data

Density estimation

We can combine the likelihood and the prior to compute the **posterior** distribution for the model parameters

The posterior distribution is the distribution for the model parameters **given the observed values**

It can be computed from Bayes rule:

$$\begin{aligned} f_{\Pi|X_1 \dots X_n}(\pi|x_1 \dots x_n) &= \frac{P(X_1 = x_1 \dots X_n = x_n|\pi)f_{\Pi}(\pi)}{P(X_1 = x_1 \dots X_n = x_n)} \\ &= \frac{P(X_1 = x_1 \dots X_n = x_n|\pi)f_{\Pi}(\pi)}{\int P(X_1 = x_1 \dots X_n = x_n|\pi)f_{\Pi}(\pi)d\pi} \end{aligned}$$

The posterior distribution reflects our knowledge of the parameters once we have observed the data

It combines both our prior information and the knowledge provided by the likelihood

Density estimation

Assuming again that observations are independent, given the model parameter π , the posterior distribution allows computing the **predictive** distribution as

$$\begin{aligned} P(X_t = x_t | X_1 = x_1 \dots X_n = x_n) \\ &= \int P(X_t | \Pi = \pi, X_1 = x_1 \dots X_n = x_n) f_{\Pi|X_1\dots X_n}(\pi | x_1 \dots x_n) d\pi \\ &= \int P(X_t | \Pi = \pi) f_{\Pi|X_1\dots X_n}(\pi | x_1 \dots x_n) d\pi \end{aligned}$$

i.e., by marginalization of the **joint** distribution for x_t, π given the observed samples

The Bayesian approach has two main limitations

- The choice of the prior is somewhat arbitrary
- The computation of the posterior may be intractable in practice

On the other hand, Bayesian models tend to provide better results for scenarios where data is scarce, since they can account for parameter uncertainty

Density estimation

When modeling continuous values, Gaussian distributions arise naturally in a wide variety of contexts

For example, we have seen that the distribution of sums of i.i.d. R.V.s converges to Gaussian distributions

In many cases we observe that data histogram present Gaussian-like shapes

Furthermore, the Gaussian density is easy to work with

For these reasons, in many cases it's reasonable to assume that our data have been generated by Gaussian R.V.s

Density estimation

Let's assume we have some data $\mathcal{D} = (x_1, \dots, x_n)$

We decide to model the data as samples of a Gaussian distribution, with mean μ and variance ν (and precision $\lambda = \nu^{-1}$)

We assume that the points have been generated by independent, identically distributed R.V.s $X_i \sim X$

Given the values of the model parameter $\theta = (\mu, \nu)$, the distribution of X is

$$f_{X|\theta}(x) = \mathcal{N}(x|\mu, \nu)$$

As for the discrete case, we can express the likelihood for θ as

$$\mathcal{L}(\theta) = \prod_{i=1}^n f_{X_i|\theta}(x_i) = \prod_{i=1}^n \mathcal{N}(x_i|\mu, \nu)$$

Density estimation

We again consider a frequentist approach, which assumes the existence of “true”, but unknown values, for the model parameters $\theta = (\mu, \nu)$

If we knew the values of these parameters μ_T, ν_T , then the density for unseen samples X_t would be

$$f_{X_t}(x_t) = \mathcal{N}(x_t | \mu_T, \nu_T)$$

Since we don't know the model parameters, we again need a way to estimate them

Density estimation

As for the discrete case, we want to find an **estimator** of μ and ν , possibly close to the true values μ_T, ν_T

We have already seen an example of estimator, the Maximum Likelihood estimator (we will show the ML solution for Gaussians in a moment)

In general, an estimator is a function T of the data, that maps our dataset \mathcal{D} to values for the model parameters θ^*

$$\theta^* = T(\mathcal{D})$$

Density estimation

We would like estimators that are **consistent**, i.e. that converge (in probability) to the “true” distribution parameter θ_T as the sample size n grows to infinity

A simple way to produce (under mild assumptions) consistent estimators consists in matching the moments of the assumed distribution to those of the data

This method is called **method of moments** (MOM)

For the Gaussian distribution, the first two moments are μ (first order moment) and ν (centered second order moment)

Density estimation

We can write the equations that match the moments to the empirical mean and covariance of the data, obtaining

$$\mu_{MOM}^* = \frac{1}{n} \sum_i x_i$$

$$v_{MOM}^* = \frac{1}{n} \sum_i (x_i - \mu_{MOM}^*)^2$$

The MOM produces, in this case, consistent estimators

Density estimation

The MOM approach does not, in general, produce very accurate estimators

We have already encountered another estimator, that is widely used in practice: the **Maximum Likelihood** (ML) estimator

The Maximum Likelihood estimator is the value that maximizes the likelihood

$$\theta_{ML}^* = \arg \max_{\theta} \mathcal{L}(\theta)$$

We can regard the ML solution as the parameter that best explains the observed dataset \mathcal{D} , i.e. the value for which it's most likely that we observe \mathcal{D} .

Density estimation

We can derive the ML estimator for the Gaussian distribution

Very often it's more practical to work with the logarithm of the likelihood⁴

$$\ell(\theta) = \log \mathcal{L}(\theta)$$

Since the logarithm is monotonically increasing, maximizing ℓ is equivalent to maximizing \mathcal{L} :

$$\arg \max_{\theta} \ell(\theta) = \arg \max_{\theta} \mathcal{L}(\theta)$$

⁴We will shortly see that the log-pdf of the Gaussian has a simple expression. Furthermore, product of densities transform to sum of log-densities. Also, as we will see in the laboratory, working with densities rather than log-densities often results in numerical problems

Density estimation

Since we assumed that X_i are independent, and $X_i \sim X$, then

$$f_{X_1 \dots X_n}(x_1 \dots x_n | \theta) = \prod_{i=1}^n f_{X_i}(x_i | \theta) = \prod_{i=1}^n f_X(x_i | \theta)$$

The log-likelihood is then

$$\ell(\theta) = \log \mathcal{L}(\theta) = \log \prod_{i=1}^n f_X(x_i | \theta) = \sum_{i=1}^n \log f_X(x_i | \theta)$$

Plugging in the Gaussian density: $X \sim \mathcal{N}(\mu, v)$:

$$\ell(\theta) = \sum_{i=1}^n \log \mathcal{N}(x_i | \mu, v) = \sum_{i=1}^n \log \mathcal{N}(x_i | \mu, \lambda^{-1})$$

In the following we parametrize the density in terms of precision λ , since this allows for simpler expressions

Density estimation

In the log domain the Gaussian log-pdf has a simple expression:

$$\log \mathcal{N}(x|\mu, \lambda^{-1}) = \xi_1 + \frac{1}{2} \log \lambda - \frac{\lambda}{2} (x - \mu)^2$$

where ξ_1 collects constant terms that are irrelevant for the optimization (they do not depend on the parameters)

The log-likelihood is then

$$\begin{aligned}\ell(\theta) &= \sum_{i=1}^n \log \mathcal{N}(x_i|\mu, \lambda^{-1}) = \xi_2 + \sum_{i=1}^n \left[\frac{1}{2} \log \lambda - \frac{1}{2} \lambda (x_i - \mu)^2 \right] \\ &= \xi_2 + \frac{1}{2} n \log \lambda - \frac{1}{2} \lambda \sum_{i=1}^n x_i^2 - \lambda \mu \sum_{i=1}^n x_i + \frac{1}{2} n \lambda \mu^2\end{aligned}$$

and ξ_2 collects all constant terms

Density estimation

The ML estimate can be obtained by taking solving for

$$\begin{cases} \frac{\partial \ell}{\partial \mu} = 0 \\ \frac{\partial \ell}{\partial \lambda} = 0 \end{cases}$$

The first derivative is

$$\frac{\partial \ell}{\partial \mu} = n\lambda\mu - \lambda \sum_{i=1}^n x_i$$

thus

$$\mu_{ML}^* = \frac{1}{n} \sum_{i=1}^n x_i$$

Density estimation

The derivative with respect to the precision is

$$\frac{\partial \ell}{\partial \lambda} = \frac{n}{2\lambda} - \frac{1}{2} \left[\sum_{i=1}^n (x_i - \mu)^2 \right]$$

thus

$$v_{ML}^* = (\lambda_{ML}^*)^{-1} = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_{ML}^*)^2$$

The solution thus corresponds to the empirical mean and empirical covariance matrix of the data

In this case, we can observe that the solution is the same as the one obtained by the MOM approach (this does not hold in general)

Density estimation

As in the binary case, we can use the ML estimates to form the predictive distribution

$$f_{X_t|X_1 \dots X_n}(x_t | x_1 \dots x_n) \approx \mathcal{N}(x_t | \mu_{ML}^*, v_{ML}^*)$$

Linear and Quadratic Classifiers I — Generative models

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Generative Classifiers

We consider a (closed set) classification problem

We have a pattern x_t that we want to classify as belonging to one of k classes

Probabilistic model: we assume that x_t is a realization of R.V. X_t

We also assume that its (unknown) class label can be described by R.V. $C_t \in \{1 \dots k\}$

$1 \dots k$ are the class labels¹

¹The actual values used to represent the classes are irrelevant, without loss of generality we assume classes are labeled using progressive integers. For binary problems, we will, in some cases, encode classes with $\{1, 0\}$.

Generative Classifiers

Optimal Bayes decision: assign the class with highest posterior probability $c_t^* = \arg \max_c P(C_t = c | X_t = \mathbf{x}_t)$

For example, we can consider an object classification task

\mathbf{x}_t is the representation of an image

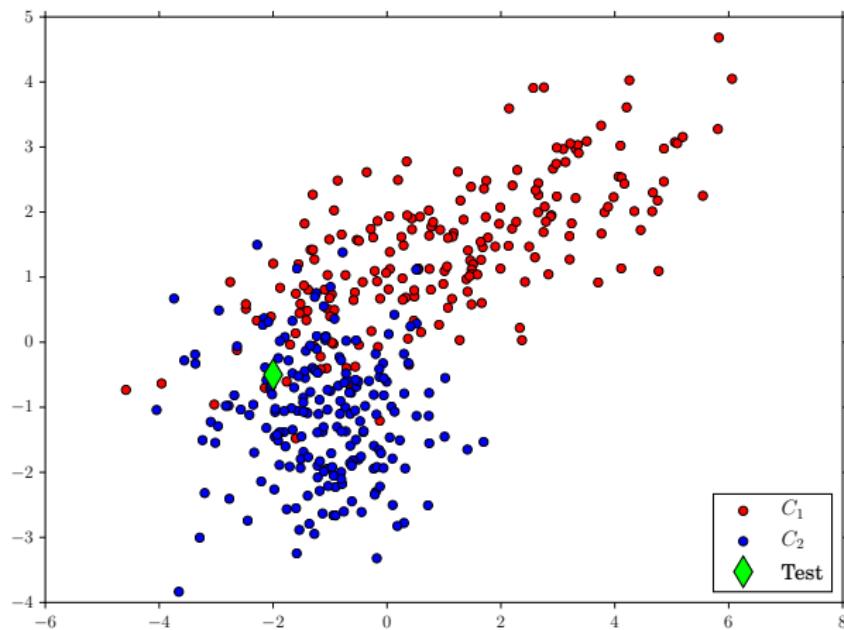
Labels represent what object is depicted (e.g. cat = 1, dog = 2, rabbit = 3, ...)

We want to find which label c_t is more likely for \mathbf{x}_t

For all labels $c \in \{1 \dots K\}$, we compute $P(C_t = c | X_t = \mathbf{x}_t)$, i.e. the probability that the class C_t for the test sample t is c , conditioned on the observed value $X_t = \mathbf{x}_t$

Generative Classifiers

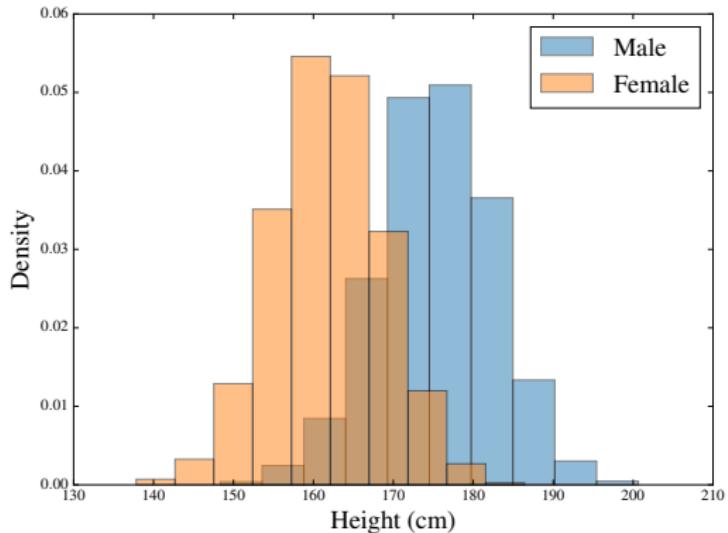
A binary example



What class is the green sample from?

Generative Classifiers

A univariate, binary example: forensics — infer the gender from the height



What's the gender of a 174 cm tall suspect?

Generative Classifiers

Simple model: assume that the samples are independent and distributed according to $X_t, C_t \sim X, C$, for any test sample t

Let the joint density of X, C be $f_{X,C}$

We can compute the joint likelihood for the hypothesized class c for the observed test sample x_t :

$$f_{X_t, C_t}(x_t, c) = f_{X,C}(x_t, c)$$

Since we are considering a closed-set classification problem, from Bayes rule we can compute the class **posterior** probability

$$P(C_t = c | X_t = x_t) = \frac{f_{X,C}(x_t, c)}{\sum_{c' \in \mathcal{C}} f_{X,C}(x_t, c')}$$

Generative Classifiers

The joint density for (X_t, C_t) can be expressed as

$$f_{X_t, C_t}(\mathbf{x}_t, c) = f_{X, C}(\mathbf{x}_t, c) = f_{X|C}(\mathbf{x}_t|c)P_C(c)$$

$P_C(c)$, or simply $P(c)$, is the **prior** probability for class c , and represents the probability of the class being c , before we observe the test sample (e.g. how likely we believe that a picture will depict a cat)

Usually, $P(c)$ is **application-dependent**: for example, in the desert we may have a prior probability for next day being rainy $P(rain) = 0.001$, whereas in the rain forest we may have $P(rain) = 0.5$

Our objective: modeling the **class-conditional** distribution $X|C$, i.e. we want to estimate the density

$$f_{X|C}(\mathbf{x}_t|c)$$

Gaussian Classifier

We consider problems where the observations are continuous
 $x \in \mathbb{R}$

How can we model $f_{X|C}(x_t|c)$?

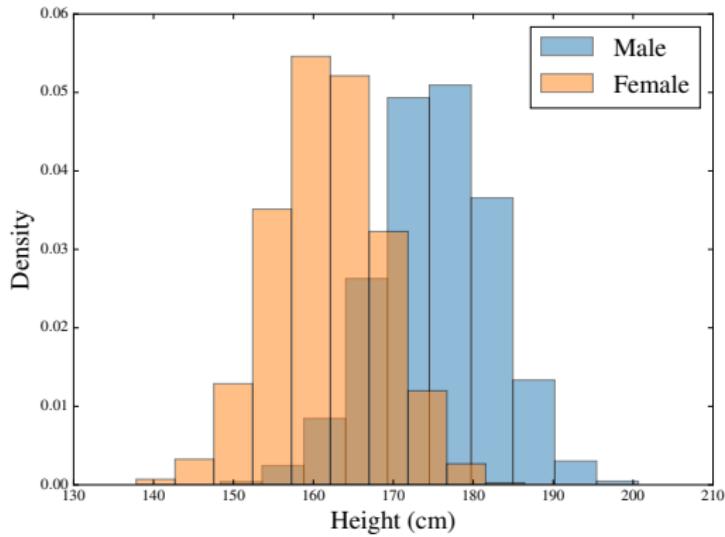
Of course, the answer depends on the data

In the following, we assume that the data of each class can be modeled by a (Multivariate) Gaussian Distribution

We will need to verify the performance of the model

Generative Classifiers

We start with a univariate example — gender inference

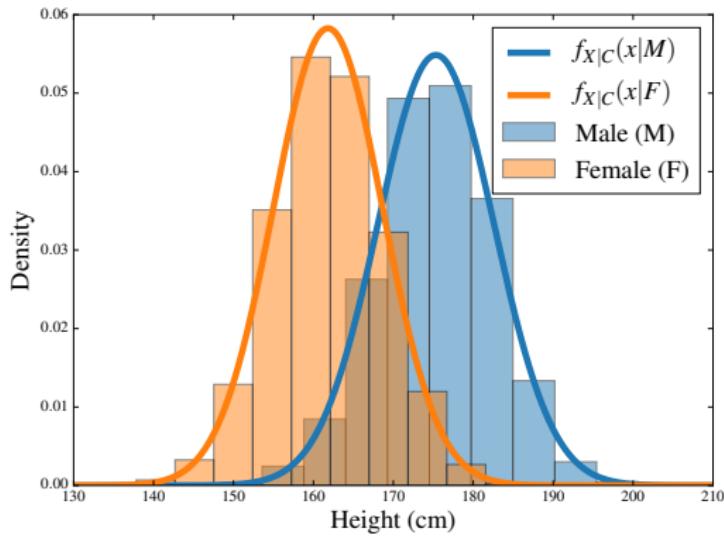


We assume we have a large set of height measurements for the population under consideration

Generative Classifiers

Intuitively, we can fit a Gaussian density over the samples of each class

We can use ML estimates to fit a Male ($C = M$) and a Female ($C = F$) Gaussian



Generative Classifiers

The ML parameters are the mean and variance of the samples of the Male and Female classes, respectively²

$$\mu_M = \frac{1}{N_M} \sum_{i|C_i=M} x_i \approx 175.33 \text{ cm}, \quad \sigma_M^2 = \frac{1}{N_M} \sum_{i|C_i=M} (x_i - \mu_M)^2 \approx 52.89 \text{ cm}^2$$

$$\mu_F = \frac{1}{N_F} \sum_{i|C_i=F} x_i \approx 161.82 \text{ cm}, \quad \sigma_F^2 = \frac{1}{N_F} \sum_{i|C_i=F} (x_i - \mu_F)^2 \approx 46.89 \text{ cm}^2$$

N_M and N_F are the number of male and female samples, and the sums extend over the male (first row) or female (second row) samples of the dataset

We are now able to compute the likelihood for the two classes for the 174 cm tall suspect

$$f_{X|C}(174|M) = \mathcal{N}(174|\mu_M, \sigma_M^2) \approx 0.05395$$

$$f_{X|C}(174|F) = \mathcal{N}(174|\mu_F, \sigma_F^2) \approx 0.01198$$

²In the following slides we will drop the unit of measurement

Generative Classifiers

It's approximately 4.5 times more likely to observe a height of 174 cm in the male population

However, this is not sufficient to answer whether the sample is from a male or a female

We need to compute the class **posterior** probability, which depends also on the class **prior** probability

$$P(C = M|X = 174) = \frac{f_{X|C}(174|M)P(C = M)}{f_X(174)}$$

$$P(C = F|X = 174) = \frac{f_{X|C}(174|F)P(C = F)}{f_X(174)}$$

If we want to just compare the two probabilities, we don't need to compute the normalization term $f_X(174)$

Gaussian Classifier

The likelihood ratio for the two hypotheses is

$$\frac{P(C = M|X = 174)}{P(C = F|X = 174)} = \frac{f_{X|C}(174|M)}{f_{X|C}(174|F)} \frac{P(C = M)}{P(C = F)}$$

The prior probabilities represent the probability that, a priori, we expect to observe a male or female sample

In this example, we may not have any knowledge of the suspect gender, so we may assume $P(C = M) = P(C = F) = \frac{1}{2}$

In this case

$$\frac{P(C = M|X = 174)}{P(C = F|X = 174)} = \frac{f_{X|C}(174|M)}{f_{X|C}(174|F)} \approx 4.5$$

i.e., the probability that the sample is from a male is 4.5 times higher than the probability that it is from a female

Gaussian Classifier

In other cases, we may have other information sources that lead us believe that the suspect is more likely to be from a specific gender.

As an example, we may believe for other reasons that the probability that the suspect is female is 90%: $P(C = F) = 0.9$ and $P(C = M) = 0.1$

In this case, the posterior likelihood ratio becomes

$$\frac{P(C = M|X = 174)}{P(C = F|X = 174)} = \frac{f_{X|C}(174|M)}{f_{X|C}(174|F)} \frac{P(C = M)}{P(C = F)} \approx \frac{4.5}{9} = \frac{1}{2}$$

i.e., the probability that the suspect is female is still twice the probability that the suspect is male, *even though* the evidence suggests otherwise

In this case, the evidence is not strong enough to change our prior belief.

Gaussian Classifier

We will now formalize the method we just employed in the example

We assume that our data, given the class, can be described by a Gaussian distribution

$$(\mathbf{X}_t | C_t = c) \sim (\mathbf{X} | C = c) \sim \mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

We have one mean and one covariance matrix per class

If we knew $\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c$, then we could compute $f_{\mathbf{X}_t | C_t = c}$ as

$$f_{\mathbf{X}_t | C_t}(x_t | c) = f_{\mathbf{X} | C}(x_t | c) = \mathcal{N}(x_t | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

We do not, however, know the values for the model parameters
 $\boldsymbol{\theta} = [(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \dots (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)]$

Gaussian Classifier

On the other hand, we have at our disposal a labeled **training** dataset

$$\mathcal{D} = \{(\mathbf{x}_1, c_1) \dots (\mathbf{x}_n, c_n)\}$$

$\mathcal{X} = \{\mathbf{x}_1 \dots \mathbf{x}_n\}$ are the observed samples

$\mathcal{C} = \{c_1 \dots c_n\}$ are the corresponding class labels $c_i \in \{1 \dots k\}$

We want to learn the model parameters from the data

Gaussian Classifier

We assume that, given the model parameters θ , observations are **independent and identically distributed** (i.i.d.)

$$[(X_i, C_i) \perp\!\!\!\perp (X_j, C_j)] | \theta$$

and

$$\forall i, \quad (X_i, C_i) | \theta \sim (X, C) | \theta$$

i.e., we assume that both the **training** set and **evaluation** samples are independent (given the model parameters) and they are distributed in the same way

Gaussian Classifier

Since we assume Gaussian distribution for $X|C$, we have

$$(X_i|C_i = c, \boldsymbol{\theta}) \sim (X_t|C_t = c, \boldsymbol{\theta}) \sim (X|C = c, \boldsymbol{\theta}) \sim \mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

i.e., the class-conditional distribution for all observations is a Gaussian with class-dependent mean $\boldsymbol{\mu}_c$ and class-dependent covariance matrix $\boldsymbol{\Sigma}_c$

Again, the model parameters are $\boldsymbol{\theta} = [(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \dots (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)]$

We follow a frequentist approach, and we thus want to compute an **estimator** (or point estimate) $\boldsymbol{\theta}^*$ of the model parameters

We will then use the estimated parameters to compute

$$f_{X_t|C_t}(\mathbf{x}_t|c) \approx \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_c^*, \boldsymbol{\Sigma}_c^*)$$

We have seen that a possible way to estimate the model parameters is to maximize the data (log-)likelihood

The likelihood for θ is

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= f_{X_1 \dots X_n, C_1 \dots C_n | \boldsymbol{\theta}}(\mathbf{x}_1 \dots \mathbf{x}_n, c_1 \dots c_n | \boldsymbol{\theta}) \\ &= \prod_{i=1}^n f_{X_i, C_i | \boldsymbol{\theta}}(\mathbf{x}_i, c_i | \boldsymbol{\theta}) \\ &= \prod_{i=1}^n f_{X_i | C_i, \boldsymbol{\theta}}(\mathbf{x}_i | c_i, \boldsymbol{\theta}) P(c_i) \\ &= \prod_{i=1}^n \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_{c_i}, \boldsymbol{\Sigma}_{c_i}) P(c_i)\end{aligned}$$

where the factorization derives from the i.i.d. assumptions

Gaussian Classifier

We again consider the log-likelihood

$$\begin{aligned}\ell(\boldsymbol{\theta}) &= \log \mathcal{L}(\boldsymbol{\theta}) \\ &= \sum_{i=1}^n \log \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_{c_i}, \boldsymbol{\Sigma}_{c_i}) + \sum_i \log P(c_i) \\ &= \sum_{c=1}^k \sum_{i|c_i=c} \log \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) + \xi\end{aligned}$$

where ξ collects terms that do not depend on $\boldsymbol{\theta}$, and thus are irrelevant for the maximization with respect to $\boldsymbol{\theta}$.

The log-likelihood corresponds to a sum over all classes of the conditional log-likelihood of the samples belonging to each class

$$\ell(\boldsymbol{\theta}) = \sum_{c=1}^k \ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) + \xi, \quad \ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = \sum_{i|c_i=c} \log \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

We can thus maximize ℓ by separately maximizing the terms $\ell_c(\mu_c, \Sigma_c)$

But $\ell_c(\mu_c, \Sigma_c)$ is simply the log-likelihood of a Gaussian model for the data of class c

We are independently estimating the Gaussian densities that best describe the data of each class c

For univariate R.V.s, we have already shown that the ML solution corresponds to the class mean and covariance matrix

We now consider the general case for multivariate samples

Gaussian Classifier

The log-density for a Gaussian distribution $\mathcal{N}(\mu, \Sigma)$ is

$$\log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{D}{2} \log 2\pi - \frac{1}{2} \log |\boldsymbol{\Sigma}| - \frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})$$

or, in terms of precision matrix $\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1}$

$$\log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{D}{2} \log 2\pi + \frac{1}{2} \log |\boldsymbol{\Lambda}| - \frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Lambda} (\mathbf{x} - \boldsymbol{\mu})$$

Gaussian Classifier

The log-likelihood $\ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$ can thus be expressed as

$$\ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = k + \frac{N_c}{2} \log |\boldsymbol{\Lambda}_c| - \frac{1}{2} \sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu}_c)^T \boldsymbol{\Lambda}_c (\mathbf{x}_i - \boldsymbol{\mu}_c)$$

We can rewrite the log-likelihood in different ways:

$$\ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = k + \frac{N_c}{2} \log |\boldsymbol{\Lambda}_c| - \frac{1}{2} \text{Tr} \left(\boldsymbol{\Lambda}_c \sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu}_c)(\mathbf{x}_i - \boldsymbol{\mu}_c)^T \right) \quad (1)$$

$$= k + \frac{N_c}{2} \log |\boldsymbol{\Lambda}_c| - \frac{1}{2} \sum_{i|c_i=c} \mathbf{x}_i^T \boldsymbol{\Lambda}_c \mathbf{x}_i + \boldsymbol{\mu}_c^T \boldsymbol{\Lambda}_c \sum_{i|c_i=c} \mathbf{x}_i - \frac{N_c}{2} \boldsymbol{\mu}_c^T \boldsymbol{\Lambda}_c \boldsymbol{\mu}_c$$

$$= k + \frac{N_c}{2} \log |\boldsymbol{\Lambda}_c| - \frac{1}{2} \text{Tr} \left(\boldsymbol{\Lambda}_c \sum_{i|c_i=c} \mathbf{x}_i \mathbf{x}_i^T \right) + \boldsymbol{\mu}_c^T \boldsymbol{\Lambda}_c \sum_{i|c_i=c} \mathbf{x}_i - \frac{N_c}{2} \boldsymbol{\mu}_c^T \boldsymbol{\Lambda}_c \boldsymbol{\mu}_c \quad (2)$$

Gaussian Classifier

From (2) we observe that the log-likelihood depends on the data only through the **statistics**

$$Z_c = N_c$$

$$\mathbf{F}_c = \sum_{i|c_i=c} \mathbf{x}_i$$

$$\mathbf{S}_c = \sum_{i|c_i=c} \mathbf{x}_i \mathbf{x}_i^T$$

These are also called **sufficient statistics**: they collect all the information contained in the dataset that is relevant for the estimation of μ_c and Σ_c

Gaussian Classifier

We can find the maximum of ℓ_c by taking the derivatives of ℓ_c and setting them equal to 0:

$$\begin{cases} \nabla_{\Lambda_c} \ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Lambda}_c) = \mathbf{0} \\ \nabla_{\boldsymbol{\mu}_c} \ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Lambda}_c) = \mathbf{0} \end{cases}$$

From (2), the derivative with respect to $\boldsymbol{\mu}_c$ is

$$\nabla_{\boldsymbol{\mu}_c} \ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Lambda}_c) = \boldsymbol{\Lambda}_c \sum_{i|c_i=c} \mathbf{x}_i - N_c \boldsymbol{\Lambda}_c \boldsymbol{\mu}_c$$

Solving for $\nabla_{\boldsymbol{\mu}_c} \ell_c(\boldsymbol{\mu}_c, \boldsymbol{\Lambda}_c) = \mathbf{0}$ gives

$$\boldsymbol{\mu}_c = \frac{1}{N_c} \sum_{i|c_i=c} \mathbf{x}_i$$

i.e., the mean of samples belonging to class c

Gaussian Classifier

From (1), we can compute the derivative³ require w.r.t. Λ_c :

$$\nabla_{\Lambda_c} \ell_c(\mu_c, \Lambda_c) = \frac{N_c}{2} \Lambda_c^{-T} - \frac{1}{2} \sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T$$

Assuming that $\sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T$ is positive-definite (we can check that it's also symmetric), solving for Λ_c^{-1} gives

$$\Sigma_c = \Lambda_c^{-1} = \frac{1}{N_c} \sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu}_c)(\mathbf{x}_i - \boldsymbol{\mu}_c)^T$$

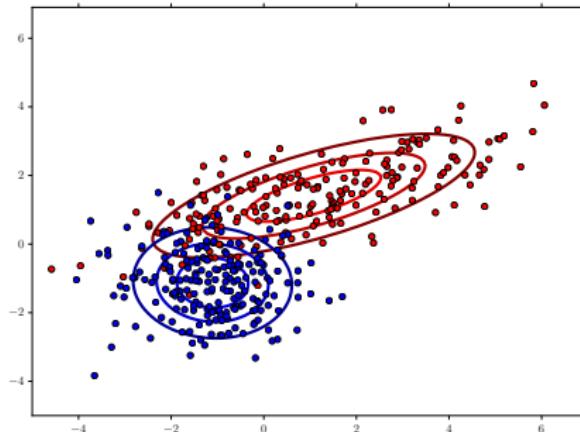
i.e., the covariance matrix of samples belonging to class c , computed using the data mean $\boldsymbol{\mu}_c$

³Since Λ_c is a precision matrix, it should be symmetric positive definite. We therefore should maximize ℓ_c under such constraint. In practice, we solve the problem for unconstrained Λ_c , and then we check whether the unconstrained solution satisfies the constraints. Since it does, it's also the optimal solution for the constrained problem

Gaussian Classifier

Summarizing, the ML solution is given by

$$\boldsymbol{\mu}_c^* = \frac{1}{N_c} \sum_{i|c_i=c} \mathbf{x}_i, \quad \boldsymbol{\Sigma}_c^* = \frac{1}{N_c} \sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu}_c^*)(\mathbf{x}_i - \boldsymbol{\mu}_c^*)^T$$



We can then compute the likelihood of class c for test point \mathbf{x}_t as

$$f_{X_t|C_t}(\mathbf{x}_t|c) = f_{X|C}(\mathbf{x}_t|c) = \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_c^*, \boldsymbol{\Sigma}_c^*)$$

Gaussian Classifier

Let's now consider a binary task, with two classes⁴ $C \in \{h_1, h_0\}$

We assign the label to a test sample \mathbf{x}_t according to the highest posterior probability, comparing $P(C = h_1|\mathbf{x}_t)$ to $P(C = h_0|\mathbf{x}_t)$

We can express the comparison in terms of class posterior ratio

$$r(\mathbf{x}_t) = \frac{P(C = h_1|\mathbf{x}_t)}{P(C = h_0|\mathbf{x}_t)}$$

or, alternatively, in terms of the class-posterior **log-likelihood ratio**

$$\log r(\mathbf{x}_t) = \log \frac{P(C = h_1|\mathbf{x}_t)}{P(C = h_0|\mathbf{x}_t)}$$

⁴For binary problems it's common to label classes as 1 (target hypothesis, true hypothesis, ...) and 0 (non-target hypothesis, false hypothesis, null hypothesis, ...). As we have already said, the chosen labeling scheme is irrelevant for our discussion — we denote the class labels as h_1 and h_0

If the log-ratio is greater than 0, then the point will be assigned to class h_1 , otherwise it will be assigned to class h_0

The class posterior ratio can be rewritten to make explicit its dependency on the likelihoods $f_{X|C}(\mathbf{x}_t|c)$ and prior class probabilities:

$$\begin{aligned}\log r(\mathbf{x}_t) &= \log \frac{P(C = h_1|\mathbf{x}_t)}{P(C = h_0|\mathbf{x}_t)} \\&= \log \frac{f_{X,C}(\mathbf{x}_t, h_1)}{f_X(\mathbf{x}_t)} \cdot \frac{f_X(\mathbf{x}_t)}{f_{X,C}(\mathbf{x}_t, h_0)} \\&= \log \frac{f_{X|C}(\mathbf{x}_t|h_1)P(C = h_1)}{f_{X|C}(\mathbf{x}_t|h_0)P(C = h_0)} \\&= \log \frac{f_{X|C}(\mathbf{x}_t|h_1)}{f_{X|C}(\mathbf{x}_t|h_0)} + \log \frac{P(C = h_1)}{P(C = h_0)}\end{aligned}$$

Gaussian Classifier

The first element of the sum is the **log-likelihood ratio**

$$llr(\mathbf{x}_t) = \log \frac{f_{X|C}(\mathbf{x}_t|h_1)}{f_{X|C}(\mathbf{x}_t|h_0)}$$

It represents the ratio between the likelihood of observing the sample given that it belongs to h_1 or to h_0

The second term represents the prior (log)-odds. For a binary problem, we have

$$P(C = h_1) = \pi, \quad P(C = h_0) = 1 - P(C = h_1) = 1 - \pi$$

thus

$$\log r(\mathbf{x}_t) = \log \frac{f_{X|C}(\mathbf{x}_t|h_1)}{f_{X|C}(\mathbf{x}_t|h_0)} + \log \frac{\pi}{1 - \pi}$$

As we have mentioned, π reflects the prior probability for class h_1 given a specific application

The first term, the log-likelihood ratio, is what our system should focus on providing — we want our system to be application-agnostic as much as possible

Whoever will use the system will plug the prior probabilities for his own task, and compute posterior log-probability ratios

Gaussian Classifier

The optimal decision is based on the comparison

$$\log r(\mathbf{x}_t) \gtrless 0$$

which means that we compare

$$\log r(\mathbf{x}_t) = \log \frac{f_{X|C}(\mathbf{x}_t|h_1)}{f_{X|C}(\mathbf{x}_t|h_0)} + \log \frac{\pi}{1-\pi} \gtrless 0$$

i.e., we assign classes based on

$$llr(\mathbf{x}_t) = \log \frac{f_{X|C}(\mathbf{x}_t|h_1)}{f_{X|C}(\mathbf{x}_t|h_0)} \gtrless -\log \frac{\pi}{1-\pi}$$

The log-likelihood ratio acts as a **score**, with a probabilistic interpretation

Greater scores values imply our system favors class h_1 , lower values mean it favors class h_0

The decision requires comparing the score to a threshold t that depends on the application, through the class prior probability π

Later we shall see that we can (and should) also account for different costs for different kind of errors — we will show that this corresponds to using different **effective** priors

Gaussian Classifier

Let's see what kind of decision surfaces correspond to the llr of the Gaussian classifier with parameters $[(\mu_1, \Lambda_1^{-1}), (\mu_0, \Lambda_0^{-1})]$

We can compute the log-likelihood ratio

$$\text{llr}(\mathbf{x}) = \log \frac{\mathcal{N}(\mathbf{x}|h_1)}{\mathcal{N}(\mathbf{x}|h_0)} = \log \frac{\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_1, \boldsymbol{\Lambda}_1^{-1})}{\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0^{-1})}$$

The decision function is **quadratic** in \mathbf{x} :

$$\text{llr}(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{x}^T \mathbf{b} + c$$

with

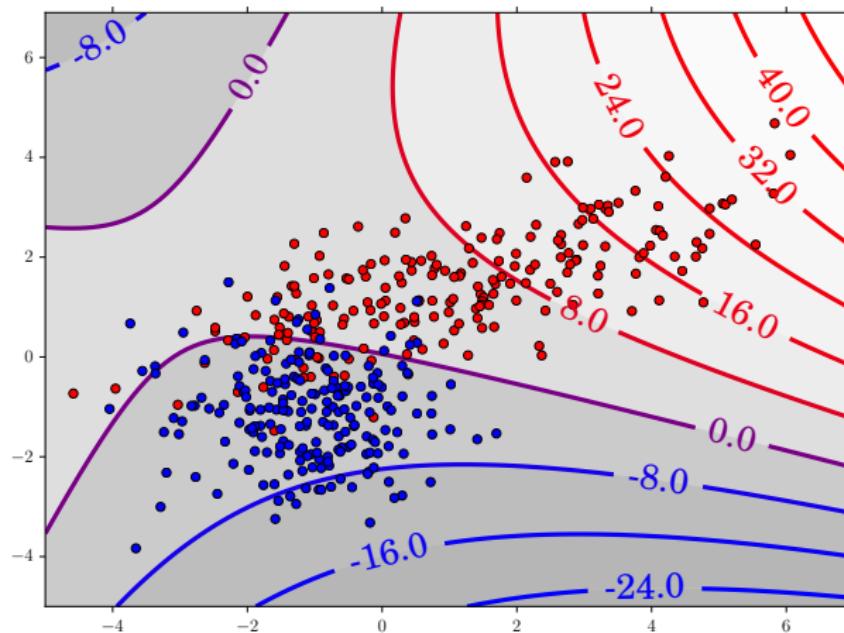
$$\mathbf{A} = -\frac{1}{2} (\boldsymbol{\Lambda}_1 - \boldsymbol{\Lambda}_0)$$

$$\mathbf{b} = (\boldsymbol{\Lambda}_1 \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_0 \boldsymbol{\mu}_0)$$

$$c = -\frac{1}{2} (\boldsymbol{\mu}_1^T \boldsymbol{\Lambda}_1 \boldsymbol{\mu}_1 - \boldsymbol{\mu}_0^T \boldsymbol{\Lambda}_0 \boldsymbol{\mu}_0) + \frac{1}{2} (\log |\boldsymbol{\Lambda}_1| - \log |\boldsymbol{\Lambda}_0|)$$

Gaussian Classifier

Binary problem — decision boundaries



Gaussian Classifier

For multiclass problems $C \in \{h_1, h_2 \dots h_k\}$ we can compute closed-set posterior probabilities as

$$P(C = h|\mathbf{x}_t) = \frac{f_{X|C}(\mathbf{x}_t|h)P(h)}{\sum_{h' \in \{h_1, h_2 \dots h_k\}} f_{X|C}(\mathbf{x}_t|h')P(h')}$$

Optimal decisions require choosing the class with highest posterior probability $c_t^* = \arg \max_h P(C = h|\mathbf{x}_t)$. Posterior probabilities are proportional to

$$P(C = h|\mathbf{x}_t) \propto f_{X|C}(\mathbf{x}_t|h)P(h)$$

and the proportionality factor is the same for all classes

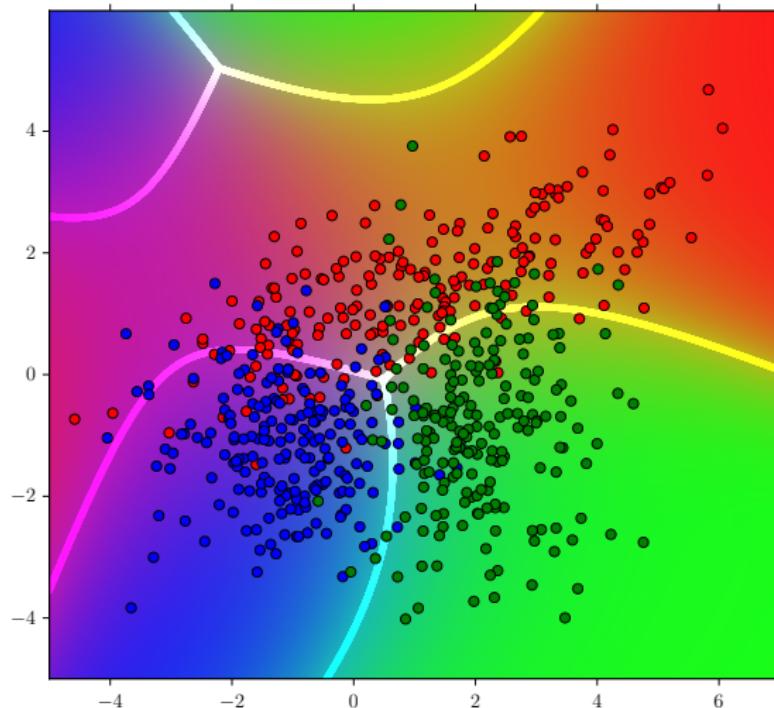
Optimal decisions can thus be computed as

$$c_t^* = \arg \max_h f_{X|C}(\mathbf{x}_t|h)P(h) = \arg \max_h \log f_{X|C}(\mathbf{x}_t|h) + \log P(h)$$

Again, the first term should be the output of the classifier, whereas the second term $\log P(h)$ depends on the application

Gaussian Classifier

3 class problem — class posteriors and pair-wise boundaries



Gaussian Classifier

The Gaussian model requires computing a mean and a covariance matrix for each class

If the samples are few compared to their dimensionality, then the estimates can be inaccurate

The issue is more evident for covariance matrices, since they have $\frac{D \times (D+1)}{2}$ independent elements

The off-diagonal terms of Σ_c represent the covariances of the different components of our feature vectors

If we know that, for each class, the different components are approximately independent, we can simplify the estimate assuming that the distribution of $X|C$ can be factorized over its components

$$f_{X|C}(\mathbf{x}|c) \approx \prod_{j=1}^D f_{X_{[j]}|C}(x_{[j]}|c)$$

where $x_{[j]}$ is the j -th component of \mathbf{x} (not to be confused with x_j , the j -th dataset sample)

This model is called **Naive Bayes**

The assumption is not tied to any specific distribution — we can even employ different distributions for different components

Gaussian Classifier

The naive Bayes assumption, combined with Gaussian assumptions, models the distributions $f_{X_{[j]}|C}(x_{[j]}|c)$ as univariate Gaussians

$$f_{X_{[j]}|C}(x_{[j]}|c) = \mathcal{N}(x_{[j]}|\mu_{c,[j]}, \sigma_{c,[j]}^2)$$

We can again compute the ML estimates. The log-likelihood factorizes over sample components:

$$\mathcal{L}(\boldsymbol{\theta}) \propto \prod_{i=1}^n \prod_{j=1}^D \mathcal{N}(\mathbf{x}_{i,[j]}|\mu_{c_i,[j]}, \sigma_{c_i,[j]}^2)$$

$$\ell(\boldsymbol{\theta}) = \xi + \sum_{c=1}^k \sum_{i|c_i=c} \sum_{j=1}^D \log \mathcal{N}(\mathbf{x}_{i,[j]}|\mu_{c,[j]}, \sigma_{c,[j]}^2)$$

$$= \xi + \sum_{j=1}^D \sum_{c=1}^k \sum_{i|c_i=c} \log \mathcal{N}(\mathbf{x}_{i,[j]}|\mu_{c,[j]}, \sigma_{c,[j]}^2)$$

Gaussian Classifier

We can optimize the log-likelihood independently for each component

For each component, we have the log-likelihood of a Gaussian model

The ML solution is

$$\mu_{c,[j]}^* = \frac{1}{N_c} \sum_{i|c_i=c} x_{i,[j]}, \quad \sigma_{c,[j]}^2 = \frac{1}{N_c} \sum_{i|c_i=c} (x_{i,[j]} - \mu_{c,[j]})^2$$

Gaussian Classifier

We can observe that the density for a sample x can be expressed as

$$f_{X|C}(\mathbf{x}|c) = \prod_{j=1}^D \mathcal{N}(x_{[j]} | \mu_{c,[j]}^* \sigma_{c,[j]}^2) = \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

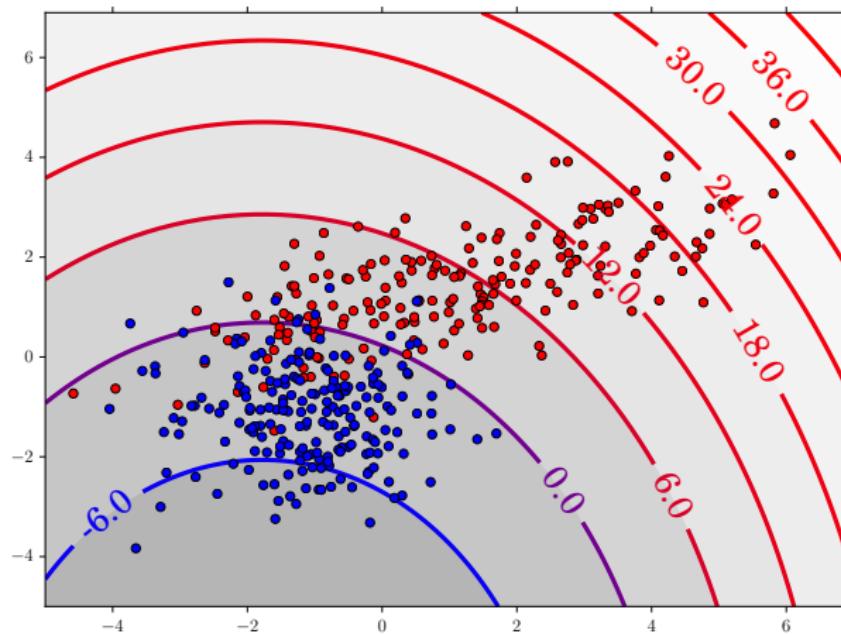
where

$$\boldsymbol{\mu}_c = \begin{bmatrix} \mu_{c,[1]} \\ \mu_{c,[2]} \\ \vdots \\ \mu_{c,[D]} \end{bmatrix}, \quad \boldsymbol{\Sigma}_c = \begin{bmatrix} \sigma_{c,[1]}^2 & 0 & \cdots & 0 \\ 0 & \sigma_{c,[2]}^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_{c,[D]}^2 \end{bmatrix}$$

The **naive Bayes Gaussian** classifier corresponds to a Multivariate Gaussian classifier with **diagonal** covariance matrices (this does not hold for a generic density!)

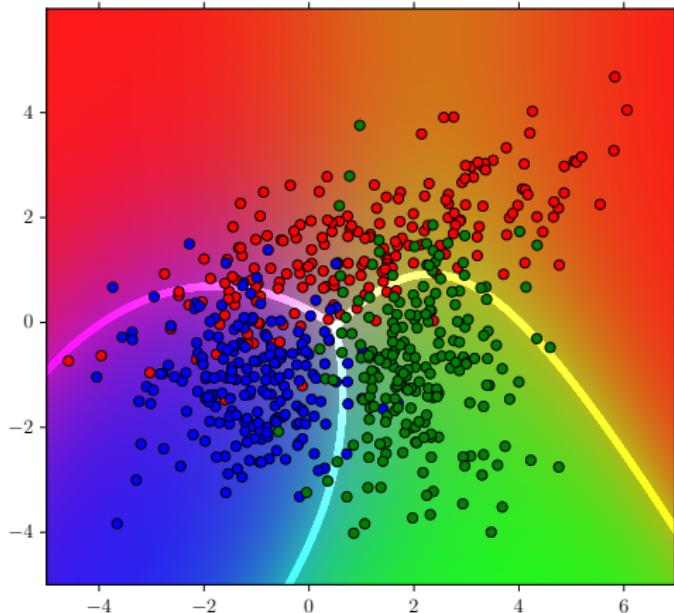
Gaussian Classifier

Binary problem — naive Bayes Gaussian classifier



Gaussian Classifier

3 class problem — naive Bayes Gaussian classifier



Gaussian Classifier

Another common Gaussian model assumes that the covariance matrices of the different classes are **tied**

- Class-independent noise: $x_{c,i} = \mu_c + \varepsilon_i, \varepsilon_i \sim \mathcal{N}(\mathbf{0}, \Lambda^{-1})$
- Badly-conditioned problems (large dimensional data, small number of samples) — A single shared covariance matrix can be more easily estimated

The tied covariance model assumes that

$$f_{X|C}(x|c) = \mathcal{N}(x|\mu_c, \Sigma)$$

i.e., each class has its own mean μ_c , but the covariance matrix is the same for all classes

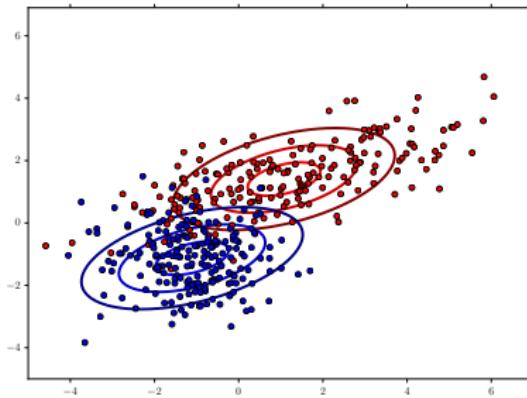
Gaussian Classifier

Again, we can estimate the parameters using the ML framework.
In this case the log-likelihood does not factorize over classes

The ML solution is

$$\boldsymbol{\mu}_c^* = \frac{1}{N_c} \sum_{i|c_i=c} \mathbf{x}_i , \quad \boldsymbol{\Sigma}^* = \frac{1}{N} \sum_c \sum_{i|c_i=c} (\mathbf{x}_i - \boldsymbol{\mu}_c) (\mathbf{x}_i - \boldsymbol{\mu}_c)^T$$

where N is the number of samples $N = \sum_{c=1}^k N_c$



Gaussian Classifier

Let's compute the binary log-likelihood ratios for the tied model:

$$\begin{aligned} llr(\mathbf{x}) &= \log \frac{f_{X|C}(\mathbf{x}|h_1)}{f_{X|C}(\mathbf{x}|h_0)} \\ &= \log \frac{\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_1, \boldsymbol{\Lambda}^{-1})}{\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}^{-1})} \\ &= \mathbf{x}^T \mathbf{b} + c \end{aligned}$$

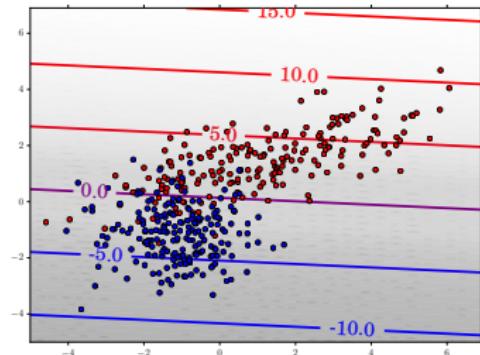
with

$$\begin{aligned} \mathbf{b} &= \boldsymbol{\Lambda} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) \\ c &= -\frac{1}{2} (\boldsymbol{\mu}_1^T \boldsymbol{\Lambda} \boldsymbol{\mu}_1 - \boldsymbol{\mu}_0^T \boldsymbol{\Lambda} \boldsymbol{\mu}_0) \end{aligned}$$

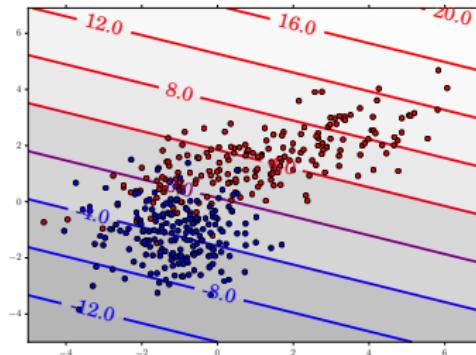
i.e., a **linear** function of \mathbf{x}

Gaussian Classifier

Binary classifier — tied covariances



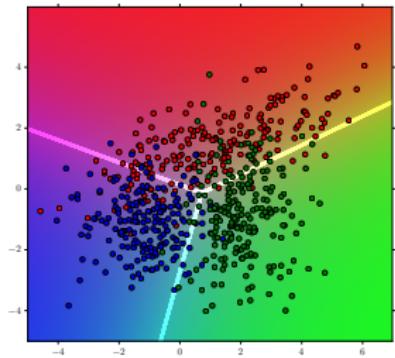
Multivariate



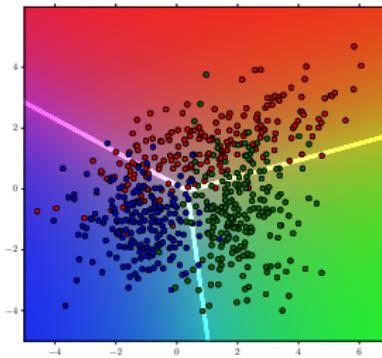
Naive Bayes

Gaussian Classifier

Multiclass classifier — tied covariances



Multivariate



Naive Bayes

Gaussian Classifier

The model is also closely related to LDA

Remember that two-class LDA looks for the direction which maximizes the generalized Rayleigh quotient

$$\frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}$$

with

$$\mathbf{S}_W = \boldsymbol{\Lambda}^{-1}$$

$$\mathbf{S}_B = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T$$

We have seen that we can solve the problem by applying the following transformations

$$\mathbf{x}' = \boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{x}$$

$$\mathbf{S}'_W = \mathbf{I}$$

$$\mathbf{S}'_B = \boldsymbol{\Lambda}^{\frac{1}{2}} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)^T \boldsymbol{\Lambda}^{\frac{1}{2}}$$

Gaussian Classifier

Since $\nu = \Lambda^{\frac{1}{2}}(\mu_1 - \mu_0)$ is a vector, the leading eigenvector of S'_B is

$$\nu = \frac{\nu}{\|\nu\|}$$

Projection over the LDA subspace is, up to a scaling factor, given by

$$w^T x = k \cdot x^T \Lambda (\mu_1 - \mu_0)$$

This corresponds to the classification rule of the Gaussian model with tied covariances!

Indeed, LDA assumes that all classes have the same within-class covariance

The Gaussian model with one covariance per class is also known as Quadratic Discriminant Analysis

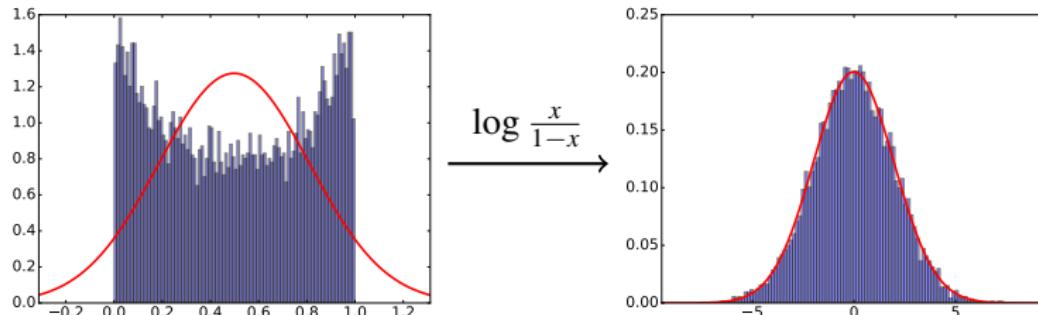
Practical considerations:

- If data is high-dimensional, PCA can simplify the estimation
- PCA also allows removing dimensions with very small variance (e.g. in the MNIST dataset, pixels that are white for all images regardless of the digit)
- Multivariate models perform better if we have enough data to reliably estimate the covariance matrices
- Naive Bayes can simplify the estimation, but may perform poorly if data are highly correlated
- Tied covariance models can capture correlations, but may perform poor when classes have very different distributions — on the other hand, if we have reason to believe that covariances should be very similar, then the model will provide a more reliable estimate

Gaussian Classifier

Practical considerations:

- If a Gaussian model is not adequate for our data we can use different distributions that are more appropriate
- Alternatively, the Gaussian model may still be effective for **transformed** data



Multiclass Gaussian Classifier

MNIST — Error rates for Gaussian classifier

Classifier	PCA (100)	PCA (50)	PCA (9)	PCA + LDA (100 → 9)
Naive Tied Gaussian	13.7%	14.4%	25.0%	12.3%
Tied Gaussian	12.3%	12.6%	23.7%	12.3%
Naive Gaussian	12.2%	12.3%	23.4%	11.4%
Gaussian	4.3%	3.6%	12.2%	10.2%

Comments:

- The best model is the unconstrained MVG — classes have significantly different between class covariances
- The Naive Bayes assumption is not good in this case (strong within-class correlations)
- PCA helps reducing the complexity — from 100 to 50 dimensions we have a significant gain. If we reduce too much we have bad results again

Comments:

- Tied model + LDA achieve the same performance as Tied model without LDA — The LDA subspace contains all the information used by the tied model
- The Naive tied model, without LDA, performs slightly worse — it ignores within-class correlations.
- Our implementation of LDA whitens the within-class covariance matrix — Tied naive model and Tied model are equivalent, since $\Sigma = I$

Modeling discrete values

We now consider a problem characterized by discrete features

For the moment, we assume we have a single categorical feature $x \in \{1 \dots m\}$.

For example, we may want to predict a cat gender from the fur color



Modeling discrete values

The categorical feature is the fur color, for example

$$x \in \{white, black, orange, gray, bi-color, calico, \dots\}$$

Also in this case, we have a set of labeled training samples

$$\mathcal{D} = \{(x_1, c_1) \dots (x_n, c_n)\}$$

Each sample x_i is simply the fur color of the i -th cat sample, and c_i is the gender label

Modeling discrete values

We also assume that samples are i.i.d. (as in the Gaussian case)

We want to compute the probabilities

$$P(X_t = x_t | C_t = c) = \pi_{c,x_t}$$

for the different hypotheses c , for the observed test sample x_t

In our example, these are the probabilities of observing fur color x_t under the different gender hypotheses $c \in \{\text{female}, \text{male}\}$ (e.g. $P(X_t = \text{white} | C_t = \text{female})$)

$\pi_c = (\pi_{c,1} \dots \pi_{c,m})$ are the model parameters for class c , with

$$\sum_{i=1}^m \pi_{c,i} = 1$$

Modeling discrete values

Again, we can adopt a frequentist approach and estimate the ML solution for $\boldsymbol{\Pi} = (\pi_1 \dots \pi_k)$ (k is again the number of classes)

We can express the likelihood for the training set as

$$\mathcal{L}(\boldsymbol{\Pi}) = \prod_{i=1}^n P(X_i = x_i | C_i = c_i)P(C_i = c_i)$$

where $c_i \in \{1 \dots k\}$ is the class of the i -th sample

Modeling discrete values

For example, if our dataset consists of⁵

(black, male), (orange, male), (black, female), (orange, male),
(white, male), (white, female), (white, male), (white, female),
(black, female), (calico, female)

the likelihood is simply the product

$$\begin{aligned}\mathcal{L}(\Pi) = & P(X_1 = \text{black}|C_1 = \text{male})P(C_1 = \text{male}) \times \\ & P(X_2 = \text{orange}|C_2 = \text{male})P(C_2 = \text{male}) \times \\ & P(X_3 = \text{black}|C_3 = \text{female})P(C_3 = \text{female}) \times \\ & \vdots \\ & P(X_{10} = \text{calico}|C_{10} = \text{female})P(C_{10} = \text{female})\end{aligned}$$

⁵We assume we have associated each color to an integer value $\{1 \dots m\}$, and that the gender labels are associated to $\{0, 1\}$

Modeling discrete values

The log-likelihood is given by

$$\begin{aligned}\ell(\boldsymbol{\Pi}) &= \sum_{i=1}^n \log P(X_i = x_i | C_i = c_i) + \xi \\ &= \sum_{i=1}^n \log \pi_{c_i, x_i} + \xi \\ &= \sum_{c=1}^k \sum_{i | c_i = c} \log \pi_{c, x_i} + \xi \\ &= \sum_{c=1}^k \ell_c(\boldsymbol{\pi}_c) + \xi\end{aligned}$$

where

$$\ell_c(\boldsymbol{\pi}_c) = \sum_{i | c_i = c} \log \pi_{c, x_i}$$

Again, the log-likelihood can be expressed as a sum of terms, each depending on a separate subset of the parameters (those of class c)

Modeling discrete values

We can thus estimate the parameters by independently optimizing $\ell_c(\pi_c)$

We have

$$\begin{aligned}\ell_c(\pi_c) &= \sum_{i|c_i=c} \log \pi_{c,x_i} \\ &= \sum_{j=1}^m N_{c,j} \log \pi_{c,j}\end{aligned}$$

where $N_{c,j}$ is the number of times we observed $x_i = j$ in the dataset for class c (i.e. the number of times the features of class c were equal to j)

In our example we would have

$$N_{female,black} = 2 \quad N_{female,orange} = 0 \quad N_{female,white} = 2 \quad N_{female,calico} = 1$$

$$N_{male,black} = 1 \quad N_{male,orange} = 2 \quad N_{male,white} = 2 \quad N_{male,calico} = 0$$

Modeling discrete values

Solving for π_{c,x_i} is a bit more complex than in the Gaussian case, since we have the constraint that $\sum_{j=1}^m \pi_{c,j} = 1$

In the following we drop subscript c , thus we look for the maximizer of

$$f(\boldsymbol{\pi}) = \sum_{j=1}^m N_j \log \pi_j$$

subject to

$$\sum_{j=1}^m \pi_j = 1$$

A solution can be obtained by means of **Lagrange multipliers**

We look for stationary points of

$$L(\boldsymbol{\pi}) = f(\boldsymbol{\pi}) - \lambda \left(\sum_{j=1}^m \pi_j - 1 \right) = \sum_{j=1}^m N_j \log \pi_j - \lambda \left(\sum_{j=1}^m \pi_j - 1 \right)$$

Modeling discrete values

We need to solve

$$\begin{cases} \frac{\partial L}{\partial \pi_i} = 0, & \forall i = 1 \dots m \\ \frac{\partial L}{\partial \lambda} = 0 \end{cases}$$

We have

$$\frac{\partial L}{\partial \pi_i} = \frac{N_i}{\pi_i} - \lambda = 0$$

$$\frac{\partial L}{\partial \lambda} = 1 - \sum_{j=1}^m \pi_j = 0$$

We can verify that $\lambda \neq 0$, so that

$$\pi_i = \frac{N_i}{\lambda}$$

Modeling discrete values

From the derivative with respect to λ we thus have

$$\sum_{j=1}^m \pi_j = \sum_{j=1}^m \frac{N_j}{\lambda} = \frac{1}{\lambda} \sum_{j=1}^m N_j = 1$$

thus

$$\lambda = \sum_{j=1}^m N_j = N$$

where $N = \sum_{j=1}^m N_j$ is the number of samples for the considered class

We finally have

$$\pi_i = \frac{N_i}{N}$$

i.e., the frequency with which we observed value i in the class

Modeling discrete values

The ML solution for each class is thus

$$\pi_{c,i}^* = \frac{N_{c,i}}{N_c}$$

We can compute the predictive distribution for x_t as

$$P(X_t = x_t | C_t = c) = \pi_{c,x_t}^*$$

In our example, we would have

$$N_{male} = 5 \quad N_{female} = 5$$

thus

$$\begin{aligned}\pi_{female,black}^* &= 0.4 & \pi_{female,orange}^* &= 0 & \pi_{female,white}^* &= 0.4 & \pi_{female,calico}^* &= 0.2 \\ \pi_{male,black}^* &= 0.2 & \pi_{male,orange}^* &= 0.4 & \pi_{male,white}^* &= 0.4 & \pi_{male,calico}^* &= 0\end{aligned}$$

Modeling discrete values

If we have more than one categorical attribute, we may model their joint probability as a categorical R.V. with values given by all possible combinations of the attributes

In practice, the number of elements would quickly become intractable

We can adopt again a naive Bayes approximation and assume that features are independent

We can obtain ML estimates for each feature, and then combine them:

$$P(X_t = \mathbf{x}_t | C_t = c) = \prod_j \pi_{c, x_t, [j]}^j$$

where j denotes the feature index

Modeling discrete values

We now consider an extended version of the problem, where features represent occurrences of events

Typical examples are topic or language modeling

We may, for example, represent a text in terms of the words that appear inside

Different topics will result in different sequences of words

Modeling all possible combinations of words is impractical, since the number of categories would grow exponentially with the number of words

Modeling discrete values

As an approximation, we may represent documents in terms of occurrences of single words

Note that such model ignores the order in which words appear — we may improve the model by considering pairs or triplets of words

Thus, we have feature vectors $\mathbf{x} = (x_{[1]} \dots x_{[m]})$, where each $x_{[i]}$ represents the number of times we observed word i in the document

Modeling discrete values

We have seen that occurrences can be modeled by multinomial distributions

Thus, for each class c , we have a set of parameters

$$\boldsymbol{\pi}_c = (\pi_{c,1} \dots \pi_{c,m})$$

that represents the probability of observing a single instance of word i

The probability for feature vector \mathbf{x} is given by the multinomial density

$$P(\mathbf{x}|C) = \frac{\left(\sum_{j=1}^m x_{[j]}\right)!}{\prod_{j=1}^m x_{[j]}!} \prod_{j=1}^m \pi_{c,j}^{x_{[j]}} \propto \prod_{j=1}^m \pi_{c,j}^{x_{[j]}}$$

Modeling discrete values

Again, we can write the likelihood of the model — as in the previous case, the likelihood factorizes over classes, thus

$$\ell(\boldsymbol{\Pi}) = \sum_c \ell_c(\boldsymbol{\pi}_c) + \xi$$

with

$$\ell_c(\boldsymbol{\pi}_c) = \sum_{i|c_i=c} \sum_{j=1}^m x_{i,[j]} \log \pi_{c,j}$$

Note that we did not write the multinomial coefficient since it's constant with respect to $\boldsymbol{\Pi}$, and thus can be absorbed in ξ .

Modeling discrete values

Also in this case we can express the log-likelihood as

$$\ell_c(\boldsymbol{\pi}_c) = \sum_{j=1}^m N_{c,j} \log \pi_{c,j}$$

where $N_{c,j}$ is the total number of occurrences of event j (word j) in the samples of class c :

$$N_{c,j} = \sum_{i|c_i=c} x_{i,[j]}$$

To solve the problem, we notice that it has exactly the same form as the one we solved for the categorical case

Modeling discrete values

Thus, the ML solution is again

$$\pi_{c,j} = \frac{N_{c,j}}{N_c}$$

where, in this case, N_c is the total number of words for class c

$$N_c = \sum_j N_{c,j} = \sum_{i|c_i=c} \sum_{j=1}^m x_{i,[j]}$$

$\pi_{c,j}$ is therefore again the relative frequency of word j in class c

Modeling discrete values

Again, we write the log-likelihood ratio for a two class problem (we use the expression of the multinomial log-probability — notice that the binomial coefficient can be simplified and thus disappears from the final expression):

$$\begin{aligned} llr(\mathbf{x}) &= \log \frac{P(\mathbf{X} = \mathbf{x} | C = h_1)}{P(\mathbf{X} = \mathbf{x} | C = h_0)} \\ &= \sum_{j=1}^m x_{[j]} \log \pi_{h_1,j} - \sum_{j=1}^m x_{[j]} \log \pi_{h_0,j} \\ &= \mathbf{x}^T \mathbf{b} \end{aligned}$$

with

$$\mathbf{b} = (\log \pi_{h_1,1} - \log \pi_{h_0,1}, \dots, \log \pi_{h_1,m} - \log \pi_{h_0,m})$$

Again, we have linear decision functions

Modeling discrete values

An example: inferring programming language

We consider the problem of automatically inferring whether a file is written in C or Python

We consider a simple model based on the occurrences of punctuation characters: {} [] () : ; . ,

We treat open and closed brackets of the same kind as a single symbol

We model scripts in terms of occurrences of the aforementioned symbols

Modeling discrete values

Let's consider the following training set

	{}	[]	()	:	;	.	,
C script 1	6	8	14	1	10	1	7
C script 2	8	10	14	0	11	1	7
C script 3	12	22	34	1	21	2	13
C script 4	4	6	10	1	6	1	4
C total	30	46	72	3	48	5	31
Py script 1	6	14	30	6	2	16	16
Py script 2	2	8	14	3	1	9	8
Py script 3	4	14	26	7	2	15	14
Py total	12	36	70	16	5	40	38

Modeling discrete values

The ML estimate for the model parameters are the symbol frequencies for each class :

	{}	[]	()	:	;	.	,
π_C	0.128	0.196	0.306	0.013	0.204	0.021	0.132
π_{Py}	0.055	0.166	0.323	0.074	0.023	0.184	0.175

To infer the class of an unknown script, we need to compute class posterior.

In this case, the task is binary so we compute log-likelihood ratios $llr(x)$, where x is the vector of occurrences of the symbols in the test script

The log-likelihood ratio can be expressed as $llr(x) = x^T b$

Modeling discrete values

Each element of \mathbf{b} is $b_i = \log \frac{\pi_{C,[i]}}{\pi_{Py,[i]}}$

	{}	[]	()	:	;	.	,
$\mathbf{b} \approx$	0.916,	0.245,	0.028,	-1.674,	2.261,	-2.079,	0.204

The model parameters already tell us what symbols are most discriminant

Observing ; is much more likely for C scripts

Observing : or . is much more likely for Python scripts

Round brackets are not very discriminant

Modeling discrete values

We consider two test scripts:

	{}	[]	()	:	;	.	,
Test script x_1 (C)	2	10	12	0	1	1	0
Test script x_2 (Py)	2	18	16	3	0	1	1

The llr for the first script is $llr(x_1) \approx 4.8$, i.e. it's much more likely that we observe the occurrences of x in C scripts

For the second script we have $llr(x_2) \approx -0.6$, i.e. it's slightly more likely that we observe the occurrences in x_2 in Python scripts

Class posterior can be computed once we choose class priors

With uniform priors, x_1 would be assigned to C class, and x_2 to Python class

Modeling discrete values

Practical considerations:

- Rare words can cause problems: if a word does not appear in a topic we will estimate a probability $\pi_{c,j} = 0$. Any test sample that contains the word will have 0 probability of being from class c
- We can mitigate the issue introducing **pseudo-counts**, i.e. assuming that each topic contains a sample where all words appear a (fixed) number of times
- In practice, we can add a fixed value to the class occurrences N_c before computing the ML solution
- This corresponds to a Maximum-a-posterior estimate using a Dirichlet prior distribution (we will briefly discuss Maximum a posteriori estimation later)

Practical considerations:

- Bayesian treatment of the hyperparameters is a more appropriate way to deal with this issue (but we will not analyze in details the Bayesian models)
- We can consider pairs of words, triplets of words and so on if we want to partially account for correlations
- We can also combine different models (categorical, multinomial, Gaussian, ...) through a naive Bayes assumption

Modeling discrete values

Some additional comments on the discrete models for categorical events:

- We can model a dataset of categorical samples as n independent categorical R.V.s $X_i \in \{1 \dots m\}$.
- Each variable represents a token.
- The distribution is described by a vector of probabilities π that allow computing $P(X = j) = \pi_j$

Modeling discrete values

Some additional comments on the discrete models for categorical events:

- Alternatively, we can model the dataset in terms of occurrences of events.
- We have a random vector $\mathbf{Y} = (Y_1 \dots Y_m)$ whose components are R.V. Y_i corresponding to the number of occurrences of event i in the dataset.
- Again, the distribution is described by a vector of probabilities π that represent probabilities of single events

The two models are related by

$$Y_j = \sum_{i=1}^n \mathbb{I}[X_i = j]$$

Modeling discrete values

As we have seen, in both cases the ML solution for π corresponds to the relative frequencies of occurrences. Indeed, we can go further and show that the two models are equivalent

Considering only samples of a single class, in the first case the log-likelihood for a given class is given by

$$\ell_X(\boldsymbol{\pi}) = \sum_{i=1}^n \log \pi_{x_i} = \sum_{j=1}^m N_j \log \pi_j$$

whereas in the second case

$$\ell_Y(\boldsymbol{\pi}) = \log \frac{\left(\sum_{j=1}^m y_{[j]}\right)!}{\prod_{i=1}^m y_{[i]}!} + \sum_{j=1}^m y_j \log \pi_j = \xi + \sum_{j=1}^m N_j \log \pi_j$$

Modeling discrete values

The corresponding likelihoods $\mathcal{L}_X(\boldsymbol{\pi})$ and $\mathcal{L}_Y(\boldsymbol{\pi})$ are proportional:

$$\mathcal{L}_X(\boldsymbol{\pi}) \propto \mathcal{L}_Y(\boldsymbol{\pi})$$

Therefore:

- Maximum likelihood estimates will be the same
- Bayesian posterior probabilities for model parameters will be the same (again, we will not go into details of Bayesian models)
- Inference will be the same

Modeling discrete values

The first point is straightforward

Regarding the last point, we can observe that, for a given test sample, the class conditional likelihoods are also proportional

$$f_{X|C}(\mathbf{x}_t|c) = \zeta(\mathbf{y}_t) \cdot f_{Y|C}(\mathbf{y}_t|c)$$

Thus, binary log-likelihood ratios are equal

$$\log \frac{f_{X|C}(\mathbf{x}_t|h_1)}{f_{X|C}(\mathbf{x}_t|h_0)} = \log \frac{f_{Y|C}(\mathbf{y}_t|h_1)}{f_{Y|C}(\mathbf{y}_t|h_0)}$$

and similarly for class posteriors:

$$P(C_t = c | \mathbf{X} = \mathbf{x}) = \frac{f_{X|C}(\mathbf{x}_t|c)P(c)}{\sum_{c'} f_{X|C}(\mathbf{x}_t|c')P(c')} = \frac{f_{Y|C}(\mathbf{y}_t|c)P(c)}{\sum_{c'} f_{Y|C}(\mathbf{y}_t|c')P(c')} = P(C_t = c | \mathbf{Y} = \mathbf{y})$$

Logistic Regression

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Discriminative Linear Models – Logistic Regression

Logistic regression, despite its name, is a discriminative approach for classification

Rather than modeling the distribution of observed samples $X|C$, we directly model the class posterior distribution $C|X$

We need to define a model for the class posterior distribution
 $P(C = c|X = \mathbf{x})$

Discriminative Linear Models – Logistic Regression

For a 2-class problem, we have seen that the Gaussian model with tied covariances provides log-likelihood ratios that are linear functions of our data

$$l(\mathbf{x}) = \log \frac{f_{X|C}(\mathbf{x}|h_1)}{f_{X|C}(\mathbf{x}|h_0)} = \mathbf{w}^T \mathbf{x} + c$$

and the posterior log-likelihood ratio is

$$\log \frac{P(C = h_1 | \mathbf{x})}{P(C = h_0 | \mathbf{x})} = \log \frac{f_{X|C}(\mathbf{x}|h_1)}{f_{X|C}(\mathbf{x}|h_0)} + \log \frac{\pi}{1 - \pi} = \mathbf{w}^T \mathbf{x} + b$$

The prior information has been absorbed in the bias term b .

Logistic Regression

Given w and b , we can compute the expression for the posterior class probability as

$$\begin{aligned} P(C = h_1 | \mathbf{x}, \mathbf{w}, b) &= e^{(\mathbf{w}^T \mathbf{x} + b)} P(C = h_0 | \mathbf{x}, \mathbf{w}, b) \\ &= e^{(\mathbf{w}^T \mathbf{x} + b)} (1 - P(C = h_1 | \mathbf{x}, \mathbf{w}, b)) \end{aligned}$$

Solving for $P(C = h_1 | \mathbf{x}, \mathbf{w}, b)$ we obtain

$$P(C = h_1 | \mathbf{x}, \mathbf{w}, b) = \frac{e^{(\mathbf{w}^T \mathbf{x} + b)}}{1 + e^{\mathbf{w}^T \mathbf{x} + b}} = \frac{1}{1 + e^{-(\mathbf{w}^T \mathbf{x} + b)}} = \sigma(\mathbf{w}^T \mathbf{x} + b)$$

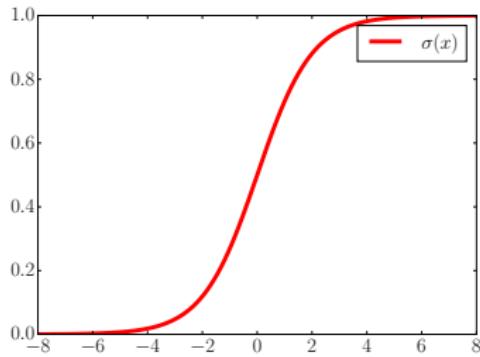
where

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

is called *sigmoid* function (or *logistic* function)

Logistic Regression

Sigmoid function:



Some properties of $\sigma(x)$ that will come useful later

- $1 - \sigma(x) = \sigma(-x)$
- $\frac{d\sigma(x)}{dx} = \sigma(x) (1 - \sigma(x))$

Logistic Regression

The equation $P(C = h_1 | \mathbf{x}, \mathbf{w}, b) = \sigma(\mathbf{w}^T \mathbf{x} + b)$ provides a model that allows computing the posterior probabilities for h_1 and h_0

The model assumes that decision rules are linear surfaces (hyperplanes) orthogonal to \mathbf{w} . The model parameters are (\mathbf{w}, b)

If we knew (\mathbf{w}, b) then we could compute the predictive distribution for the class labels $P(C = h_1 | \mathbf{x}, \mathbf{w}, b)$

We have seen how we can compute estimates for (\mathbf{w}, b) from a generative Gaussian model

However, in the following, we ignore the generative model for X and concentrate on the form of the class posterior probabilities

Again, we will follow a frequentist approach, i.e. compute an estimate for \mathbf{w} and b from a set of training samples

Logistic Regression

We assume we have a labeled dataset

$$\mathcal{D} = [(\mathbf{x}_1, c_1), \dots (\mathbf{x}_n, c_n)]$$

We also assume classes are independently distributed as

$$C_i | \mathbf{x}_i, \mathbf{w}, b \sim C | \mathbf{x}_i, \mathbf{w}, b$$

The class-posterior model allows expressing the likelihood for the observed labels as

$$P(C_1 = c_1, \dots, C_n = c_n | \mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{w}, b) = \prod_{i=1}^n P(C_i = c_i | \mathbf{x}_i, \mathbf{w}, b)$$

We can apply a ML approach to estimate the model parameters that best describe the observed labels $(c_1 \dots c_n)$

Logistic Regression

Rather than estimating w from class-conditional likelihoods, we estimate the value of w that maximizes the likelihood of our training labels.

We assume that the label for class h_1 is 1, and the label for class h_0 is 0

Let

$$y_i = P(C_i = 1 | \mathbf{x}_i, \mathbf{w}, b) = \sigma(\mathbf{w}^T \mathbf{x}_i + b)$$

It follows that

$$P(C_i = 0 | \mathbf{x}_i, \mathbf{w}, b) = 1 - y_i = 1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b) = \sigma(-\mathbf{w}^T \mathbf{x} - b)$$

Logistic Regression

The distribution for $C_i | \mathbf{x}_i, \mathbf{w}, b$ is a Bernoulli distribution

$$C_i | \mathbf{x}_i, \mathbf{w}, b \sim \text{Ber}(\sigma(\mathbf{w}^T \mathbf{x}_i + b)) = \text{Ber}(y_i)$$

The likelihood for our label set is

$$\begin{aligned}\mathcal{L}(\mathbf{w}, b) &= P(C_1 = c_1, \dots, C_n = c_n | \mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{w}, b) \\ &= \prod_{i=1}^n P(C_i = c_i | \mathbf{x}_i, \mathbf{w}, b) \\ &= \prod_i y_i^{c_i} (1 - y_i)^{(1 - c_i)}\end{aligned}$$

Again, it's more practical to work with the log-likelihood

$$\ell(\mathbf{w}, b) = \log \mathcal{L}(\mathbf{w}, b) = \sum_{i=1}^n [c_i \log y_i + (1 - c_i) \log(1 - y_i)]$$

Logistic Regression

Our goal is the maximization of ℓ . We thus seek w^*, b^* that maximize $\ell(w, b)$:

$$w^*, b^* = \arg \max_{w,b} \ell(w, b) = \arg \max_{w,b} \sum_{i=1}^n [c_i \log y_i + (1 - c_i) \log(1 - y_i)]$$

The ML solution is also the solution that minimizes the average **cross-entropy** between the distribution of observed and predicted labels

Rather than maximizing $\ell(w, b)$, we can **minimize**

$$J(w, b) = -\ell(w, b) = \sum_{i=1}^n -[c_i \log y_i + (1 - c_i) \log(1 - y_i)]$$

Logistic Regression

The expression

$$H(c_i, y_i) = -[c_i \log y_i + (1 - c_i) \log (1 - y_i)]$$

represents the binary *cross-entropy* between the distribution of observed and predicted labels for the i -th sample

More in general, let P and Q be two distributions over the same domain

The cross-entropy between the two distributions is defined as

$$H(P, Q) = -\mathbb{E}_{P(x)} [\log Q(x)]$$

For discrete distributions, this can be expressed as

$$H(P, Q) = -\sum_{x \in S} P(x) \log Q(x)$$

Logistic Regression

In our case, P is the empirical distribution of class labels, from the point of view of an observer \mathcal{E} who knows the actual label:

$$P(C_i = 1 | X_i = \mathbf{x}_i, \mathcal{E}) = \begin{cases} 1 & \text{if } c_i = 1 \\ 0 & \text{if } c_i = 0 \end{cases}$$

$$P(C_i = 0 | X_i = \mathbf{x}_i, \mathcal{E}) = \begin{cases} 0 & \text{if } c_i = 1 \\ 1 & \text{if } c_i = 0 \end{cases}$$

or, equivalently

$$P(C_i = 1 | X_i = \mathbf{x}_i, \mathcal{E}) = c_i, \quad P(C_i = 0 | X_i = \mathbf{x}_i, \mathcal{E}) = 1 - c_i$$

i.e., a Bernoulli distribution with parameter c_i

Logistic Regression

Distribution Q is the distribution for the predicted labels according to our recognizer \mathcal{R}

$$Q(c) = P(C_i = c | X_i = \mathbf{x}_i, \mathcal{R}(\mathbf{w}, b))$$

i.e.

$$Q(1) = P(C_i = 1 | X_i = \mathbf{x}_i, \mathcal{R}(\mathbf{w}, b)) = y_i = \sigma(\mathbf{w}^T \mathbf{x}_i + b)$$

$$Q(0) = P(C_i = 0 | X_i = \mathbf{x}_i, \mathcal{R}(\mathbf{w}, b)) = 1 - y_i = 1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b)$$

Logistic regression looks for the minimizer of the average cross-entropy between the distributions for the training set labels of an evaluator \mathcal{E} who knows the real label and the distributions for the training set labels as predicted by the model $\mathcal{R}(\mathbf{w}, b)$ itself

The cross-entropy is a measure of goodness of the predictions, and the evaluation is performed over the training data itself

Logistic Regression

The cross-entropy, as a function of Q , is minimized when $Q = P$

The cross-entropy can also be interpreted as a measure of the difference between P and Q

In our case, it measures how different is the predicted distribution $\text{Ber}(y_i)$ from the empirical label distribution $\text{Ber}(c_i)$ (the distribution of the evaluator \mathcal{E})

Minimization of the average cross-entropy means we are looking for a label distribution that is (on average) as similar as possible to the empirical one

Alternatively, as we have seen, we can regard the process as maximization of the likelihood for the observed labels

Logistic Regression

Another interesting interpretation of the logistic regression objective can be obtained by rewriting the cross-entropy in terms of $z_i = 2c_i - 1$, i.e.

The terms z_i still represent class labels, however for samples of class h_1 we have $z_i = 1$, whereas for samples of class h_0 we have $z_i = -1$:

$$z_i = \begin{cases} 1 & \text{if } c_i = 1 \\ -1 & \text{if } c_i = 0 \end{cases}$$

Logistic Regression

The objective function that we want to minimize corresponds to the sum of n terms

$$J(\mathbf{w}, b) = \sum_i H(c_i, y_i)$$

where

$$H(c_i, y_i) = -[c_i \log y_i + (1 - c_i) \log(1 - y_i)]$$

Note that $H(c_i, y_i)$ is a function of c_i , but also of \mathbf{w} , b and \mathbf{x}_i , since

$$y_i = \sigma(\mathbf{w}^T \mathbf{x}_i + b)$$

Logistic Regression

Let $s_i = \mathbf{w}^T \mathbf{x}_i + b$. In terms of z_i we can rewrite H as

$$\begin{aligned} H(c_i, y_i) &= -[c_i \log y_i + (1 - c_i) \log(1 - y_i)] \\ &= \begin{cases} -\log \sigma(s_i) & \text{if } c_i = 1 \ (z_i = 1) \\ -\log(1 - \sigma(s_i)) = -\log \sigma(-s_i) & \text{if } c_i = 0 \ (z_i = -1) \end{cases} \\ &= -\log \sigma(z_i s_i) \\ &= -\log \sigma(z_i(\mathbf{w}^T \mathbf{x}_i + b)) \\ &= \log \left(1 + e^{-z_i(\mathbf{w}^T \mathbf{x}_i + b)} \right) \end{aligned}$$

Logistic Regression

The objective function can thus be rewritten as

$$\begin{aligned} J(\mathbf{w}, b) &= \sum_{i=1}^n H(c_i, y_i) \\ &= \sum_{i=1}^n \log \left(1 + e^{-z_i(\mathbf{w}^T \mathbf{x}_i + b)} \right) \\ &= \sum_{i=1}^n l(z_i(\mathbf{w}^T \mathbf{x}_i + b)) \end{aligned}$$

where

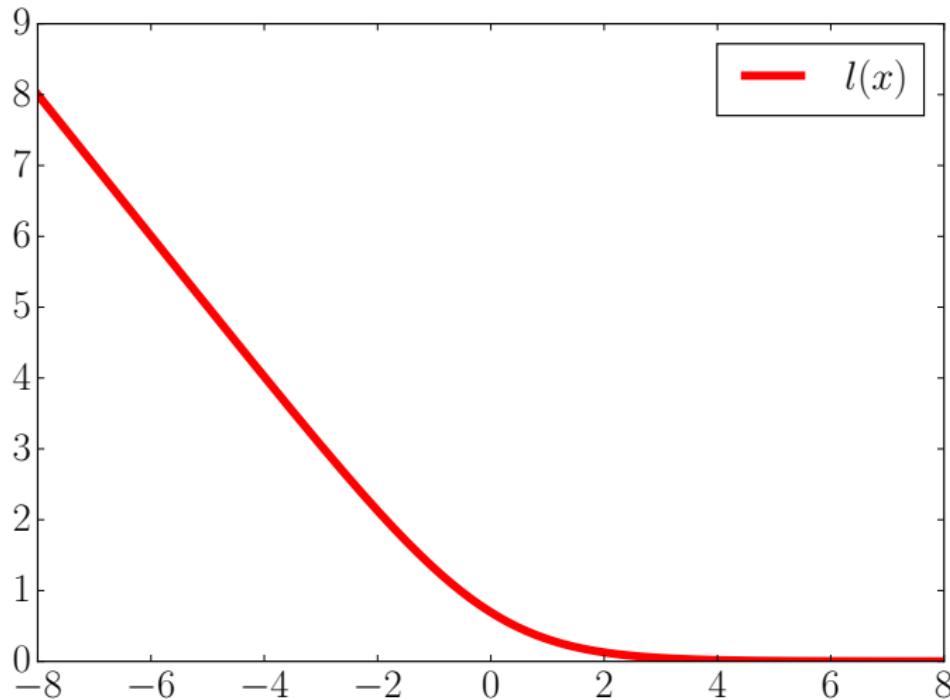
$$l(x) = \log \left(1 + e^{-x} \right)$$

is the **logistic loss** function.

Our goal is to find the **minimizer** of $J(\mathbf{w}, b)$

Logistic Regression

Logistic loss



Logistic Regression

We can interpret the function as the **cost** of the prediction made with model (\mathbf{w}, b) for each sample

Remember that posterior log-likelihood ratio is

$$\log \frac{P(h_1|\mathbf{x}_i)}{P(h_0|\mathbf{x}_i)} = \mathbf{w}^T \mathbf{x}_i + b = s_i$$

Decision rules take the form $s_i \leq t$

Since $s_i = \mathbf{w}^T \mathbf{x}_i + b$, decision rules are linear hyperplane orthogonal to the vector \mathbf{w}

s_i is related to the distance of the sample \mathbf{x}_i from the separating surface

Logistic Regression

When s_i is positive, our classifier is favoring class h_1 , whereas negative s_i means we are classifying the sample as belonging to class h_0 .

The cost we pay for each sample is $l(z_i s_i)$

- The prediction and the actual class agree: $z_i = 1, s_i > 0$ or $z_i = -1, s_i < 0$. Then $z_i s_i > 0$, and we pay a low cost. The cost becomes exponentially smaller (asymptotically) as the absolute value of s_i increases (we move away from the separation surface)
- The prediction and the actual class disagree: $z_i = 1, s_i < 0$ or $z_i = -1, s_i > 0$. Then $z_i s_i < 0$, and we pay a cost that increases (asymptotically) linearly with s_i

Logistic Regression

We can thus interpret the logistic regression objective as a measure of an **empirical risk**¹. Our goal is minimizing the empirical risk

More in general, empirical risk minimization is a framework for the estimation of classification models which aims at minimizing an **empirical risk function** over our training data

Generalized risk minimization problem: minimize the risk $R(\theta)$

$$R(\theta) = \sum_i l(w, x_i, z_i)$$

where l is called loss (or cost) function, and θ are the parameters of the classification model, e.g. $\theta = (w, b)$ in our case.

¹Empirical because it's computed on the observed samples

Logistic Regression

Logistic regression solutions cannot be computed in closed form

We will resort to numerical solvers

A numerical solver iteratively looks for the minimizer of a function

We will use the L-BFGS algorithm

The algorithm requires a function that computes the loss and its gradient with respect to w and b

In the laboratory we will see how to implement the minimization

Logistic Regression

If classes are linearly separable, the logistic regression solution is not defined

Linearly separable classes: there exist w and b such that all training samples lie on the correct side of the corresponding separation surface ($z_i > 0 \iff s_i > 0$)

In this case, we can make the values of s_i arbitrarily high by simply increasing the norm of w (and changing accordingly the value of b)

As we increase $\|w\|$, the loss becomes lower, thus we are decreasing the objective function

The function does not have a minimum, but has an infimum
 $\inf J(w, b) = 0$, corresponding to $\|w\| \rightarrow \infty$

Logistic Regression

To make the problem solvable again, we can look for solutions with small norm by introducing a norm penalty to the objective function

The penalty is called **regularization** term

The objective function that we minimize is

$$\tilde{R}(\mathbf{w}, b) = \frac{\tilde{\lambda}}{2} \|\mathbf{w}\|^2 + \sum_{i=1}^n \log \left(1 + e^{-z_i(\mathbf{w}^T \mathbf{x}_i + b)} \right)$$

where $\tilde{\lambda}$ is a hyper-parameter that allows specifying the relative weight of the regularization term

Alternatively, we look for the minimizer of

$$R(\mathbf{w}, b) = \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{n} \sum_{i=1}^n \log \left(1 + e^{-z_i(\mathbf{w}^T \mathbf{x}_i + b)} \right)$$

where the risk is *averaged* over all samples, and λ is the regularization coefficient.

Logistic Regression

λ is a hyper-parameter, and should be selected as to optimize the performance of the classifier

Note that λ cannot be computed by minimizing R with respect to λ , as we would obtain the trivial solution $\lambda = 0$

The selection of good values for λ should thus be based on other approaches, such as **cross-validation**

The model is called **regularized** Logistic Regression, and is an example of a **regularized risk minimization** problem

$$R(\mathbf{w}, b) = \Omega(\mathbf{w}, b) + \frac{1}{n} \sum_{i=1}^n l(\mathbf{x}_i, z_i, \mathbf{w}, b)$$

Logistic Regression

The regularization term Ω (in our case $\frac{\lambda}{2} \|\mathbf{w}\|^2$) can be interpreted as a term that favors simpler solutions (we will see explicitly why small norm of \mathbf{w} can be interpreted as a simpler solution when discussing Support Vector Machines)

Regularization allows reducing the risk of over-fitting the training data

Of course, if λ is too large, we will obtain a solution that has small norm, but is not able to well separate the classes

On the other hand, if λ is too small, we will get a solution that has good separation on the training set, but may have poor classification accuracy for unseen data (i.e. poor **generalization**)

Logistic Regression

Some considerations:

- The non-regularized model is invariant to linear transformations of the feature vectors.
- The regularized version of the model, on the other hand, is not invariant.
- It is therefore useful, in some cases, to pre-process data so that dynamic ranges of different features are similar
- Cross-validation can help in identifying good pre-processing strategies

Logistic Regression

Common **preprocessing strategies** that may be worth trying:

- Center the data (either using the training set, or a weighted mean — e.g. the average of class means): $x'_i = x_i - \mu$
- Standardize variances (e.g. divide each feature by its own standard deviation computed over the training set): $x'_{i,[j]} = x_{i,[j]} / \sigma_{[j]}$
- Whiten the covariance matrix (i.e. normalize variances while making features uncorrelated): $x'_i = Ax_i$, where $A = \Sigma^{\frac{1}{2}}$ and Σ is the training set covariance (a variation consists in replacing Σ with the within-class covariance)
- L2 (or length) normalization: $x'_i = \frac{x_i}{\|x_i\|}$ (often after centering and whitening)

Logistic Regression

Some considerations:

- The Logistic Regression score can be interpreted as a posterior log-likelihood ratio
- It reflects the empirical class prior of the training data
- We can simulate different empirical priors π_T using a prior-weighted version of the model

$$R(w) = \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{\pi_T}{n_T} \sum_{i|z_i=1} l(z_i s_i) + \frac{1 - \pi_T}{n_F} \sum_{i|z_i=-1} l(z_i s_i)$$

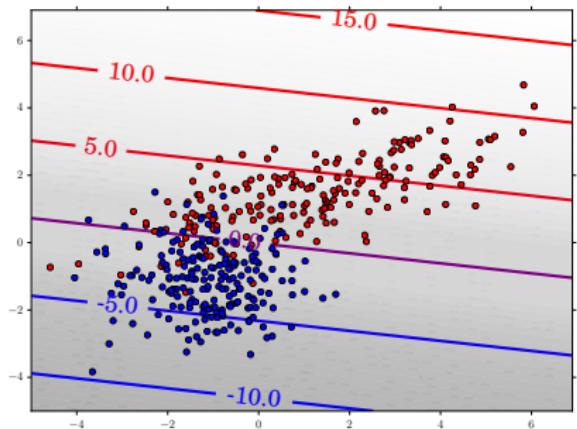
Logistic Regression

Some considerations:

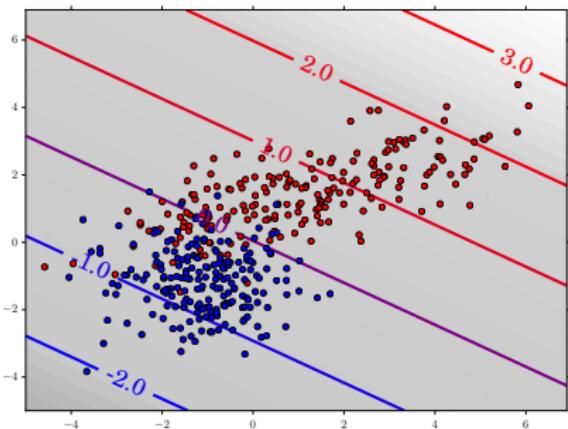
- If our application is characterized by the same effective prior π_T , the optimal decision should correspond to $w^T x + b \leq 0$
- In theory, we can also recover log-likelihood ratios by subtracting from the score s the empirical prior log-odds of the training set $\log \frac{n_T}{n_F}$, where n_T is the number of samples of class $h_1(c_i = 1, z_i = +1)$ and n_F is the number of samples of class $h_0(c_i = 0, z_i = -1)$
- In some cases, especially if the dimensionality of the space is large, the score s may not provide the correct probabilistic interpretation, and optimal decisions may require either recalibration of the scores or selection of an optimal threshold based on a validation set

Logistic Regression

$$\lambda = 0$$



$$\lambda = 1$$



Logistic Regression

We test the model on MNIST digit pairs (e.g. 0 vs 1, 0 vs 2, ...)

MNIST — Average Pairwise EER for Logistic Regression

DimRed	$\lambda = 0$	$\lambda = 0.00001$	$\lambda = 0.001$	$\lambda = 0.1$	Tied Gau
RAW [768]	1.7%	1.4%	1.2%	2.0%	—
PCA [50]	1.4%	1.4%	1.4%	2.1%	1.7%
PCA [100]	1.3%	1.2%	1.2%	2.0%	1.5%

LogReg obtains better performance than the Gaussian model

Regularization is important, especially when we do not reduce the dimensionality (we have more parameters to estimate, so over-fitting is more severe)

If we regularize too much the model performs poorly again

Multiclass Logistic Regression

We now consider a problem with K classes, labeled from 1 to K

To extend the Logistic Regression model to multiclass tasks we start again from the form of the posterior likelihood ratios of the Linear Gaussian classifier with uniform priors

$$\log \frac{P(C=j|\mathbf{x})}{P(C=r|\mathbf{x})} = (\mathbf{w}_j - \mathbf{w}_r)^T \mathbf{x} + (b_j - b_r)$$

corresponding to pair-wise linear classification surfaces between class j and a reference class r

Notice that we over-parametrized the model by introducing an extra set of parameters (\mathbf{w}_r, b_r) , so that we do not have to explicitly enforce $\log \frac{P(C=r|\mathbf{x})}{P(C=r|\mathbf{x})} = 0$

Multiclass Logistic Regression

It follows that, for all classes $j \in 1, \dots, K$,

$$P(C = j | \mathbf{x}) = P(C = r | \mathbf{x}) e^{(\mathbf{w}_j - \mathbf{w}_r)^T \mathbf{x} + (b_j - b_r)}$$

Remembering that

$$\sum_{j=1}^K P(C = j | \mathbf{x}) = 1$$

We have

$$P(C = r | \mathbf{x}) = 1 - \sum_{j \neq r} P(C = j | \mathbf{x}) = 1 - \sum_{j \neq r} P(C = r | \mathbf{x}) e^{(\mathbf{w}_j - \mathbf{w}_r)^T \mathbf{x} + (b_j - b_r)}$$

Multiclass Logistic Regression

Therefore

$$\begin{aligned} P(C = r | \mathbf{x}) &= \frac{1}{1 + \sum_{j \neq r} e^{(\mathbf{w}_j - \mathbf{w}_r)^T \mathbf{x} + (b_j - b_r)}} \\ &= \frac{1}{\sum_j e^{(\mathbf{w}_j - \mathbf{w}_r)^T \mathbf{x} + (b_j - b_r)}} \\ &= \frac{e^{\mathbf{w}_r^T \mathbf{x} + b_r}}{\sum_j e^{\mathbf{w}_j^T \mathbf{x} + b_j}} \end{aligned}$$

Note that the model is completely specified (and actually over-parametrized) once we know the K vectors \mathbf{w}_k and the terms b_k . Repeating the operations for all classes we obtain

$$P(C = k | \mathbf{x}) = \frac{e^{\mathbf{w}_k^T \mathbf{x} + b_k}}{\sum_j e^{\mathbf{w}_j^T \mathbf{x} + b_j}}$$

Function $f_i(s) = \frac{e^{s_i}}{\sum_j e^{s_j}}$ is called **softmax**

Multiclass Logistic Regression

Given the model parameters

$$\mathbf{W} = [\mathbf{w}_1 \quad \dots \quad \mathbf{w}_K], \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_K \end{bmatrix}$$

the logistic regression model allows computing the probability of each class

$$P(C = k | \mathbf{W}, \mathbf{b}, \mathbf{x}) = \frac{e^{\mathbf{w}_k^T \mathbf{x} + b_k}}{\sum_j e^{\mathbf{w}_j^T \mathbf{x} + b_j}}$$

If we consider sample \mathbf{x}_i , its class posterior distribution is thus a **categorical** distribution

$$C_i | \mathbf{W}, \mathbf{b}, X_i = \mathbf{x}_i \sim \text{Cat}(\mathbf{y}_i)$$

where

$$\mathbf{y}_{ik} = \frac{e^{\mathbf{w}_k^T \mathbf{x}_i + b_k}}{\sum_j e^{\mathbf{w}_j^T \mathbf{x}_i + b_j}}$$

Multiclass Logistic Regression

As for the binary case, we can express the log-likelihood for the training class labels as

$$\ell(\mathbf{W}, \mathbf{b}) = \sum_{i=1}^n \log P(C_i = c_i | X_i = \mathbf{x}_i, \mathbf{W}, \mathbf{b})$$

Remember that the categorical density $P(C_i = c_i | X_i = \mathbf{x}_i, \mathbf{W}, \mathbf{b})$ can be expressed using a 1-of-K encoding, as

$$\log P(C_i = c_i | X_i = \mathbf{x}_i, \mathbf{W}, \mathbf{b}) = \log P(C_i = c_i | \mathbf{y}_i) = \sum_{k=1}^K z_{ik} \log y_{ik}$$

where \mathbf{z}_i is a vector that has all component equal to 0, except for the index c_i which is equal to 1

$$\mathbf{z}_i = [0 \dots 0, 1, 0 \dots 0] , \quad z_{ik} = \begin{cases} 1 & \text{if } c_i = k \\ 0 & \text{otherwise} \end{cases}$$

Multiclass Logistic Regression

The log-likelihood can thus be expressed as

$$\ell(\mathbf{W}, \mathbf{b}) = \sum_{i=1}^n \sum_{k=1}^K z_{ik} \log y_{ik}$$

The terms y_{ik} are, again,

$$y_{ik} = \frac{e^{\mathbf{w}_k^T \mathbf{x}_i + b_k}}{\sum_j e^{\mathbf{w}_j^T \mathbf{x}_i + b_j}}$$

and represent the distribution for the class labels according to the Logistic Regression model

Multiclass Logistic Regression

As for the binary case, the expression

$$H(\mathbf{z}_i, \mathbf{y}_i) = - \sum_{k=1}^K z_{ik} \log y_{ik}$$

represents the (multiclass) cross-entropy between the observed and predicted label distributions for sample x_i

As for the binary case, we estimate \mathbf{W} and \mathbf{b} as to maximize the likelihood for the training labels

The ML solution is again the solution that minimizes the (average) cross-entropy:

$$\arg \max_{\mathbf{W}, \mathbf{b}} \ell(\mathbf{W}, \mathbf{b}) = \arg \max_{\mathbf{W}, \mathbf{b}} \left[- \sum_{i=1}^n H(\mathbf{z}_i, \mathbf{y}_i) \right] = \arg \min_{\mathbf{W}, \mathbf{b}} \sum_{i=1}^n H(\mathbf{z}_i, \mathbf{y}_i)$$

Multiclass Logistic Regression

Compared to the binary case, the model is over-parametrized (i.e., we can add a constant vector to all terms w_i without changing the model)

In particular, for a 2-class problem, if we subtract w_2 from both w_1 and w_2 , we recover exactly the binary logistic regression objective.

Multiclass Logistic Regression

Finally, as for the binary class, we can cast the problem as a minimization of a loss function

We rewrite the objective in terms of class labels c as

$$\begin{aligned} J(\mathbf{W}, \mathbf{b}) &= - \sum_{i=1}^n \sum_{k=1}^K z_{ik} \log y_{ik} \\ &= - \sum_{i=1}^n \log \frac{e^{\mathbf{w}_{c_i}^T \mathbf{x}_i}}{\sum_{c'=1}^K e^{\mathbf{w}_{c'}^T \mathbf{x}}} \\ &= \sum_{i=1}^n \left[\log \left(\sum_{c'=1}^K e^{\mathbf{w}_{c'}^T \mathbf{x}} \right) - \mathbf{w}_{c_i}^T \mathbf{x}_i \right] \\ &= \sum_{i=1}^n l(\mathbf{x}_i, c_i, \mathbf{W}, \mathbf{b}) \end{aligned}$$

l is also called softmax loss

Multiclass Logistic Regression

Again, we can add a regularization term to reduce over-fitting.
We thus look for the minimizer of

$$R(\mathbf{W}, \mathbf{b}) = \Omega(\mathbf{W}) + \frac{1}{n} J(\mathbf{W}, \mathbf{b})$$

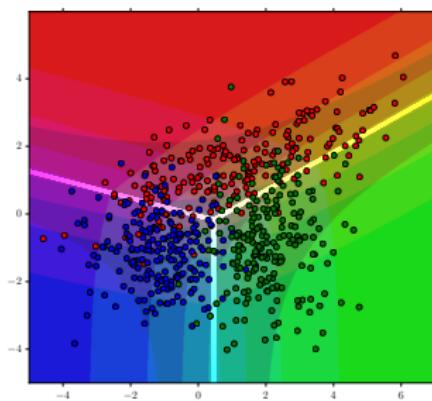
Different regularizers can be used, for example

$$\Omega(\mathbf{w}_1, \dots, \mathbf{w}_N) = \frac{1}{2} \sum_i \|\mathbf{w}_i\|^2$$

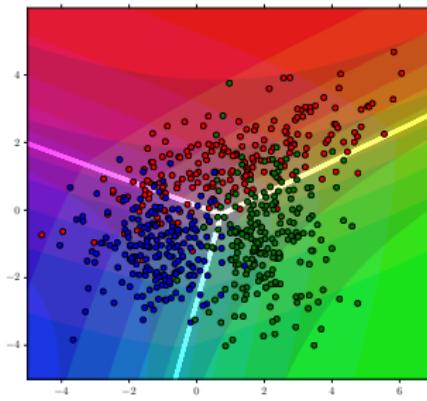
Again, we can replace the average cross-entropy with prior-weighted average cross entropy to account for priors that are different from the empirical training set prior

Multiclass Logistic Regression

Logistic Regression



Gaussian



Multiclass Logistic Regression

MNIST — Error rates for Logistic Regression

DimRed	$\lambda = 0$	$\lambda = 0.00001$	$\lambda = 0.001$	$\lambda = 0.1$	Tied Gau
RAW [768]	8.0%	7.4%	7.9%	12.9%	—
PCA [50]	8.8%	8.8%	8.9%	13.3%	12.6 %
PCA [100]	7.8%	7.8%	8.2%	12.9%	12.3%
PCA+LDA [9]	10.9%	10.9%	11.0%	12.4%	12.3 %

The multiclass logistic regression performs better than the Gaussian model — indeed, the Gaussian assumption is not very accurate for the features we are considering. LogReg only assumes linear separation, but does not impose a distribution over the features

Again, regularization is important, especially when the feature space is large

Multiclass Logistic Regression

Linear logistic regression on MNIST performs better than our Tied-Covariance Gaussian classifier, however it's far worse than our non-linear Gaussian classifier

Remember that, for binary LR, we assumed linear separation surfaces

$$\log \frac{P(C = h_1 | \mathbf{x})}{P(C = h_0 | \mathbf{x})} = \mathbf{w}^T \mathbf{x} + b$$

which has the same form as the Gaussian classifier with tied covariances

For Gaussian classifier with non-tied covariances we have

$$\log \frac{P(C = h_1 | \mathbf{x})}{P(C = h_0 | \mathbf{x})} = \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c = s(\mathbf{x}, \mathbf{A}, \mathbf{b}, c)$$

Multiclass Logistic Regression

The expression

$$s(\mathbf{x}, \mathbf{A}, \mathbf{b}, c) = \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c$$

is quadratic in \mathbf{x} , however it is **linear** in \mathbf{A} and \mathbf{b}

Indeed, we can rewrite $s(\mathbf{x}, \mathbf{A}, \mathbf{b}, c)$ as

$$s(\mathbf{x}, \mathbf{A}, \mathbf{b}, c) = \langle \mathbf{x} \mathbf{x}^T, \mathbf{A} \rangle + \mathbf{b}^T \mathbf{x} + c$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product

$$\langle \mathbf{A}, \mathbf{B} \rangle = \sum_i \sum_j \mathbf{A}_{ij} \mathbf{B}_{ij}$$

We can further express $\langle \mathbf{A}, \mathbf{x} \mathbf{x}^T \rangle$ as

$$\langle \mathbf{A}, \mathbf{x} \mathbf{x}^T \rangle = \text{vec}(\mathbf{x} \mathbf{x}^T)^T \text{vec}(\mathbf{A})$$

$\text{vec}(\mathbf{M})$ is the operator that stacks the columns of matrix \mathbf{M}

Multiclass Logistic Regression

If we define

$$\phi(\mathbf{x}) = \begin{bmatrix} \text{vec}(\mathbf{x}\mathbf{x}^T) \\ \mathbf{x} \end{bmatrix}$$

and

$$\mathbf{w} = \begin{bmatrix} \text{vec}(\mathbf{A}) \\ \mathbf{b} \end{bmatrix}$$

then the posterior log-likelihood ratio can be expressed as

$$s(\mathbf{x}, \mathbf{w}, c) = \mathbf{w}^T \phi(\mathbf{x}) + c$$

We can thus train a LR model using feature vectors $\phi(\mathbf{x})$ rather than \mathbf{x}

We will obtain a model that has linear separation surface in the space defined by the mapping ϕ

This space is also called **expanded feature space**

Multiclass Logistic Regression

The LR model (both binary and multiclass) allows computing linear separation rules for the transformed features $\phi(\mathbf{x})$

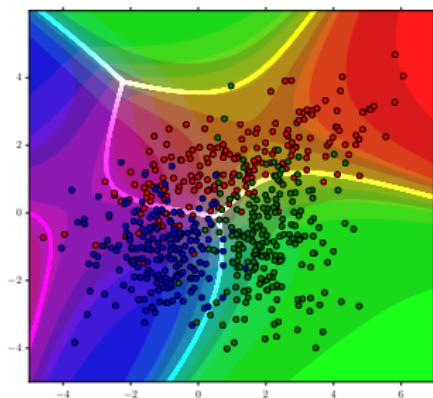
Since expressions $\mathbf{w}^T \phi(\mathbf{x}) + c$ correspond to quadratic forms in the original feature space, we are actually estimating **quadratic separation surfaces** in the original space

In general, we can consider a transformation $\phi(\mathbf{x})$ of our feature space such that our classes are (approximately) linearly separable in the expanded feature space

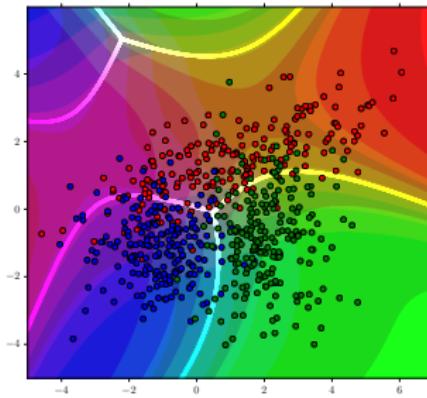
We have to pay attention that the dimensionality of the expanded feature space can grow very quickly — For example, polynomial expansions of degree d result in a feature space of dimensionality $O(M^d)$, where M is the dimensionality of \mathbf{x} .

Multiclass Logistic Regression

Logistic Regression



Gaussian



Multiclass Logistic Regression

MNIST — Average pairwise EER for LR with quadratic feature expansion

DimRed	$\lambda = 0$	$\lambda = 1e^{-5}$	$\lambda = 1e^{-3}$	$\lambda = 1e^{-1}$	Gaussian
PCA [50]	1.0%	1.0%	0.9%	1.5%	0.8%

MNIST — Multiclass error rates for LR with quadratic feature expansion

DimRed	$\lambda = 0$	$\lambda = 1e^{-5}$	$\lambda = 1e^{-3}$	$\lambda = 1e^{-1}$	Gaussian
PCA [50]	2.3%	1.9%	1.7%	3.1%	3.6%

In the next lectures we will see a different approach for non-linear classification: Support Vector Machines (SVM)

Alternatively, neural networks (not part of this course) allow jointly estimating a parametric transformation $\phi(x, \Pi)$ and a classification rule (w, b) in the transformed space

Bayes decisions and Model evaluation

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Model evaluation for classification

Assess how good our model is on a held-out evaluation (test) set

We start considering binary classification problems

A possible solution is to compute the **accuracy** of the model, or, equivalently, the **error rate**, defined as

$$\text{accuracy} = \frac{\# \text{ of correctly classified samples}}{\# \text{ of samples}}$$

$$\text{error rate} = \frac{\# \text{ of incorrectly classified samples}}{\# \text{ of samples}} = 1 - \text{accuracy}$$

Model evaluation for classification

Accuracy can be misleading if the classes are not balanced

Let's consider, for example, rain prediction in arid climates. Over one year, the model makes the following predictions:

	Rain	Clear
Prediction: Rain	15 days	30 days
Prediction: Clear	20 days	300 days

$$\text{accuracy} = \frac{300 + 15}{365} \approx 86\%$$

Model evaluation for classification

Accuracy can be misleading if the classes are not balanced

Let's consider, for example, rain prediction in arid climates. Over one year, the model makes the following predictions:

	Rain	Clear
Prediction: Rain	15 days	30 days
Prediction: Clear	20 days	300 days

$$\text{accuracy} = \frac{300 + 15}{365} \approx 86\%$$

86% looks like a good accuracy ... But a model that always predicts Clear would achieve an accuracy of $\approx 90\%$!

Model evaluation for classification

Let's analyze the outcomes table

	Rain	Clear
Prediction: Rain	15 days	30 days
Prediction: Clear	20 days	300 days

This table is also called **confusion matrix**. In general:

	Class \mathcal{H}_F	Class \mathcal{H}_T
Prediction \mathcal{H}_F	True Negative	False Negative
Prediction \mathcal{H}_T	False Positive	True Positive

Model evaluation for classification

We can compute different accuracy measures

- False negative rate FNR (false rejection / miss rate): $\frac{FN}{FN+TP}$
- False positive rate FPR (false acceptance): $\frac{FP}{FP+TN}$
- True positive rate TPR (recall, sensitivity): $\frac{TP}{FN+TP} = 1 - \text{FNR}$
- True negative rate TNR (specificity): $\frac{TN}{FP+TN} = 1 - \text{FPR}$
- ...

We can also compute a *weighted* accuracy

$$acc = \alpha FPR + (1 - \alpha) FNR$$

The weight α measures how important are different kind of errors (we shall see this in more detail later)

Model evaluation for classification

Different kind of errors may have different impact on applications

Systems providing only hard decisions do not allow for trade-offs between different error types

Rather than labels, often decision functions output **scores**

- Generative models: log-likelihood ratios

$$s = \log \frac{f(x|\mathcal{H}_T)}{f(x|\mathcal{H}_F)}$$

- Discriminative models: posterior log-likelihood ratios

$$s = \log \frac{P(\mathcal{H}_T|x)}{P(\mathcal{H}_F|x)}$$

- Non-probabilistic models: score (e.g. SVM)

$$s = \mathbf{w}^T \mathbf{x}$$

Model evaluation for classification

A higher score means we should favor class \mathcal{H}_T

Class assignment is performed by comparing scores to a threshold t

$$s \geq t \longrightarrow \mathcal{H}_T$$

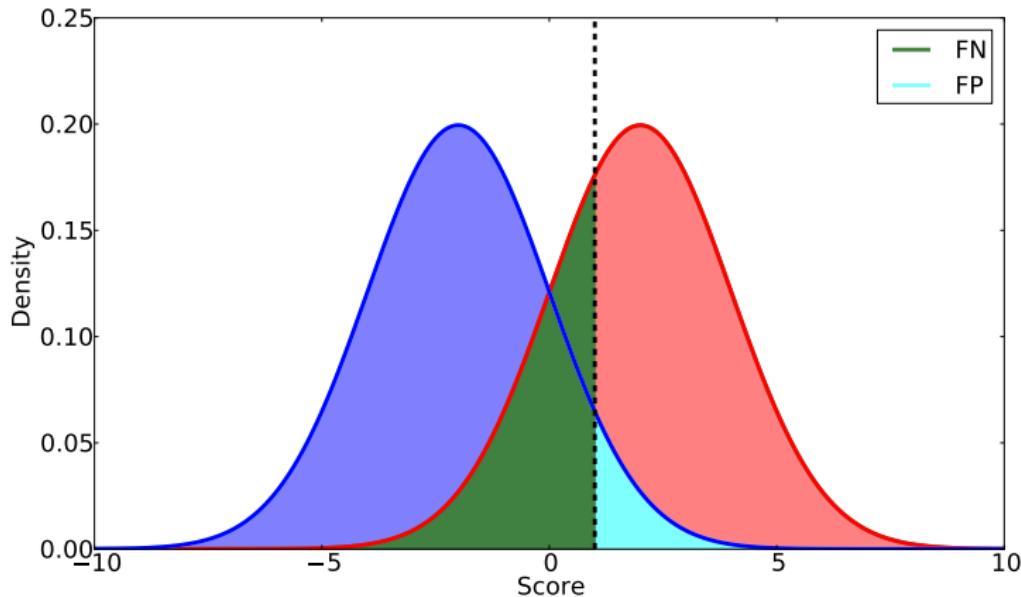
$$s < t \longrightarrow \mathcal{H}_F$$

Different thresholds correspond to different error rates

Thresholds are related to class priors and error costs

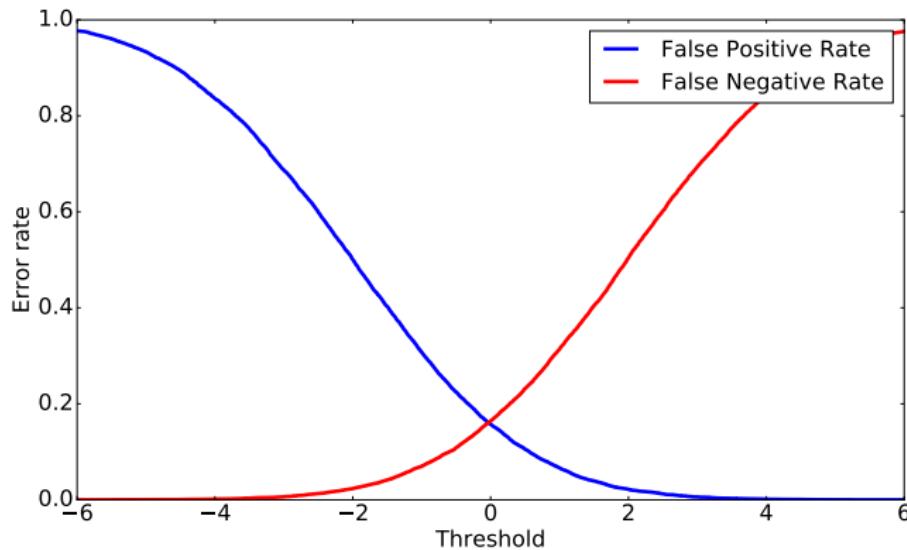
Model evaluation for classification

Score thresholding:



Model evaluation for classification

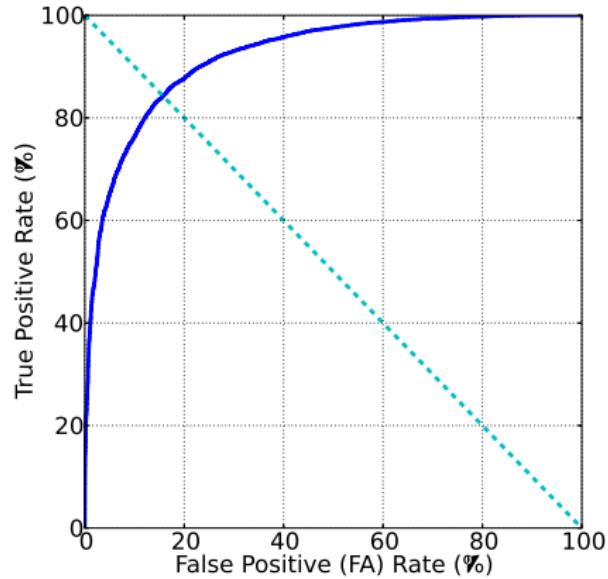
We can visualize the performance of the classifier for different thresholds by plotting the error rates as a function of the threshold



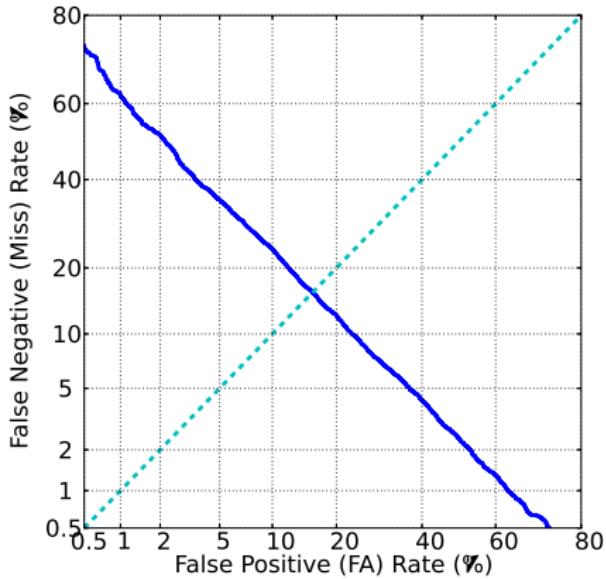
Equal Error Rate (EER): The error rate for which $FPR = FNR$

Model evaluation for classification

- Receiver Operating Characteristic (ROC) curve



- Detection Error Trade-off (DET) curve



Bayes decisions

The goal of the classifier is to allow us to choose an action a to perform among a set of actions \mathcal{A}

Example: accepting vs rejecting a sample

Example: assign label a to the sample

We can associate to each action a **cost** $C(a|k)$ that we have to pay when we choose action a and the sample belongs to class k

In the following we consider the set of actions corresponding to labeling a sample with label a

Bayes decisions

$C(a|k)$ represents the cost of labeling the sample as belonging to class a when it actually belongs to class k

We do not know k , however we have a classifier \mathcal{R} that allows us computing class posterior probabilities $P(C = k|x)$ for sample x

We can thus compute the expected cost of action a when the posterior probability for each class is $P(C = k|x, \mathcal{R})$

$$\mathcal{C}_{x,\mathcal{R}}(a) = \mathbb{E}[C(a|k)|x, \mathcal{R}] = \sum_{k=1}^K C(a|k)P(C = k|x, \mathcal{R})$$

It measures the cost that we expect to pay given our knowledge of the class distribution $P(C = k|x, \mathcal{R})$

Bayes decisions

The Bayes decision consists in choosing the action $a^*(x, \mathcal{R})$ that minimizes the expected cost: $a^*(x, \mathcal{R}) = \arg \min_a \mathcal{C}_{x, \mathcal{R}}(a)$

It represents the action that will result in the lower expected cost, according to the recognizer beliefs

Different recognizers may have different posterior beliefs, and thus provide different decisions

Bayes decisions

For example, let's consider we have a 3-class problem, with cost matrix and priors given by

$$\mathbf{C} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}, \quad \boldsymbol{\pi} = \begin{bmatrix} 0.3 \\ 0.4 \\ 0.3 \end{bmatrix}$$

For a test sample \mathbf{x}_t , we have computed the posterior class probabilities (using the prior $\boldsymbol{\pi}$)

$$\mathbf{q}_t = \begin{bmatrix} P(C = 1|\mathbf{x}_t, \mathcal{R}) \\ P(C = 2|\mathbf{x}_t, \mathcal{R}) \\ P(C = 3|\mathbf{x}_t, \mathcal{R}) \end{bmatrix} = \begin{bmatrix} 0.40 \\ 0.25 \\ 0.35 \end{bmatrix}$$

Bayes decisions

The expected cost of actions “Predict k” are

$$C_{x_t, \mathcal{R}}(1) = 0 \times 0.40 + 1 \times 0.25 + 2 \times 0.35 = 0.95$$

$$C_{x_t, \mathcal{R}}(2) = 1 \times 0.40 + 0 \times 0.25 + 1 \times 0.35 = 0.75$$

$$C_{x_t, \mathcal{R}}(3) = 2 \times 0.40 + 1 \times 0.25 + 0 \times 0.35 = 1.05$$

or, in matrix form:

$$\begin{bmatrix} C_{x_t, \mathcal{R}}(1) \\ C_{x_t, \mathcal{R}}(2) \\ C_{x_t, \mathcal{R}}(3) \end{bmatrix} = \mathbf{Cq}_t$$

The optimal decision would therefore to assign label 2, even though it has the lowest posterior probability, since the expected cost due to mis-calssifications would be lower.

Bayes decisions

Let's now consider again a binary problem

We have four costs:

	Class \mathcal{H}_F	Class \mathcal{H}_T
Prediction \mathcal{H}_F	$C(\mathcal{H}_F \mathcal{H}_F)$	$C(\mathcal{H}_F \mathcal{H}_T)$
Prediction \mathcal{H}_T	$C(\mathcal{H}_T \mathcal{H}_F)$	$C(\mathcal{H}_T \mathcal{H}_T)$

Without loss of generality we assume

$$C(\mathcal{H}_T|\mathcal{H}_T) = 0, \quad C(\mathcal{H}_F|\mathcal{H}_F) = 0$$

i.e. correct decisions have no cost.

We also assume $C(\mathcal{H}_F|\mathcal{H}_T) \geq 0$ and $C(\mathcal{H}_T|\mathcal{H}_F) \geq 0$

Bayes decisions

The costs reflect the costs of the two different kind of errors:

	Class \mathcal{H}_F	Class \mathcal{H}_T
Prediction \mathcal{H}_F	0	$C(\mathcal{H}_F \mathcal{H}_T) = C_{fn}$
Prediction \mathcal{H}_T	$C(\mathcal{H}_T \mathcal{H}_F) = C_{fp}$	0

C_{fn} is the cost of false negative errors, C_{fp} is the cost of false positive errors

Bayes decisions

The expected Bayes cost for action \mathcal{H}_T (i.e. for predicting \mathcal{H}_T) is

$$\mathcal{C}_{x,\mathcal{R}}(\mathcal{H}_T) = C_{fp}P(\mathcal{H}_F|x, \mathcal{R}) + 0 \cdot P(\mathcal{H}_T|x, \mathcal{R}) = C_{fp}P(\mathcal{H}_F|x, \mathcal{R})$$

whereas the cost for action \mathcal{H}_F (i.e. for predicting \mathcal{H}_F) is

$$\mathcal{C}_{x,\mathcal{R}}(\mathcal{H}_F) = C_{fn}P(\mathcal{H}_T|x, \mathcal{R}) + 0 \cdot P(\mathcal{H}_F|x, \mathcal{R}) = C_{fn}P(\mathcal{H}_T|x, \mathcal{R})$$

The optimal decision is the labeling that has lowest cost

Bayes decisions

For binary problems, the optimal decision can be expressed as

$$a^*(x, \mathcal{R}) = \begin{cases} \mathcal{H}_T & \text{if } C_{fp}P(\mathcal{H}_F|x, \mathcal{R}) < C_{fn}P(\mathcal{H}_T|x, \mathcal{R}) \\ \mathcal{H}_F & \text{if } C_{fp}P(\mathcal{H}_F|x, \mathcal{R}) > C_{fn}P(\mathcal{H}_T|x, \mathcal{R}) \end{cases}$$

and we can choose any action when the two costs are equal.

Alternatively, we can express the optimal decision (up to tie-breaking) as

$$a^*(x, \mathcal{R}) = \begin{cases} \mathcal{H}_T & \text{if } r(x) > 0 \\ \mathcal{H}_F & \text{if } r(x) < 0 \end{cases}$$

where

$$r(x) = \log \frac{C_{fn}P(\mathcal{H}_T|x, \mathcal{R})}{C_{fp}P(\mathcal{H}_F|x, \mathcal{R})}$$

Bayes decisions

If \mathcal{R} is a generative model for x , then we can express r in terms of costs, prior probabilities and conditional likelihoods as

$$r(x) = \log \frac{\pi_T C_{fn}}{(1 - \pi_T) C_{fp}} \cdot \frac{f_{X|\mathcal{H},\mathcal{R}}(x|\mathcal{H}_T)}{f_{X|\mathcal{H},\mathcal{R}}(x|\mathcal{H}_F)}$$

where $\pi_T = P(\mathcal{H} = \mathcal{H}_T)$ is the prior probability for class \mathcal{H}_T .

The decision rule thus becomes

$$r(x) \leq 0 \iff \log \frac{f_{X|\mathcal{H},\mathcal{R}}(x|\mathcal{H}_T)}{f_{X|\mathcal{H},\mathcal{R}}(x|\mathcal{H}_F)} \leq -\log \frac{\pi_T C_{fn}}{(1 - \pi_T) C_{fp}}$$

Bayes decisions

The triplet (π_T, C_{fn}, C_{fp}) represents the **working point** of an **application** for a binary classification task.

We can show that the triplet is actually redundant, in the sense that we can build equivalent applications $(\pi'_T, C'_{fn}, C'_{fp})$ which have the same decision rule as the original application, but different costs and priors.

For example, the application $(\tilde{\pi}, 1, 1)$ with

$$\tilde{\pi} = \frac{\pi_T C_{fn}}{\pi_T C_{fn} + (1 - \pi_T) C_{fp}}$$

is equivalent to the application (C_{fn}, C_{fp}, π_T) . Indeed, we have

$$\frac{\tilde{\pi}}{1 - \tilde{\pi}} = \frac{\frac{\pi_T C_{fn}}{\pi_T C_{fn} + (1 - \pi_T) C_{fp}}}{1 - \frac{\pi_T C_{fn}}{\pi_T C_{fn} + (1 - \pi_T) C_{fp}}} = \frac{\pi_T C_{fn}}{(1 - \pi_T) C_{fp}}$$

Bayes decisions

We can interpret $\tilde{\pi}$ as an **effective** prior: if the class prior for \mathcal{H}_T was $\tilde{\pi}$ and we assumed uniform costs, we would obtain the same decisions as for our original application

Similarly, we can devise an equivalent application where the effective prior is uniform $\tilde{\pi} = \frac{1}{2}$, and the application prior π_T absorbed in “effective” classification costs (we won’t prove it here)

Bayes decisions

We have, up to now, considered how to perform decisions for a sample x

We now return to the problem of evaluating the goodness of our decisions

Remember that our decisions are taken using the posterior class distribution defined by the classifier \mathcal{R}

We can compute the cost of the Bayes decisions taken with the recognizer \mathcal{R} :

$$\mathcal{C}^*(x, \mathcal{R}|c) = C(a^*(x, \mathcal{R})|c)$$

If actions correspond to class labeling, it's the cost of predicting the minimum-expected-cost label a^* when the actual label is c .

Empirical Bayes Risk

We would like to know how well the classifier performs on some data.

We can formalize this as computing an expectation of the Bayes cost of optimal Bayes decisions made with \mathcal{R} for the evaluation population.

We define the posterior **Bayes risk** \mathcal{B} as the expected value of the Bayes cost of Bayes decisions made by \mathcal{R} over evaluation data sampled from $X, C|\mathcal{E}$:

$$\mathcal{B} = \mathbb{E}_{X,C|\mathcal{E}} [\mathcal{C}^*(x, \mathcal{R}|c)] = \sum_{c=1}^K \pi_c \int \mathcal{C}^*(x, \mathcal{R}|c) f_{X|C,\mathcal{E}}(x|c) dx$$

Empirical Bayes Risk

The distribution $X|C, \mathcal{E}$ is the conditional distribution of the evaluation population

\mathcal{E} represents the **evaluator**

Note that the distribution reflects the knowledge of the evaluator \mathcal{E} , not the knowledge of the recognizer \mathcal{R}

The evaluator \mathcal{E} is measuring how good are the decisions made by the recognizer \mathcal{R} for his own task (data sampled from $X|C, \mathcal{E}$)

Empirical Bayes Risk

In general, we won't have access to $f_{X|C,\mathcal{E}}(x|c)$.

However, if we have at our disposal a set of labeled evaluation samples $(x_1, c_1) \dots (x_N, c_N)$, then we can approximate the expectations by averaging the cost over the samples

Indeed, if samples x_i are generated by $X|C, \mathcal{E}$, as the number of samples per class becomes large, it's possible to show that

$$\int \mathcal{C}^*(x, \mathcal{R}|c) f_{X|C,\mathcal{E}}(x|c) dx \approx \frac{1}{N_c} \sum_{i|c_i=c} \mathcal{C}^*(x_i, \mathcal{R}|c)$$

i.e. the integral can be approximated by the average cost computed over samples of each class

Empirical Bayes Risk

We can finally define the **empirical Bayes risk** as

$$\mathcal{B}_{emp} = \sum_{c=1}^K \frac{\pi_c}{N_c} \sum_{i|c_i=c} \mathcal{C}^*(x_i, \mathcal{R}|c)$$

The risk measures the costs of our decisions over the evaluation samples.

We can use \mathcal{B}_{emp} to compare recognizers.

A recognizer that has lower cost will provide more accurate answers

Empirical Bayes Risk

The empirical Bayes risk can be computed from the confusion matrix and the matrix of costs

For example, let's consider again the 3-class problem, with cost matrix and priors given by

$$\mathbf{C} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}, \quad \boldsymbol{\pi} = \begin{bmatrix} 0.3 \\ 0.4 \\ 0.3 \end{bmatrix}$$

For all test samples, we computed the Bayes decisions. We can then compute the confusion matrix, Let's assume that we obtain

$$\mathbf{M} = \begin{bmatrix} 205 & 111 & 56 \\ 145 & 199 & 121 \\ 50 & 92 & 225 \end{bmatrix}$$

Empirical Bayes Risk

We can compute, for each class, the term

$$\frac{\pi_c}{N_c} \sum_{i|c_i=c} \mathcal{C}^*(x_i, \mathcal{R}|c)$$

For samples that belong to class 1, we have

$$\pi_1 = 0.3, \quad N_1 = 205 + 145 + 50 = 400.$$

For samples that are correctly classified (205) the cost is 0; for samples that are classified as class 2 (145) the cost is 1; for samples that are classified as class 3 (50) the cost is 2. Thus

$$\frac{\pi_1}{N_1} \sum_{i|c_i=1} \mathcal{C}^*(x_i, \mathcal{R}|c) = \frac{0.3}{400} (0 \times 205 + 1 \times 145 + 2 \times 50) = 0.18375$$

Empirical Bayes Risk

Similarly,

$$\frac{\pi_2}{N_2} \sum_{i|c_i=2} \mathcal{C}^*(x_i, \mathcal{R}|c) = \frac{0.4}{402} (1 \times 111 + 0 \times 199 + 1 \times 92) \approx 0.20199$$

$$\frac{\pi_3}{N_3} \sum_{i|c_i=3} \mathcal{C}^*(x_i, \mathcal{R}|c) = \frac{0.3}{402} (2 \times 56 + 1 \times 121 + 0 \times 225) \approx 0.17388$$

The empirical Bayes risk is

$$\mathcal{B}_{emp} \approx 0.18375 + 0.20199 + 0.17388 = 0.55962$$

Empirical Bayes Risk

Let's now consider again the binary problem

	Class \mathcal{H}_F	Class \mathcal{H}_T
Prediction \mathcal{H}_F	0	$C(\mathcal{H}_F \mathcal{H}_T) = C_{fn}$
Prediction \mathcal{H}_T	$C(\mathcal{H}_T \mathcal{H}_F) = C_{fp}$	0

We have seen that we can compute predicted labels by comparing the log-likelihood ratio to a threshold that depends on (π_T, C_{fn}, C_{fp})

Empirical Bayes Risk

Let c_i^* be the predicted label for sample x_i . The empirical Bayes risk is

$$\begin{aligned}\mathcal{B}_{emp} &= \frac{\pi_T}{N_T} \sum_{i|c_i=\mathcal{H}_T} \mathcal{C}^*(x_i, \mathcal{R}|\mathcal{H}_T) + \frac{1-\pi_T}{N_F} \sum_{i|c_i=\mathcal{H}_F} \mathcal{C}^*(x_i, \mathcal{R}|\mathcal{H}_F) \\ &= \pi_T \frac{\sum_{i|c_i=\mathcal{H}_T} C_{fn} \mathbb{I}[h_i^* = \mathcal{H}_F]}{N_T} + (1-\pi_T) \frac{\sum_{i|c_i=\mathcal{H}_F} C_{fp} \mathbb{I}[h_i^* = \mathcal{H}_T]}{N_F} \\ &= \pi_T \frac{\sum_{i|c_i=\mathcal{H}_T, c_i^*=\mathcal{H}_F} C_{fn}}{N_T} + (1-\pi_T) \frac{\sum_{i|c_i=\mathcal{H}_F, c_i^*=\mathcal{H}_T} C_{fp}}{N_F} \\ &= \pi_T C_{fn} \cdot FNR + (1-\pi_T) C_{fp} \cdot FPR \\ &= \pi_T C_{fn} P_{fn} + (1-\pi_T) C_{fp} P_{fp}\end{aligned}$$

where $P_{fn} = FNR$ is the false negative rate (false rejection rate) and $P_{fp} = FPR$ is the false positive rate (false acceptance rate)

\mathcal{B}_{emp} is also called (un-normalized) Detection Cost Function (DCF)

Detection Cost Functions:

- Define the costs of different kind of errors (C_{fn} , C_{fp})
- Define the class prior probability (π_T , $\pi_F = 1 - \pi_T$)
- Evaluate by computing empirical Bayes risk

$$DCF_u(C_{fn}, C_{fp}, \pi_T) = \pi_T C_{fn} P_{fn} + (1 - \pi_T) C_{fp} P_{fp}$$

- P_{fn} and P_{fp} are the false negative and false positive rates, and depend on the selected threshold t

Model evaluation for classification

C_{fn} , C_{fp} and π_T depend only on the application

A dummy system that always accepts a test segment ($c_t^* = \mathcal{H}_T$):

$$P_{fp} = 1, P_{fn} = 0 \implies DCF_u = (1 - \pi_T)C_{fp}$$

A dummy system that always rejects a test segment ($c_t^* = \mathcal{H}_F$):

$$P_{fp} = 0, P_{fn} = 1 \implies DCF_u = \pi_T C_{fn}$$

Normalized DCF: we compare the system DCF w.r.t. the best dummy system

$$DCF(\pi_T, C_{fn}, C_{fp}) = \frac{DCF_u(\pi_T, C_{fn}, C_{fp})}{\min(\pi_T C_{fn}, (1 - \pi_T)C_{fp})}$$

Model evaluation for classification

Normalized DCF is invariant to scaling

We can thus re-scale the un-normalized DCF by $\frac{1}{\pi_T C_{fn} + (1 - \pi_T) C_{fp}}$

Let $\tilde{\pi} = \frac{\pi_T C_{fn}}{\pi_T C_{fn} + (1 - \pi_T) C_{fp}}$, so that $1 - \tilde{\pi} = \frac{(1 - \pi_T) C_{fp}}{\pi_T C_{fn} + (1 - \pi_T) C_{fp}}$

The un-normalized DCF becomes

$$DCF_u(\tilde{\pi}) = \tilde{\pi} P_{fn} + (1 - \tilde{\pi}) P_{fp}$$

whereas the corresponding normalized DCF has the same value

In terms of normalized DCF, the applications (π_T, C_{fp}, C_{fn}) and $(\tilde{\pi}, 1, 1)$ are again equivalent

Model evaluation for classification

We can observe that the error rate we defined at the beginning as

$$e = \frac{\# \text{ of incorrectly classified samples}}{\# \text{ of samples}} = 1 - \text{accuracy}$$

corresponds to

$$e = \frac{N_T P_{fn} + N_F P_{fp}}{N} = \frac{N_T}{N} P_{fn} + \frac{N_F}{N} P_{fp}$$

i.e., up to a scaling factor, to the DCF of an application $(\frac{N_T}{N}, 1, 1)$

The weighted error rate

$$e = \frac{1}{2}(P_{fn} + P_{fp})$$

corresponds to the application $(\frac{1}{2}, 1, 1)$

Model evaluation for classification

For systems producing well-calibrated log-likelihood ratios

$$s = \log \frac{f_{X|C}(x|\mathcal{H}_T)}{f_{X|C}(x|\mathcal{H}_F)}$$

the optimal threshold is given by

$$t = -\log \frac{\tilde{\pi}}{1 - \tilde{\pi}}$$

Indeed, since $\tilde{\pi}$ is the effective prior probability of \mathcal{H}_T , the posterior log-likelihood ratio is

$$\log \frac{P(\mathcal{H}_T|x)}{P(\mathcal{H}_F|x)} = \log \frac{f_{X|C}(x|\mathcal{H}_T)}{f_{X|C}(x|\mathcal{H}_F)} + \log \frac{\tilde{\pi}}{1 - \tilde{\pi}}$$

Model evaluation for classification

LLRs allow disentangling the classifier from the application

In general, systems often do not produce well-calibrated LLRs

- Non-probabilistic scores (e.g. SVM)
- Mis-match between train and test populations
- Non-accurate model assumptions

In these cases, we say that scores are **mis-calibrated**

The theoretical threshold – $\log \frac{\tilde{\pi}}{1-\tilde{\pi}}$ is not optimal anymore

Model evaluation for classification

For a given application, we can measure the additional cost due to the use of mis-calibrated scores

We can define the **minimum** cost DCF_{min} corresponding to the use of the optimal threshold for the evaluation set

We consider varying the threshold t to obtain all possible combinations of P_{fn} and P_{fp} for the evaluation set

We select the threshold corresponding to the lowest DCF

Model evaluation for classification

The corresponding value DCF_{min} is the cost we would pay if we knew before-hand the optimal threshold for the evaluation

We can think of this value as a measure of the quality of the classifier

We can also compute the **actual** DCF obtained using the threshold corresponding to the effective prior $\tilde{\pi}$

The difference between the actual and minimum DCF represents the loss due to score mis-calibration

Model evaluation for classification

We can also compare different systems over different applications through Bayes error plots

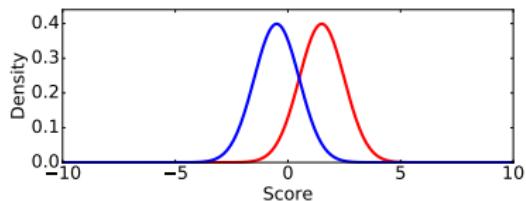
These plots can be used to report actual and / or minimum DCF for different applications

A binary application is parametrized by a single value $\tilde{\pi}$

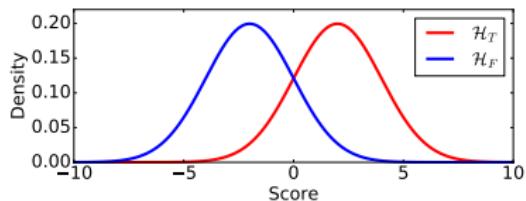
We can thus plot the DCF as a function of prior log-odds $\log \frac{\tilde{\pi}}{1-\tilde{\pi}}$, i.e. the negative of the Bayes optimal threshold.

Model evaluation for classification

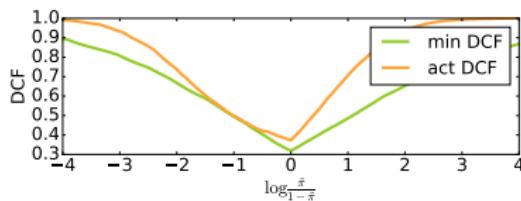
Non calibrated scores



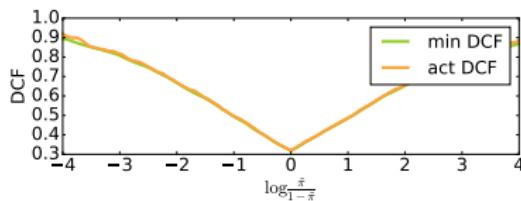
Calibrated scores



Bayes error plot



Bayes error plot



Model evaluation for classification

To reduce mis-calibration costs we can adopt different calibration strategies

We can use a validation set to find a (close-to) optimal threshold for a given application

More general approaches look for functions that transform the classifier scores s into well-calibrated LLRs, possibly in a way that is as much as possible independent from the target application

These usually also employ a validation set (calibration set in this case) to estimate the mapping between the classifier scores and well-calibrated scores

Score calibration approaches:

- Isotonic regression
- Prior-weighted logistic regression
- Generative score models (e.g. Gaussian score models)

Model evaluation for classification

Evaluation of multiclass tasks is more complex

As we have seen, we can build confusion matrices

We can also compute the empirical Bayes risk for multiclass problems

We can compute a normalized detection cost, obtained by scaling the Bayes risk by the cost of the best dummy system — in this case, we have K dummy systems, each of them predicting a different class k regardless of the sample

For the multiclass problem, it's more difficult to separate the costs due to mis-calibration from those due to poor discriminant capabilities of the classifier

Support Vector Machines

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Support Vector Machines

Let's consider again a binary classification problem

We have shown that logistic regression provides a linear classification rule that maximizes the log-probability of class assignments $\sum_{i=1}^n \log P(C_i = c_i | \mathbf{X}_i = \mathbf{x}_i, \mathbf{w}, b)$ for the training set

The LR solution can be found by minimizing the logistic loss

Support Vector Machines

As we have mentioned, it is often useful to add a regularization term of the form $\frac{\lambda}{2} \|\mathbf{w}\|^2$:

$$\mathbf{w}^*, b = \arg \min_{\mathbf{w}, b} \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{n} \sum_{i=1}^n \log \left(1 + e^{-z_i(\mathbf{w}^T \mathbf{x}_i + b)} \right)$$

where z_i is the class label for sample \mathbf{x}_i , encoded as

$$z_i = \begin{cases} +1 & \text{if } c_i = \mathcal{H}_T \\ -1 & \text{if } c_i = \mathcal{H}_F \end{cases}$$

We can alternatively minimize a scaled version of the objective:

$$\mathbf{w}^*, b = \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \log \left(1 + e^{-z_i(\mathbf{w}^T \mathbf{x}_i + b)} \right)$$

where $C = \frac{1}{n\lambda}$

Support Vector Machines

We now consider a different classifier that allows us to give a geometrical interpretation of the regularization term: the Support Vector Machine

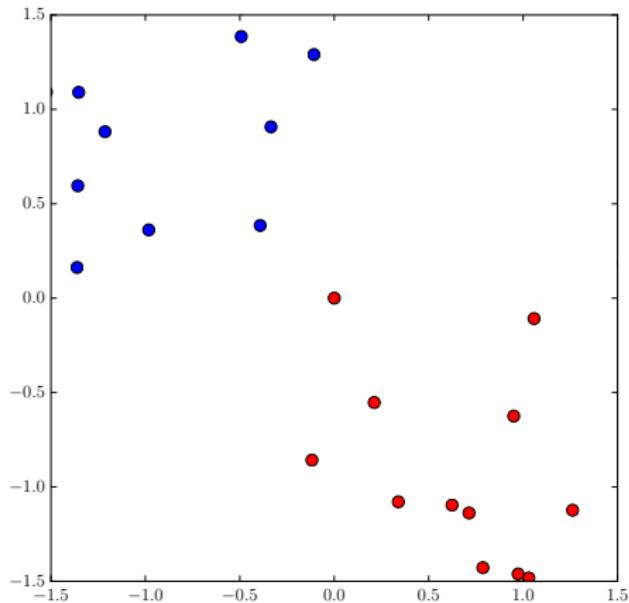
We will show that SVM can be cast as a generalized risk minimization problem

Furthermore, SVMs provide a natural way to achieve non-linear separation [without the need for an explicit expansion](#) of our features.

In contrast with LR, however, the output of SVMs cannot be directly interpreted as class posteriors.

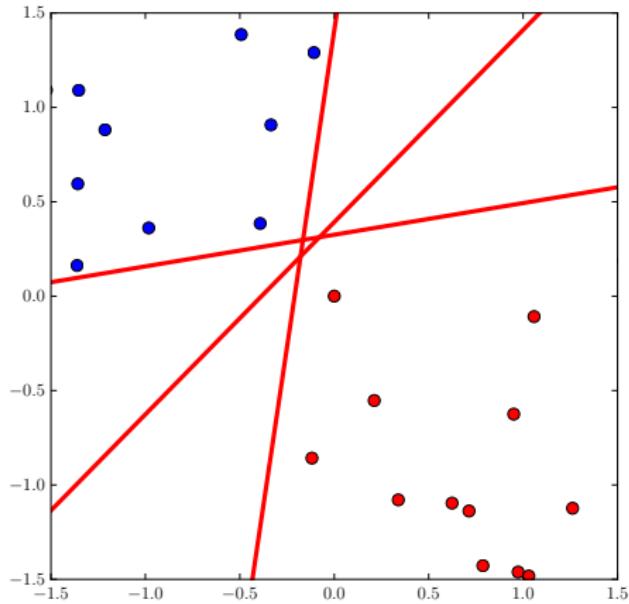
Support Vector Machines

Assume that we have two classes that are linearly separable (i.e., we can find a linear separation hyperplane that correctly classifies all points of our training set)



Support Vector Machines

Problem: there is an infinite number of separating hyperplanes



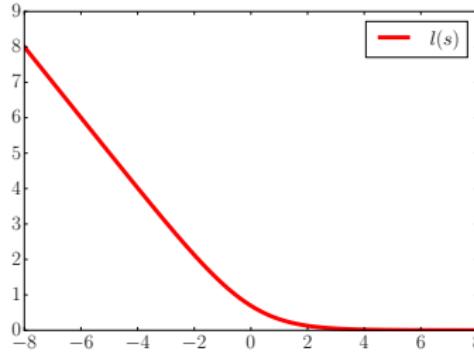
Support Vector Machines

Logistic Regression will find the hyperplane that maximizes class probabilities

Since $P(C_i = c_i | \mathbf{x}, \mathbf{w}, b) = \sigma(z_i(\mathbf{w}^T \mathbf{x} + b))$, and classes are separable, posterior probabilities can grow arbitrarily close to 1 and 0 (depending on the class) as long as the norm of \mathbf{w} is sufficiently large

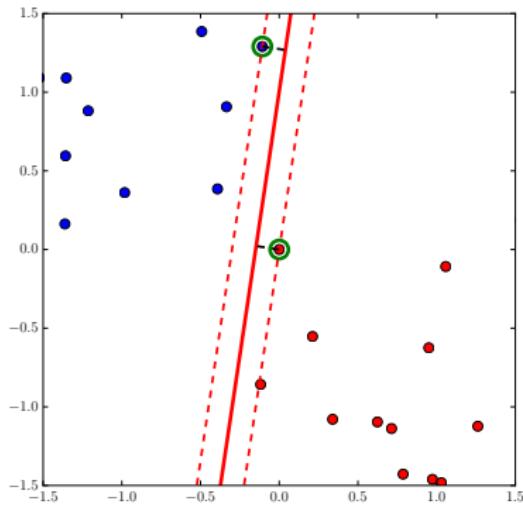
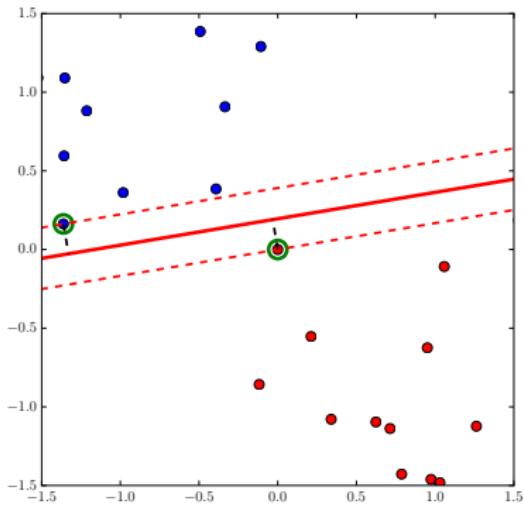
The “optimal” LR solution would have $\|\mathbf{w}\|^2 \rightarrow +\infty$

LR optimizes the Logistic Loss. As we move further along the x axis the loss tends to 0



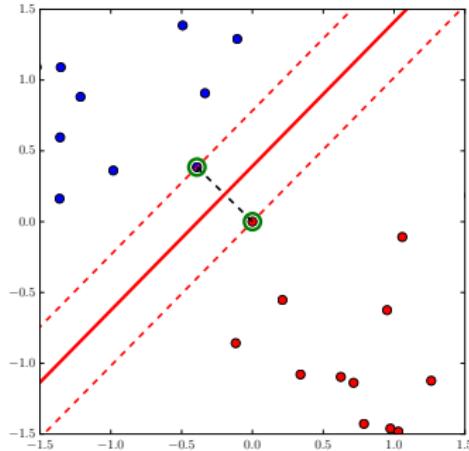
Support Vector Machines

Intuitively, we can select the hyperplane that separates the classes with the largest margin



Support Vector Machines

Intuitively, we can select the hyperplane that separates the classes with the largest margin



The margin is defined as the distance of the closest point w.r.t. the separation hyperplane¹

¹Some authors separately introduce the margin for each class. This distinction has no practical effects on the derivation of the SVM objective.

Support Vector Machines

Let $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$ be the function representing the separation surface

The distance of \mathbf{x}_i from the hyperplane is

$$d(\mathbf{x}_i) = \frac{|f(\mathbf{x}_i)|}{\|\mathbf{w}\|}$$

Let z_i denote the class for \mathbf{x}_i , with the usual encoding

$$z_i = \begin{cases} +1 & \text{if } c_i = \mathcal{H}_T \\ -1 & \text{if } c_i = \mathcal{H}_F \end{cases}$$

Since classes are separable, we consider solutions which correctly classify all points:

$$\begin{aligned} f(\mathbf{x}_i) &> 0 \text{ if } c_i = \mathcal{H}_T \\ f(\mathbf{x}_i) &< 0 \text{ if } c_i = \mathcal{H}_F \end{aligned}$$

Support Vector Machines

The distance of \mathbf{x} from the hyperplane can be rewritten as

$$d(\mathbf{x}) = \frac{|f(\mathbf{x})|}{\|\mathbf{w}\|} = \frac{|z_i(\mathbf{w}^T \mathbf{x} + b)|}{\|\mathbf{w}\|}$$

The maximum margin hyperplane is the hyperplane which maximizes the minimum distance of all points from the hyperplane:

$$\mathbf{w}^*, b^* = \arg \max_{\mathbf{w}, b} \min_{i \in \{1 \dots n\}} d(\mathbf{x}_i) = \arg \max_{\mathbf{w}, b} \min_{i \in \{1 \dots n\}} \frac{|z_i(\mathbf{w}^T \mathbf{x}_i + b)|}{\|\mathbf{w}\|}$$

subject to $z_i(\mathbf{w}^T \mathbf{x} + b) > 0$.

Support Vector Machines

For values \mathbf{w}, b which can correctly separate the classes we have $z_i(\mathbf{w}^T \mathbf{x}_i + b) > 0$ for all samples, and $\min_i z_i(\mathbf{w}^T \mathbf{x}_i + b) > 0$.

We can drop the constraint by considering an equivalent problem

$$\begin{aligned}\mathbf{w}^*, b^* &= \arg \max_{\mathbf{w}, b} \min_i \frac{z_i(\mathbf{w}^T \mathbf{x}_i + b)}{\|\mathbf{w}\|} \\ &= \arg \max_{\mathbf{w}, b} \frac{1}{\|\mathbf{w}\|} \min_i [z_i(\mathbf{w}^T \mathbf{x}_i + b)]\end{aligned}\tag{1}$$

Support Vector Machines

We can show the two problems are equivalent:

- Hyperplanes that do not satisfy this constraint will have $\min_i z_i(\mathbf{w}^T \mathbf{x} + b) < 0$, and thus cannot be an optimal solution of the second formulation (classes are separable, so an optimal solution exists with $z_i(\mathbf{w}^T \mathbf{x} + b) > 0, \forall i = 1 \dots n$)
- For hyperplanes that correctly classify all patterns, the objective functions are the same

Support Vector Machines

Direct optimization of (1) is non trivial, thus we further transform the problem as to arrive to an easier-to-solve, **equivalent**, problem.

We can observe that the objective function in (1) is invariant under re-scaling of the parameters:

$$\frac{1}{\|\mathbf{w}\|} \min_i [z_i(\mathbf{w}^T \mathbf{x}_i + b)] = \frac{1}{\|\alpha \mathbf{w}\|} \min_i [z_i(\alpha \mathbf{w}^T \mathbf{x}_i + \alpha b)]$$

for $\alpha > 0$ Thus, if (\mathbf{w}^*, b^*) is an optimal solution, then also $(\mathbf{w}', b') = (\alpha \mathbf{w}^*, \alpha b^*)$ is optimal, and viceversa.

Support Vector Machines

The collections of values $\{(\alpha \mathbf{w}, \alpha b)\}_{\alpha \in \mathbb{R}_+}$ forms an equivalence class of equivalent solutions

For each of these equivalence classes, we are free to select any one of the equivalent solutions. In particular, we restrict our problem to solutions for which

$$\min_i z_i (\mathbf{w}^T \mathbf{x}_k + b) = 1$$

For all training points we will thus have

$$z_i (\mathbf{w}^T \mathbf{x}_i + b) \geq 1, \quad i = 1 \dots n$$

Support Vector Machines

The problem in (1) then becomes equivalent to optimizing

$$\begin{aligned} & \arg \max_{w,b} \frac{1}{\|w\|} \\ \text{s.t. } & \begin{cases} z_i(w^T x_i + b) \geq 1, & i = 1 \dots n \\ \min_i z_i(w^T x_k + b) = 1 \end{cases} \end{aligned}$$

or, equivalently, minimizing the squared norm of w :

$$\begin{aligned} & \arg \min_{w,b} \frac{1}{2} \|w\|^2 \\ \text{s.t. } & \begin{cases} z_i(w^T x_i + b) \geq 1, & i = 1 \dots n \\ \min_i z_i(w^T x_k + b) = 1 \end{cases} \end{aligned}$$

Support Vector Machines

Finally, we observe that we can drop the last constraint, and solve the equivalent problem

$$\begin{aligned} & \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{s.t. } & z_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1, \quad i = 1 \dots n \end{aligned} \tag{2}$$

Indeed, an optimal solution of (2) will automatically satisfy $\min_i z_i(\mathbf{w}^T \mathbf{x}_i + b) = 1$

In fact, if \mathbf{w}^*, b^* was a solution for which $\min_i z_i(\mathbf{w}^T \mathbf{x}_k + b) = \psi > 1$, then we could build the solution $(\mathbf{w}', b') = \left(\frac{\mathbf{w}}{\psi}, \frac{b}{\psi} \right)$ which would still satisfy all constraints, and have a lower norm, thus \mathbf{w}^*, b^* would not be optimal

Support Vector Machines

The SVM objective thus consist in finding the minimizer of

$$\begin{aligned} & \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{s.t. } & z_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1, \quad i = 1 \dots n \end{aligned}$$

This is a [convex quadratic programming](#) problem

- The objective function is convex
- The constraints form a convex set

Support Vector Machines

To solve the SVM problem we consider a Lagrangian formulation of the problem

For each constraint we introduce the Lagrange multiplier $\alpha_i \geq 0$:

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^n \alpha_i [z_i (\mathbf{w}^T \mathbf{x}_i + b) - 1]$$

The optimal solution is obtained by minimizing L w.r.t. \mathbf{w}, b while requiring that the derivatives w.r.t. α_i vanish, subject to the constraints that $\alpha_i \geq 0$

Since the problem is convex, we can equivalently maximize L w.r.t. α_i while requiring that the derivatives w.r.t. \mathbf{w} and b vanish, subject to $\alpha_i \geq 0$

Support Vector Machines

Setting the derivative of L w.r.t. w and b equal to zero gives

$$w = \sum_{i=1}^n \alpha_i z_i x_i$$

$$0 = \sum_{i=1}^n \alpha_i z_i$$

Replacing these constraints in L gives the *dual* SVM problem

$$\max_{\alpha} L_D(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j z_i z_j x_i^T x_j$$

s.t.

$$\alpha_i \geq 0, \quad i = 1, \dots, n$$

$$\sum_{i=1}^n \alpha_i z_i = 0$$

Support Vector Machines

In matrix form:

$$L_D(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^T \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{H} \boldsymbol{\alpha}$$

where \mathbf{H} is the matrix

$$\mathbf{H}_{ij} = z_i z_j \mathbf{x}_i^T \mathbf{x}_j$$

Matrix \mathbf{H} depends on the data only through dot-products $\mathbf{x}_i^T \mathbf{x}_j$

As we will see shortly, this allows to extend SVMs to non-linear classification

Support Vector Machines

The optimal solution satisfies the Karush–Kuhn–Tucker (KKT) conditions²

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\alpha}) = \mathbf{w} - \sum_{i=1}^n \alpha_i z_i \mathbf{x}_i = \mathbf{0}$$

$$\frac{\partial L(\mathbf{w}, b, \boldsymbol{\alpha})}{\partial b} = - \sum_{i=1}^n \alpha_i z_i = 0$$

$$z_i (\mathbf{w}^T \mathbf{x}_i + b) - 1 \geq 0 \quad \forall i$$

$$\alpha_i \geq 0 \quad \forall i$$

$$\alpha_i [z_i (\mathbf{w}^T \mathbf{x}_i + b) - 1] = 0 \quad \forall i$$

²These are both necessary and sufficient conditions for optimality of the SVM solution

Support Vector Machines

The first and second equations encode that, at the optimal **primal** solution (w, b) , the gradient of the Lagrangian with respect to w and b becomes zero.

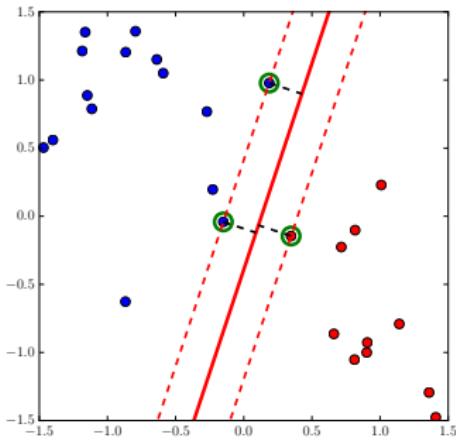
The next two equations encode that both the **primal** solution (w, b) and the corresponding **dual** solutions are feasible.

The last equation encodes that the optimal **dual** solution maximizes the Lagrangian. This requires that either the maximum is on the constraint, i.e. $\alpha_i = 0$, or the derivative of the Lagrangian is $\frac{\partial L}{\partial \alpha_i} = 0$. In compact form, $\alpha_i \frac{\partial L}{\partial \alpha_i} = 0$

Support Vector Machines

The optimal solution satisfies the Karush–Kuhn–Tucker (KKT) conditions

- For all points that do not lie on the margin (i.e. $z_i (\mathbf{w}^T \mathbf{x}_i + b) > 1$) the corresponding Lagrange multiplier is $\alpha_i = 0$
- $\alpha_i \neq 0$ implies that the corresponding point is on the margin, i.e., it is a **Support Vector**
- After we have obtained α , the KKT conditions allow us to estimate b



Support Vector Machines

Since $\mathbf{w} = \sum_{i=1}^n \alpha_i z_i \mathbf{x}_i$, the score for a test point x_t can be computed as

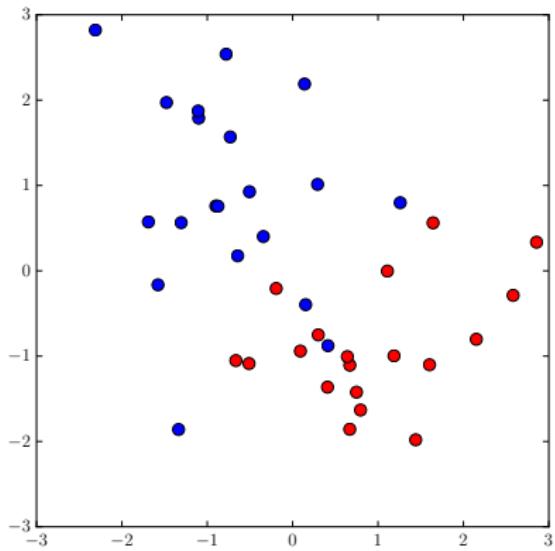
$$s(\mathbf{x}_t) = \mathbf{w}^T \mathbf{x}_t + b = \sum_{i=1}^n \alpha_i z_i \mathbf{x}_i^T \mathbf{x}_t + b$$

Notice that, again, this depends only on dot products $\mathbf{x}_i^T \mathbf{x}_t$

The training points that are not Support Vectors do not affect the separation surface

Support Vector Machines

Let's consider a problem where classes are not linearly separable



Support Vector Machines

No matter the value of w , some points will violate the constraint

$$z_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$$

We can try to minimize the number of points that violate the constraint

To do this, we introduce the **slack variables** $\xi_i \geq 0$, which represent how much a point is violating the constraint

We can replace the constraints $z_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1$ with

$$z_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

i.e. we allow training points to be inside the margin by a factor ξ_i

Support Vector Machines

We want to minimize the number of data points that lie inside the margin, i.e. the number of points for which $\xi_i > 0$

We can consider the functional

$$\Phi(\boldsymbol{\xi}) = \sum_{i=1}^n \xi_i^\sigma$$

with $\sigma > 0$

For sufficiently small values of σ , $\Phi(\boldsymbol{\xi})$ represents the number of points inside the margin

If we remove these data points the classes become linearly separable and we can thus train a maximum margin hyperplane over the remaining points

Support Vector Machines

Formally, this corresponds to the minimization of the functional

$$\frac{1}{2} \|\mathbf{w}\|^2 + C F \left(\sum_{i=1}^n \xi_i^\sigma \right)$$

s.t.

$$\begin{aligned} z_i(\mathbf{w}^T \mathbf{x}_i + b) &\geq 1 - \xi_i & \forall i \\ \xi_i &\geq 0 & \forall i \end{aligned}$$

where F is a monotone convex function and C is a constant

For sufficiently large C and small σ the optimal solution minimizes the number of points that violate the margin constraints and maximizes the margin between the remaining points

Support Vector Machines

Unfortunately, for small values of σ the problem is difficult

We simplify the problem by considering $\sigma = 1$, which guarantees a unique solution for the separation surface, and³ $F(u) = u$

The objective function becomes

$$\min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i$$

$$\text{s.t. } z_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i = 1 \dots n$$

$$\xi_i \geq 0 \quad \forall i = 1 \dots n$$

Note that this is again a convex quadratic programming problem

³SVMs with $\sigma = 2$ are sometimes used, and are usually referred to as L2-SVMs

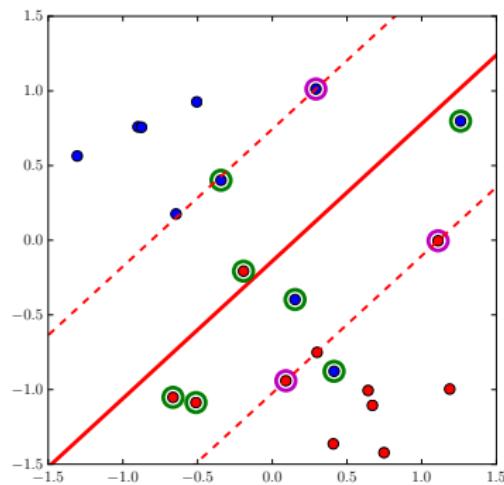
Support Vector Machines

The terms ξ_i do not describe the number of errors, but act as a penalty

- Points inside the margin have $\xi_i > 0$, and miss-classified points have $\xi_i > 1$
- The further the point is from the hyperplane, the larger the value of ξ_i
- $\sum_{i=1}^n \xi_i$ is an upper bound on the number of miss-classified points (errors)
- C allows selecting a trade-off between margin and errors on the training set
- The solution is often called a *soft margin* hyperplane

Support Vector Machines

Soft margin classification:



Support Vector Machines

As we did for the hard-margin SVM, we can introduce the Lagrangian problem

$$\begin{aligned} L(\mathbf{w}, b, \xi, \alpha, \mu) = \\ = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i [z_i (\mathbf{w}^T \mathbf{x} + b) - 1 + \xi_i] - \sum_{i=1}^n \mu_i \xi_i \end{aligned}$$

where $\alpha_i \geq 0$, and $\mu_i \geq 0$ are an additional set of Lagrange multipliers relative to the constraints $\xi_i \geq 0$

Support Vector Machines

The necessary and sufficient KKT conditions are

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\alpha}, \boldsymbol{\mu}) = \mathbf{w} - \sum_{i=1}^n \alpha_i z_i \mathbf{x}_i = \mathbf{0}$$

$$\frac{\partial L(\mathbf{w}, b, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial b} = - \sum_{i=1}^n \alpha_i z_i = 0$$

$$\frac{\partial L(\mathbf{w}, b, \boldsymbol{\alpha}, \boldsymbol{\mu})}{\partial \xi_i} = C - \alpha_i - \mu_i = 0 \quad \forall i$$

$$z_i (\mathbf{w}^T \mathbf{x}_i + b) - 1 + \xi_i \geq 0 \quad \forall i$$

$$\xi_i \geq 0 \quad \forall i$$

$$\alpha_i \geq 0 \quad \forall i$$

$$\mu_i \geq 0 \quad \forall i$$

$$\alpha_i [z_i (\mathbf{w}^T \mathbf{x}_i + b) - 1 + \xi_i] = 0 \quad \forall i$$

$$\mu_i \xi_i = 0 \quad \forall i$$

Support Vector Machines

From the KKT conditions (1)–(3)

$$\mathbf{w} = \sum_{i=1}^n \alpha_i z_i \mathbf{x}_i , \quad \sum_{i=1}^n \alpha_i z_i = 0 , \quad \alpha_i = C - \mu_i$$

Note that $\mu_i \geq 0$ implies that $\alpha_i \leq C$

We can then optimize the Lagrangian w.r.t. \mathbf{w} , b and ξ to obtain the dual problem

$$\max_{\boldsymbol{\alpha}} L_D(\boldsymbol{\alpha}) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j z_i z_j \mathbf{x}_i^T \mathbf{x}_j$$

s.t.

$$0 \leq \alpha_i \leq C, \quad i = 1, \dots, n$$

$$\sum_{i=1}^n \alpha_i z_i = 0$$

Support Vector Machines

Note that this problem is very similar to the hard-margin SVM

The constraint

$$\alpha_i \geq 0$$

is replaced by

$$0 \leq \alpha_i \leq C$$

Predictions are again computed as in the hard-margin case

$$s(\mathbf{x}_t) = \mathbf{w}^T \mathbf{x}_t + b = \sum_{i=1}^n \alpha_i z_i \mathbf{x}_i^T \mathbf{x}_t + b$$

Again, this depends only on dot products $\mathbf{x}_i^T \mathbf{x}_t$

Support Vector Machines

Several methods to solve the *dual* problem

Packages are available for several solvers

If we are interested in linear separation we can directly solve the *primal* SVM problem

Let's recall the primal formulation

$$\min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i$$

s.t.

$$z_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i \quad \forall i$$

$$\xi_i \geq 0 \quad \forall i$$

Support Vector Machines

For data points that are on the correct side of the margin

$$\xi_i = 0$$

while for the others we have

$$\xi_i = 1 - z_i(\mathbf{w}^T \mathbf{x}_i + b)$$

Thus the SVM primal problem can then be rewritten as

$$\min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \max [0, 1 - z_i(\mathbf{w}^T \mathbf{x}_i + b)]$$

where the constraints have been absorbed by the *loss* terms

$$\max [0, 1 - z_i(\mathbf{w}^T \mathbf{x}_i + b)]$$

Support Vector Machines

Function

$$f(s) = \max(0, 1 - s)$$

is known as *hinge loss*, and is sometimes denoted as

$$f(s) = [1 - s]_+$$

We can rewrite the objective as

$$\min_{\mathbf{w}, b} \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{n} \sum_{i=1}^n \max [0, 1 - z_i(\mathbf{w}^T \mathbf{x}_i + b)]$$

Compare with Logistic Regression:

$$\min_{\mathbf{w}, b} \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{n} \sum_i \log [1 + e^{-z_i(\mathbf{w}^T \mathbf{x}_i + b)}]$$

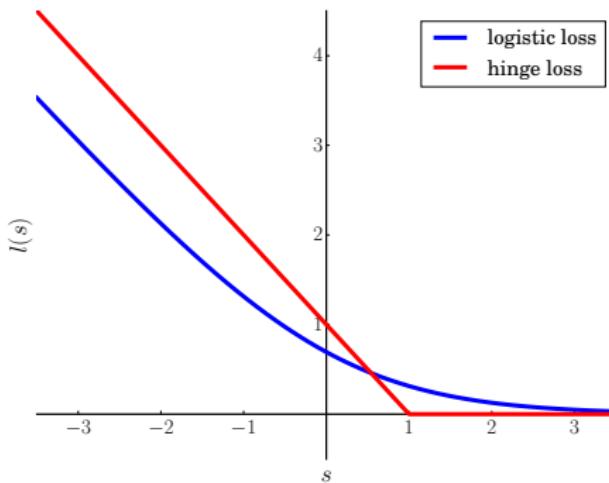
Support Vector Machines

LR: Logistic Loss

$$l(s) = \log(1 + e^{-s})$$

SVM: Hinge Loss

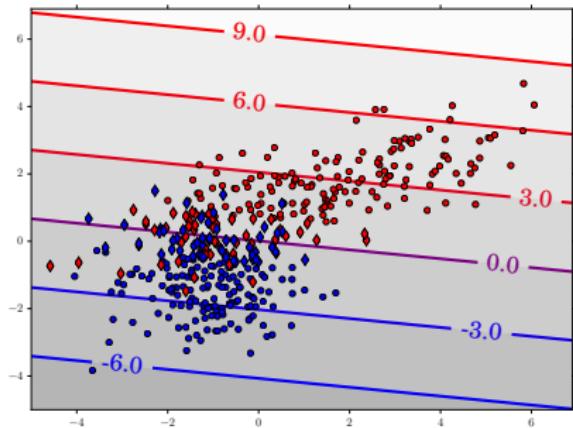
$$l(s) = \max(0, 1 - s)$$



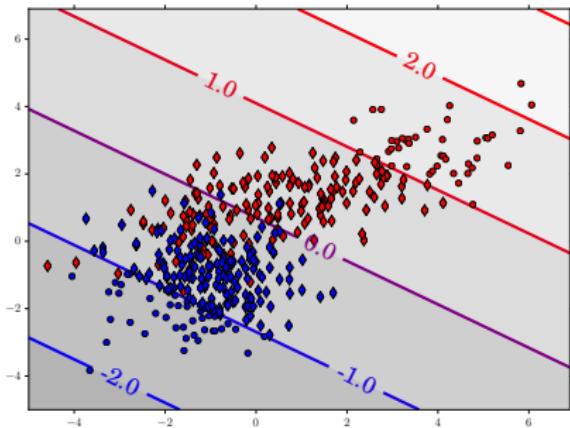
Support Vector Machines

Binary classification:

$$C = 1.0 \ (\lambda = 0.0025)$$



$$C = 0.001 \ (\lambda = 2.5)$$



Support Vector Machines

We have seen that we can obtain the separation hyperplane by solving either the primal or the dual problem

Primal

Minimize w.r.t. \mathbf{w} and b

$$\frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n [1 - z_i(\mathbf{w}^T \mathbf{x}_i + b)]_+$$

Dual

Maximize w.r.t. $\boldsymbol{\alpha}$

$$\boldsymbol{\alpha}^T \mathbf{1} - \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{H} \boldsymbol{\alpha}$$

subject to

$$0 \leq \alpha_i \leq C, \quad i = 1, \dots, n$$

$$\sum_{i=1}^n \alpha_i z_i = 0$$

Support Vector Machines

Parameters $w \in \mathbb{R}^D$, $b \in \mathbb{R}$

Scoring complexity is $O(D)$:

$$s(\mathbf{x}_t) = w^T \mathbf{x}_t + b$$

Embedding a non-linear transformation requires explicitly defining the mapping from \mathbb{R}^D to the Hilbert space \mathcal{H}

$$\Phi : \mathbb{R}^D \longrightarrow \mathcal{H}$$

Parameters $\alpha \in \mathbb{R}^N$

Scoring complexity is $O(SV)$:

$$\begin{aligned} s(\mathbf{x}_t) &= \sum_{i=1}^N \alpha_i z_i \mathbf{x}_i^T \mathbf{x}_t + b \\ &= \sum_{i|\alpha_i>0} \alpha_i z_i \mathbf{x}_i^T \mathbf{x}_t + b \end{aligned}$$

Embedding a non-linear transformation only requires **dot-products in the expanded space**

$$\Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j)$$

Support Vector Machines

If we have a function that efficiently computes the dot-products in the expanded space

$$k(\mathbf{x}_1, \mathbf{x}_2) = \Phi(\mathbf{x}_1)^T \Phi(\mathbf{x}_2)$$

then both training and scoring can be performed by using only k

Remember that, for mapping Φ , matrix H is given

$$H_{ij} = z_i z_j \Phi(x_i)^T \Phi(x_j)$$

From the definition of k :

$$H_{ij} = z_i z_j k(x_i, x_j)$$

Scoring becomes

$$s(\mathbf{x}_t) = \sum_{i=1 | \alpha_i > 0} \alpha_i z_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_t) + b = \sum_{i=1 | \alpha_i > 0} \alpha_i z_i k(\mathbf{x}_i, \mathbf{x}_t) + b$$

Support Vector Machines

Function k is called **kernel function**

A kernel function allows training a SVM in a large (even infinite) dimensional Hilbert space \mathcal{H} , without requiring to explicitly compute the mapping

Even if the expansion was finite, the complexity of the primal problem may be too large

On the other hand, the complexity of the dual problem only depends on the number of training points.

In practice, we are computing a linear separation surface in the expanded space, which corresponds to a non-linear separation surface in the original feature space

Support Vector Machines

We have already seen an example of feature expansion that provides quadratic separation surfaces, given by the mapping

$$\Phi(\mathbf{x}) = \begin{bmatrix} \text{vec}(\mathbf{x}\mathbf{x}^T) \\ \sqrt{2}\mathbf{x} \\ 1 \end{bmatrix} \quad (3)$$

We can compute

$$\Phi(\mathbf{x}_1)^T \Phi(\mathbf{x}_2) = (\mathbf{x}_1^T \mathbf{x}_2)^2 + 2\mathbf{x}_1^T \mathbf{x}_2 + 1$$

It is easy verifying that the corresponding kernel is

$$k(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1^T \mathbf{x}_2 + 1)^2$$

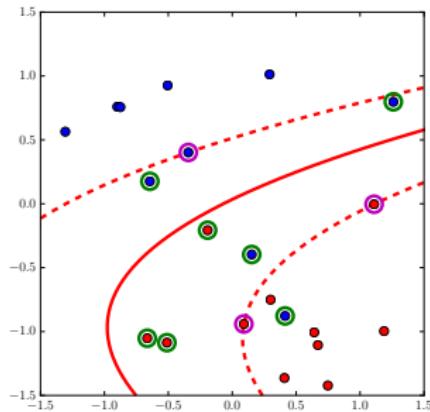
Support Vector Machines

In general, we can define the polynomial kernels of degree d as

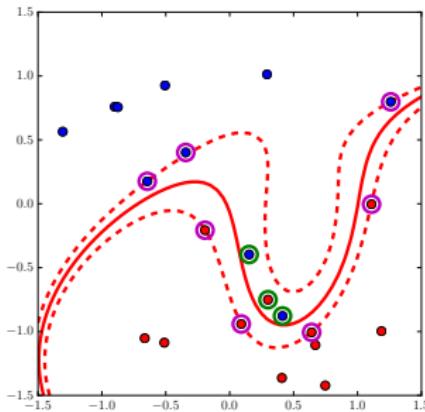
$$k(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1^T \mathbf{x}_2 + 1)^d$$

Note that the expanded space, although finite, grows very quickly

Poly — $d = 2, C = 1.0$



Poly — $d = 4, C = 10.0$



Support Vector Machines

It is clear that, if we know Φ , we can define a corresponding kernel function

In general, we want to know for which kernel function we can actually find Φ and \mathcal{H}

Mercer's condition provides a sufficient condition for k to define a dot-product in an expanded space

Support Vector Machines

Let $k(\mathbf{u}, \mathbf{v})$ be a symmetrical function in L_2

If, for all functions $g(\mathbf{u})$ such that

$$\int g(\mathbf{u})^2 d\mathbf{u} < \infty$$

we have that

$$\int k(\mathbf{u}, \mathbf{v})g(\mathbf{u})g(\mathbf{v})d\mathbf{u}d\mathbf{v} > 0$$

then k defines a dot-product in some expanded space

This condition does not tell us how to construct suitable kernels

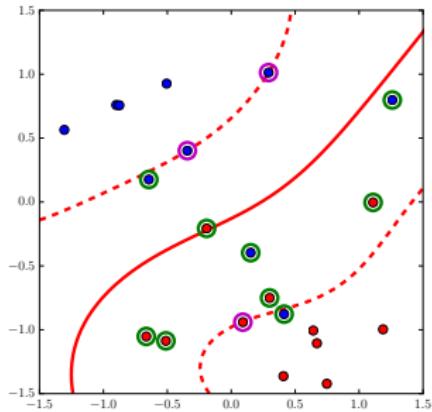
In general, we can make use of well known kernels, or use rules to combine kernel functions that guarantee that the combination is still a kernel (e.g., summing kernel functions)

Support Vector Machines

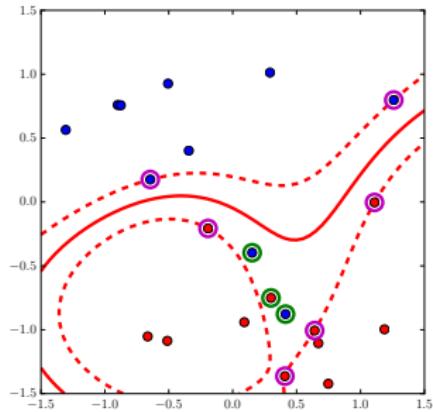
Another example of kernel function associated to an infinite-dimensional mapping (Gaussian Radial Basis Function kernel):

$$k(\mathbf{x}_1, \mathbf{x}_2) = e^{-\gamma \|\mathbf{x}_1 - \mathbf{x}_2\|^2}$$

RBF — $\gamma = 0.5, C = 1.0$



RBF — $\gamma = 0.5, C = 100.0$

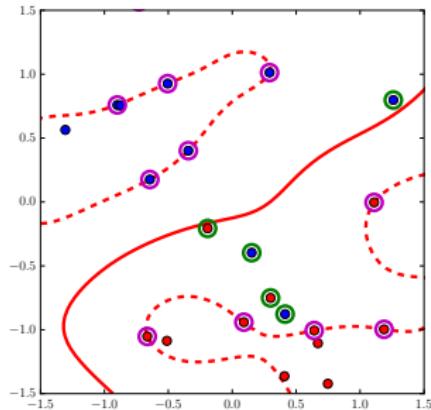


Support Vector Machines

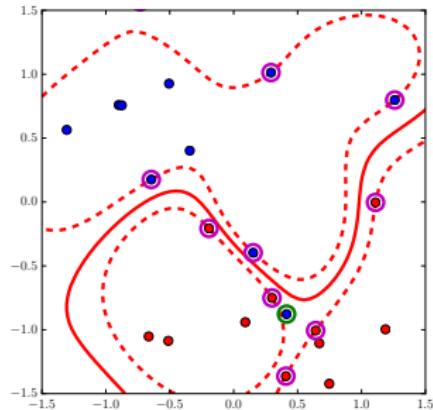
Another example of kernel function associated to an infinite-dimensional mapping (Gaussian Radial Basis Function kernel):

$$k(\mathbf{x}_1, \mathbf{x}_2) = e^{-\gamma \|\mathbf{x}_1 - \mathbf{x}_2\|^2}$$

RBF — $\gamma = 2.0, C = 1.0$



RBF — $\gamma = 2.0, C = 100.0$



Support Vector Machines

Another example of kernel function associated to an infinite-dimensional mapping (Gaussian Radial Basis Function kernel):

$$k(\mathbf{x}_1, \mathbf{x}_2) = e^{-\gamma \|\mathbf{x}_1 - \mathbf{x}_2\|^2}$$

The kernel depends on the distance of the points

“Similar” (i.e. closer) points have higher kernel value

Support Vector Machines

Another example of kernel function associated to an infinite-dimensional mapping (Gaussian Radial Basis Function kernel):

$$k(\mathbf{x}_1, \mathbf{x}_2) = e^{-\gamma \|\mathbf{x}_1 - \mathbf{x}_2\|^2}$$

γ defines the width of the kernel

- Small γ : the kernel is wide, a S.V. influences most other points
- Large γ : the kernel is narrow, a S.V. has very small influence on points that are not close. Assuming⁴ $b = 0$, we obtain the same classification rule of 1-NN⁵

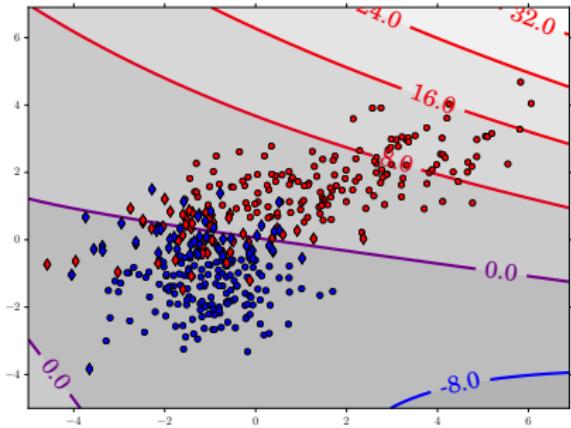
⁴Remember that in general we assign labels comparing scores with a threshold that depends on our application, and is not necessarily 0

⁵In practice we should not use SVMs as replacement for K-NN, since we will incur into numerical problems due to test points all having scores $s \approx 0$

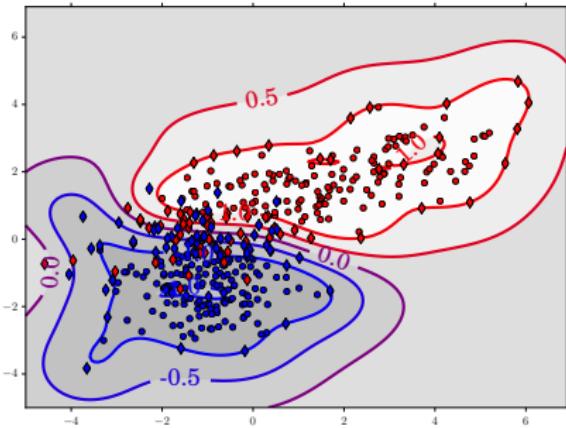
Support Vector Machines

Binary classification:

Poly — $d = 2, C = 1.0$



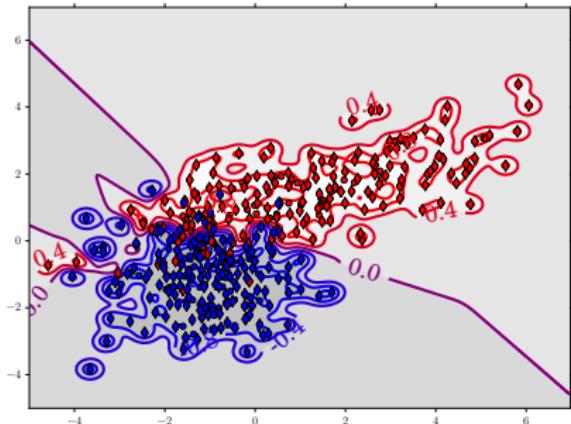
RBF — $\gamma = 0.5, C = 1.0$



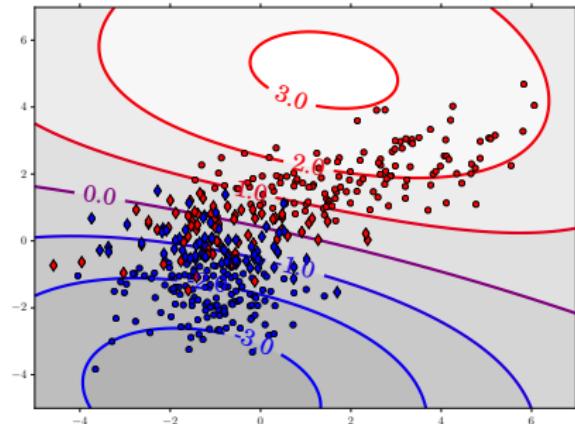
Support Vector Machines

Binary classification:

RBF (no bias) — $\gamma = 10, C = 1.0$



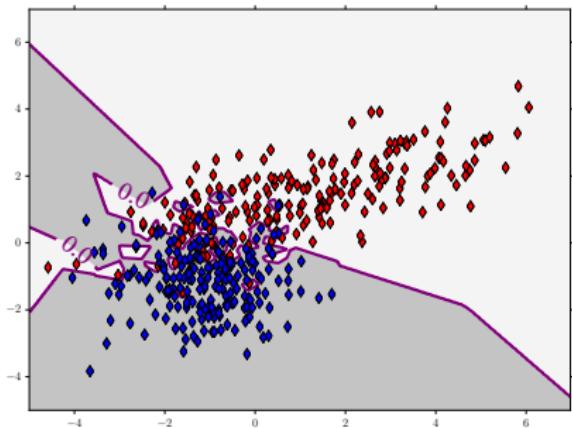
RBF (no bias) — $\gamma = 0.02, C = 1.0$



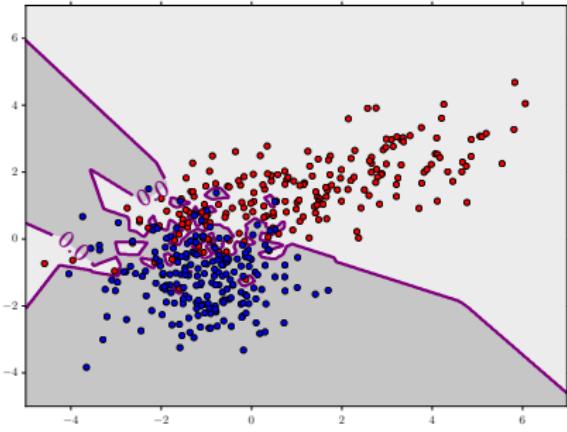
Support Vector Machines

Binary classification:

RBF (no bias) — $\gamma = 200, C = 1.0$



1-NN



Support Vector Machines

Practical considerations:

- We need to take care in selecting the appropriate kernel
- Kernels often include hyper-parameters (e.g. γ for RBF)
 - Cross-validation can help selection of good values
- We need to select good values of the coefficient C
 - Again, cross-validation can help

Support Vector Machines

Practical considerations:

- In some cases linear classifiers are sufficient
 - Our feature set is of sufficient high dimension to make our classes almost linearly separable

Fast algorithms for training the primal problem are available

Furthermore, linear SVM scoring can be implemented as a simple dot-product

- SVM training is not invariant under affine transformations
 - Feature pre-processing can be relevant
 - Often it is useful to center and whiten data (see the Logistic Regression slides)

Support Vector Machines

Practical considerations:

- SVM scores have no probabilistic interpretation
 - Score post-processing can be applied to estimate class posterior probabilities (e.g. we can train a generative or discriminative probabilistic model over SVM scores using an held-out dataset)
- Finally, we have to take care when training with highly unbalanced datasets. In this case, rebalancing the cost of different errors may be beneficial.

Support Vector Machines

MNIST — Comparison of average pairwise EER for different kernels (raw features)

Kernel	$C = 10$	$C = 1$	$C = 0.1$	$\text{LogReg } (\lambda = 1e^{-3})$
Linear	1.6%	1.3%	1.1%	1.2%
Poly ($d = 2$)	0.3%	0.3%	0.3%	$0.9\%^6$
Poly ($d = 3$)	0.4%	0.4%	0.4%	—
RBF ($\gamma = 0.2$)	0.7%	0.7%	0.9%	—
RBF ($\gamma = 1.0$)	0.5%	0.5%	0.6%	—
RBF ($\gamma = 2.0$)	0.1%	0.1%	0.1%	—

⁶Quadratic expansion of PCA-reduced features

Support Vector Machines

SVMs provide good performance for binary classification problems

Difficult to extend to multiclass problems

Some possibilities:

- One-versus-all training
- Train over all possible pairs

In both cases we need some voting scheme to decide the actual class

Support Vector Machines

Some possibilities (cont.):

- Introduce an objective function that maximizes the margin between each class and all the others. For example, (we ignore the bias terms)

$$\min_{\mathbf{W}} \frac{1}{2} \|\mathbf{W}\|_2^2 + C \sum_{i=1}^n \max_{y'} [(\mathbf{w}_{y'} - \mathbf{w}_{z_i})^T \mathbf{x}_i + \delta_{y', z_i}]$$

Compare with multiclass logistic regression

$$\min_{\mathbf{W}} \frac{\lambda}{2} \|\mathbf{W}\|_2^2 + \frac{1}{n} \sum_i \log \left(\sum_{y'} e^{(\mathbf{w}_{y'} - \mathbf{w}_{z_i}^T)^T \mathbf{x}_i} \right)$$

Support Vector Machines

Kernel methods are not restricted to SVMs

- Logistic Regression can be extended to incorporate kernels
- In general, we can apply the kernel tricks to problems which depend only on dot-products
 - Kernel PCA
 - Gaussian Processes
 - ...

Gaussian Mixture Models

Sandro Cumani

sandro.cumani@polito.it

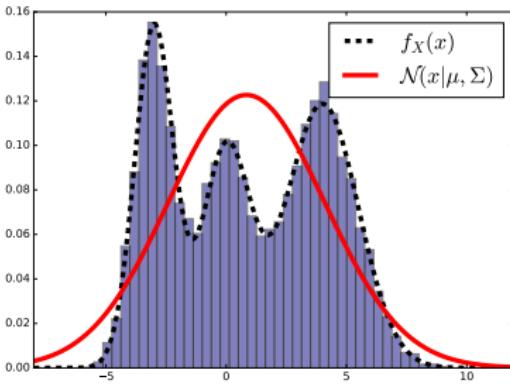
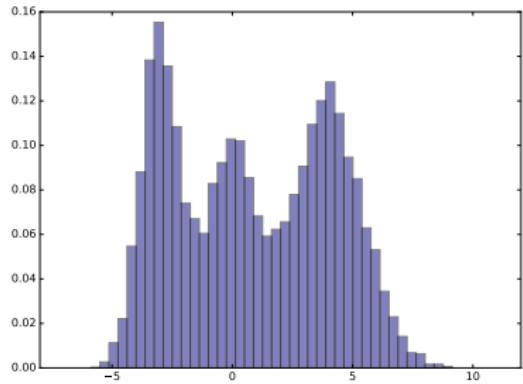
Politecnico di Torino

Gaussian Mixture models

We have seen that we can solve classification problems by building generative models that describe the distribution of samples

The Gaussian classifier is an example that assumes that class-conditional distributions are Gaussian

In many cases, however, the assumption can be quite inaccurate



Density estimation

Different distributions may be used in such cases

Depending on the task, we may be able to identify a reasonably good family of distributions

Gaussian Mixture Models are an alternative to model a generic distribution

They allow approximating any sufficiently regular distribution to a desired degree

Of course, since we are estimating the density from data, we require a sufficient amount of data to obtain good estimates

Density estimation

The use of GMMs is not restricted to classification

GMMs can be employed also in other tasks that require estimating a population density

As we will see, they also allow to solve different kind of problems

For example, GMMs provide an alternative to K-means for clustering

Gaussian Mixture Models

We have already encountered an example of GMM

Let's consider again the Gaussian classifier

The samples of each class are modeled by a Gaussian density

$$f_{X|C}(\mathbf{x}|c) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

To compute class posterior probabilities we had to compute the marginal density $f_X(\mathbf{x})$

If the class prior probabilities are $P(C = c) = \pi_c, c = 1 \dots K$, then $f_X(\mathbf{x})$ is given by

$$f_X(\mathbf{x}) = \sum_{c=1}^K f_{X|C}(\mathbf{x}|c)P(C = c) = \sum_{c=1}^K \pi_c \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \quad (1)$$

Gaussian Mixture Models

Expression (1) is an example of a K -components Gaussian Mixture Model:

$$X \sim GMM(\boldsymbol{M}, \boldsymbol{\Sigma}, \boldsymbol{\Pi}) \implies f_X(\boldsymbol{x}) = \sum_{c=1}^K \pi_c \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

More in general, a Gaussian Mixture Model is a density model obtained as a weighted combination of Gaussians

$$f_X(\boldsymbol{x}) = \sum_{c=1}^K w_c \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

Gaussian Mixture Models

The distribution parameters are the component means

$$\boldsymbol{M} = [\boldsymbol{\mu}_1 \dots \boldsymbol{\mu}_K]$$

the component covariances

$$\boldsymbol{S} = [\boldsymbol{\Sigma}_1 \dots \boldsymbol{\Sigma}_K]$$

and the weights

$$\boldsymbol{w} = [w_1 \dots w_K]$$

Remember that, for f_X to be a density, we need that its integral is equal to 1. Integrating w.r.t. \mathbf{x} we have:

$$\int f_X(\mathbf{x}) = \int \sum_{c=1}^K w_c \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) d\mathbf{x} = \sum_{c=1}^K w_c = 1$$

i.e., the weights must sum to 1

Gaussian Mixture Models

Given a dataset $\mathcal{D} = [x_1 \dots x_n]$ we can thus assume that the samples have been **independently** generated by a GMM

We assume that R.V.s describing the samples X_i are i.i.d., with $X_i \sim X \sim GMM(\mathbf{M}, \mathbf{S}, \mathbf{w})$

Note: we are considering a density estimation problem

If we are using GMMs for classification \mathcal{D} may correspond to the samples of a given class — however, in the following we do not assume any specific task, so that \mathcal{D} is just a set of samples that we want to model by means of a GMM

In particular, we consider the dataset \mathcal{D} as **unlabeled**

Gaussian Mixture Models

As we did with the Gaussian density, we can resort to Maximum Likelihood to estimate the model parameters of the GMM that best describes the dataset \mathcal{D}

In contrast with the Gaussian model, ML estimation for GMMs is an ill-posed problem

Indeed, as long as we have more than 1 component, we can devise degenerate solutions for which the likelihood is not bounded above

Care has to be taken to avoid these pathological solutions

In practice, the ML approach, combined with heuristics to avoid degeneracy, provides good density estimates

Gaussian Mixture Models

We can write the likelihood for the model parameters $\theta = [\mathbf{M}, \mathbf{S}, \mathbf{w}]$ as

$$\begin{aligned}\mathcal{L}(\theta) &= \prod_{i=1}^n f_{X_i}(\mathbf{x}_i) = \prod_{i=1}^n GMM(\mathbf{x}_i | \mathbf{M}, \mathbf{S}, \mathbf{w}) \\ &= \prod_{i=1}^n \left(\sum_{c=1}^K w_c \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \right)\end{aligned}$$

and the corresponding log-likelihood

$$\ell(\theta) = \log \mathcal{L}(\theta) = \sum_{i=1}^n \log \left(\sum_{c=1}^K w_c \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \right)$$

Gaussian Mixture Models

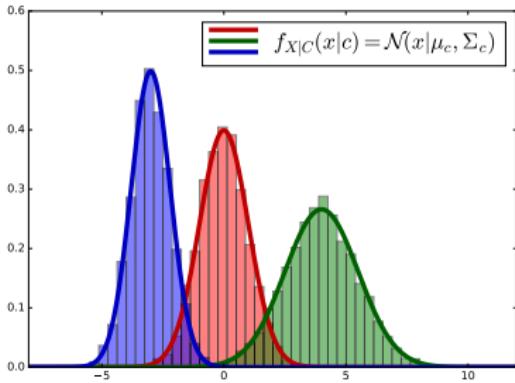
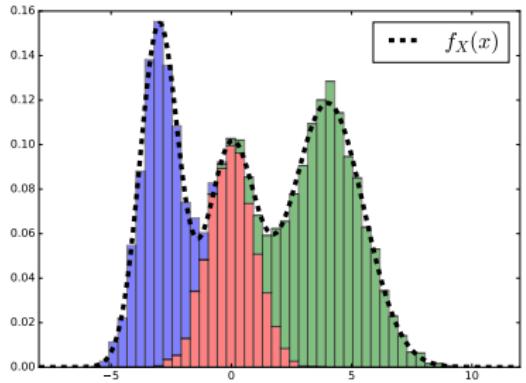
A GMM can be interpreted as the marginal of a joint distribution of data points and corresponding clusters

$$f_{\mathbf{X}_i}(\mathbf{x}_i) = \sum_{c=1}^K f_{\mathbf{X}_i|C_i}(\mathbf{x}_i|c)P(C_i=c) = \sum_{c=1}^K w_c \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

Although our training set cannot be well modeled by a Gaussian distribution, we can imagine that the set can be partitioned into subsets (**components** or **clusters**), in such a way that the distribution of the points of each component can be modeled by a Gaussian p.d.f.

If we knew the component responsible for each sample (i.e. its cluster label), we could estimate the parameters of each Gaussian by ML from the points of each cluster

Gaussian Mixture Models



Gaussian Mixture Models

Unfortunately, in general the clusters are **unknown**

We treat cluster membership as an **unobserved (latent)** random variables¹

Intuitively, we want to estimate both cluster assignments and model parameters as to maximize the *marginal* distribution of the data

¹Note that the model is **not identifiable**: for example, exchanging any two components results in the same marginal likelihood

Gaussian Mixture Models

Let's consider a set of GMM parameters $\theta = (\mathbf{M}, \mathcal{S}, \mathbf{w})$

The GMM defines a **joint density** of components (the clusters) and patterns. The density for sample x_i and component c is:

$$f_{X_i, C_i}(\mathbf{x}_i, c) = w_c \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

We can compute cluster (component) **posterior probabilities**:

$$\begin{aligned}\gamma_{c,i} &= P(C_i = c | X_i = \mathbf{x}_i) = \frac{f_{X_i, C_i}(\mathbf{x}_i, c)}{f_{X_i}(\mathbf{x}_i)} \\ &= \frac{w_c \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)}{\sum_{c'} w_{c'} \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_{c'}, \boldsymbol{\Sigma}_{c'})}\end{aligned}$$

$\gamma_{c,i}$ are also called **responsibilities**

Gaussian Mixture Models

As a first approximation, we might decide to assign a point to the cluster c with highest posterior probability $P(C_i = c | X_i = \mathbf{x}_i)$

We thus **associate** a cluster label

$$c_i^* = \arg \max_c P(C_i = c | X_i = \mathbf{x}_i)$$

to each sample

Given the cluster assignments, we can then estimate by ML the new GMM parameters $\theta^{new} = (\mathbf{M}^{new}, \mathbf{S}^{new}, \mathbf{w}^{new})$

Gaussian Mixture Models

We treat the cluster assignments as if they were known class labels

The log-likelihood is similar to that of a (multivariate) Gaussian classifier:

$$\begin{aligned}\ell(\theta) &= \sum_{i=1}^n [\log f_{X_i|C_i}(\mathbf{x}_i|c_i^*) + \log P(C_i = c_i^*)] \\ &= \sum_{i=1}^n [\log \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_{c_i^*}, \boldsymbol{\Sigma}_{c_i^*})] + \sum_{i=1}^n [\log w_{c_i^*}]\end{aligned}$$

and corresponds to a sum of two terms that depend on different subsets of the parameters

$$\ell(\theta) = \ell_{\mathcal{N}}(\mathbf{M}, \mathbf{S}) + \ell_{\mathcal{C}}(\mathbf{w})$$

Gaussian Mixture Models

We can observe that the first term

$$\ell_{\mathcal{N}}(\mathbf{M}, \mathcal{S}) = \sum_{i=1}^n \left[\log \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_{c_i^*}, \boldsymbol{\Sigma}_{c_i^*}) \right]$$

corresponds to the log-likelihood of a (multivariate) Gaussian classification model, where the class labels are assumed to be the estimated c_i^* .

Let N_c be the number of samples for which $c_i^* = c$. The solution for $\boldsymbol{\mu}_c$ and $\boldsymbol{\Sigma}_c$ is thus

$$\boldsymbol{\mu}_c^* = \frac{1}{N_c} \sum_{i|c_i^*=c} \mathbf{x}_i , \quad \boldsymbol{\Sigma}_c^* = \frac{1}{N_c} \sum_{i|c_i^*=c} (\mathbf{x}_i - \boldsymbol{\mu}_c^*)(\mathbf{x}_i - \boldsymbol{\mu}_c^*)^T$$

Gaussian Mixture Models

The second term

$$\ell_C(\mathbf{w}) = \sum_{i=1}^n [\log w_{c_i^*}] = \sum_{c=1}^K \sum_{i|c_i^*=c} \log w_c$$

corresponds to the log-likelihood of a categorical model with parameters w_c . The ML solution is thus

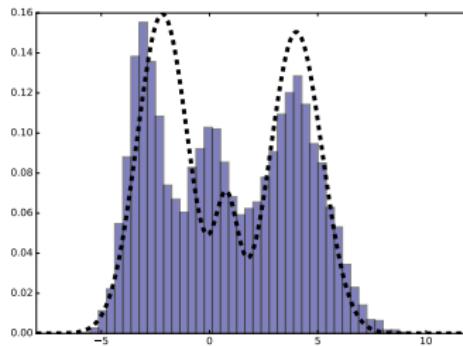
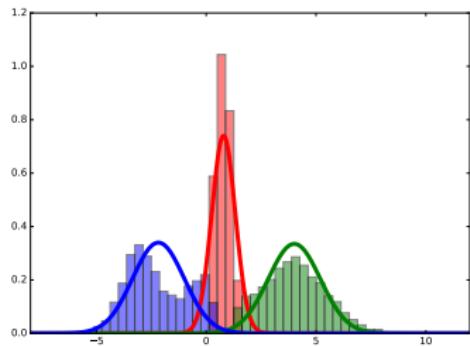
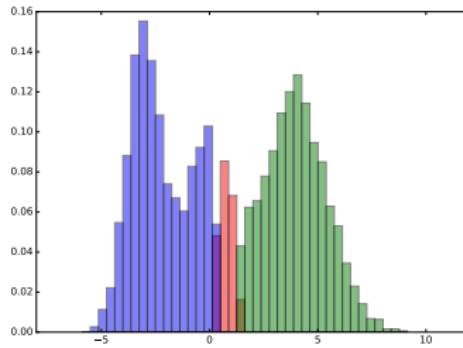
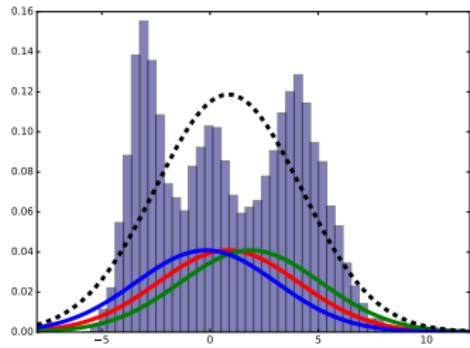
$$w_c^* = \frac{N_c}{\sum_{c=1}^K N_c}$$

Gaussian Mixture Models

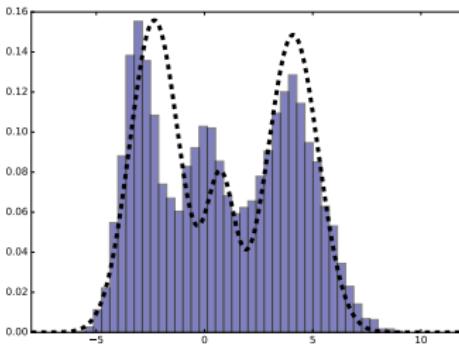
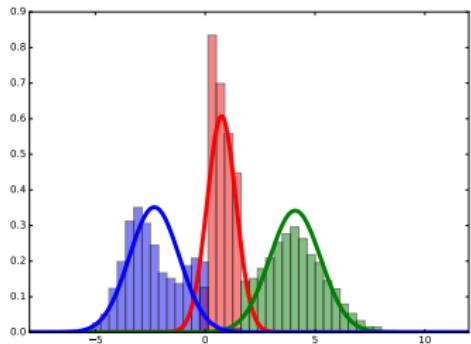
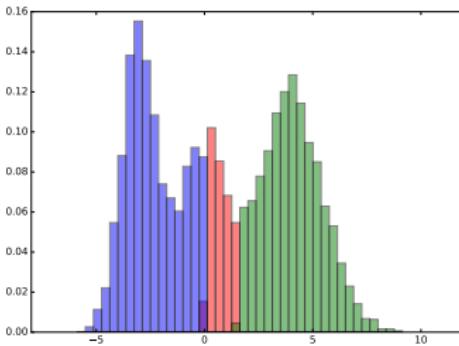
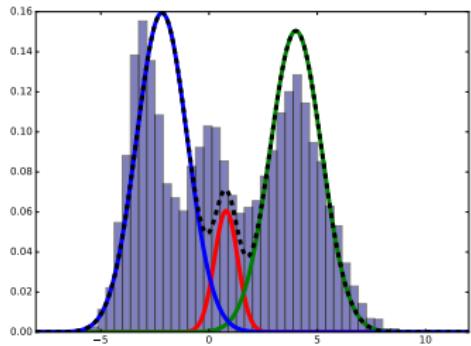
We could then obtain the updated set of model parameters $\theta^{new} = (\mathbf{M}^*, \mathbf{S}^*, \mathbf{w}^*)$

We could iterate the process by computing new cluster assignments using θ^{new} , and using the updated assignments to update once again the model parameters, stopping when some criterion is met

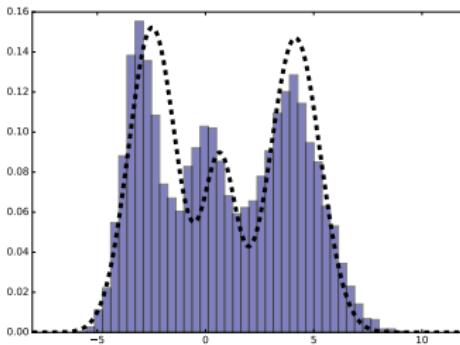
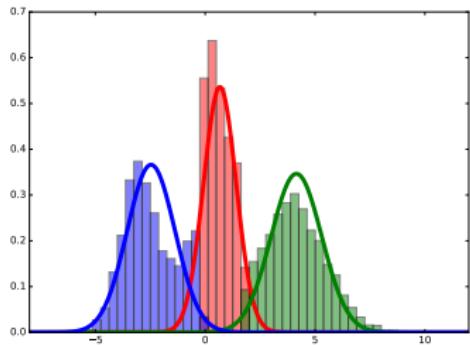
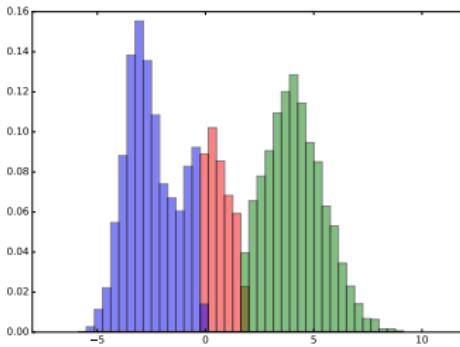
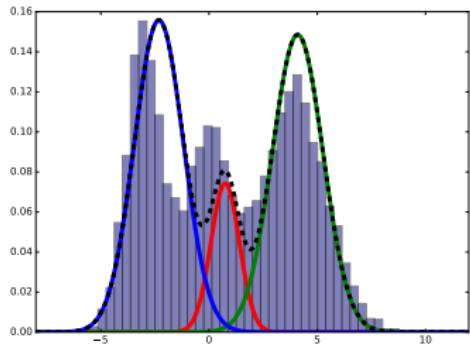
Gaussian Mixture Models



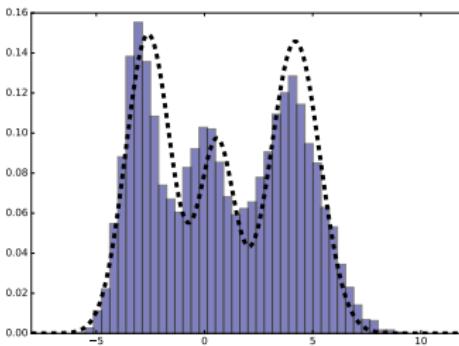
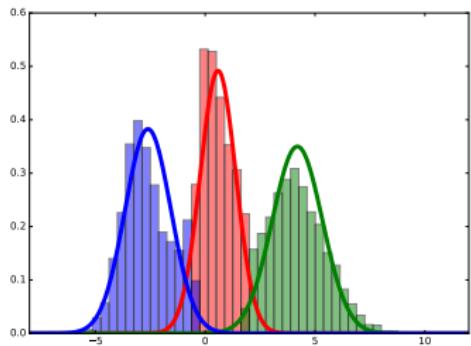
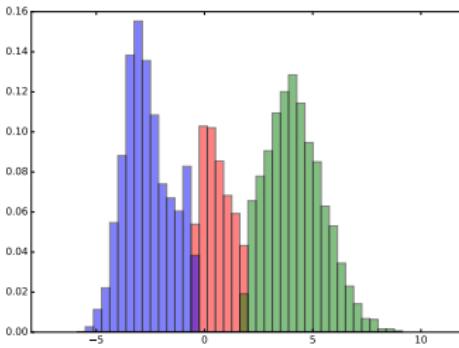
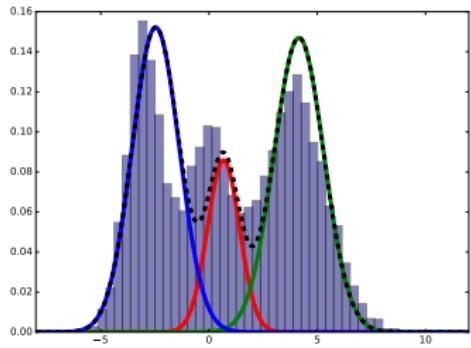
Gaussian Mixture Models



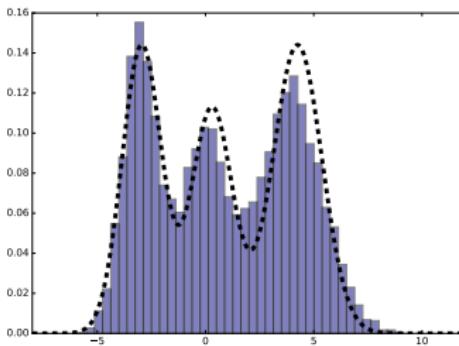
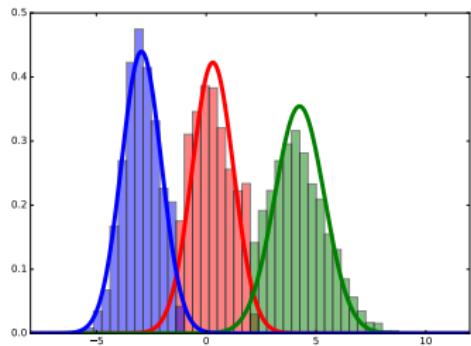
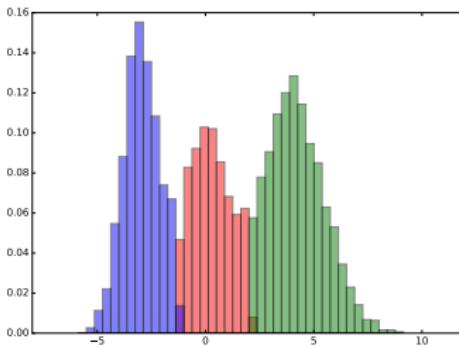
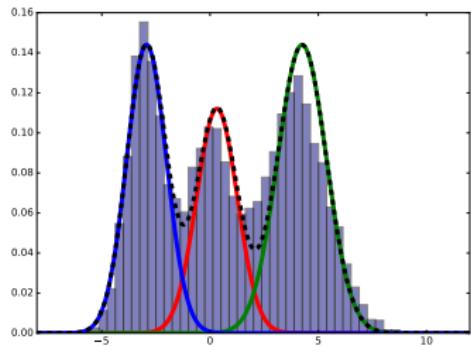
Gaussian Mixture Models



Gaussian Mixture Models



Gaussian Mixture Models



Gaussian Mixture Models

Problem: we form hard **clusters** — a point is assigned to **one, and only one**, component of the GMM

If $P(C_i = c_i^* | X_i = \mathbf{x}_i) \approx 1$ then we can correctly assume that the point belongs to that component

However, when $P(C_i = c_1 | X_i = \mathbf{x}_i) \approx P(C_i = c_2 | X_i = \mathbf{x}_i)$ we are making a crude approximation: **both c_1 and c_2 might have been responsible** for the generation of \mathbf{x}_i

In general, the algorithm we discussed is not maximizing the likelihood of the observed samples \mathbf{x}_i

Gaussian Mixture Models

We will shortly see a method to estimate a local maximum of the likelihood

However, let's still consider hard-assignments

Let's also assume that we fix the covariance matrices of our GMM to $\Sigma_c = \mathbf{I}$

We also fix the weights as $w_c = \frac{1}{K}$

In this case cluster assignment corresponds to the rule

$$c_i^* = \arg \max_c P(C_i = c | X_i = \mathbf{x}_i) = \arg \min_c \|\mathbf{x}_i - \boldsymbol{\mu}_c\|^2$$

Our algorithm becomes

- Compute the component or cluster c_i^* whose centroid $\mu_{c_i^*}$ is closest to our point and assign x_i to that cluster
- Re-estimate the cluster centroids from the given points, and iterate until convergence

This is the **K-Means clustering algorithm**

GMMs can also be applied to **clustering** tasks as a **generalization of K-Means**

Gaussian Mixture Models

The algorithm we considered can be extended to handle soft assignments

We will see that a point is not completely associated to a single Gaussian component, but contributes to the estimation of different components according to its cluster (component) posterior probability

Gaussian Mixture Models

Consider the log-likelihood for our data (we now make explicitly the dependency on the model parameters in the conditional densities):

$$\sum_{i=1}^n \log f_{X_i}(\mathbf{x}_i | \boldsymbol{\theta}) = \sum_{i=1}^N \log \sum_{c=1}^K w_c \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

Let's take the gradient with respect to $\boldsymbol{\mu}_c$

$$\begin{aligned} \mathbf{0} &= - \sum_i \frac{w_c \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)}{\sum_{c'} w_{c'} \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_{c'}, \boldsymbol{\Sigma}_{c'})} \boldsymbol{\Sigma}_c (\mathbf{x}_i - \boldsymbol{\mu}_c) \\ &= - \sum_i \gamma_{c,i} \boldsymbol{\Sigma}_c (\mathbf{x}_i - \boldsymbol{\mu}_c) \end{aligned} \tag{2}$$

which gives

$$\boldsymbol{\mu}_c = \frac{\sum_i \gamma_{c,i} \mathbf{x}_i}{\sum_i \gamma_{c,i}} \tag{3}$$

Gaussian Mixture Models

Observe that the responsibilities $\gamma_{c,i}$ depend on μ_c . If we knew the responsibilities we could compute μ_c as in (3)

We can interpret (3) as a **weighted** empirical mean. The weight of each sample is the corresponding **responsibility**.

The terms

$$N_c = \sum_{i=1}^N \gamma_{c,i}$$

and

$$\mathbf{F}_c = \sum_{i=1}^N \gamma_{c,i} \mathbf{x}_i$$

are also called **zero and first order statistics**

Note that we are **summing over all samples** in \mathcal{D}

Gaussian Mixture Models

We can adopt a similar strategy for the covariance matrix, obtaining

$$\boldsymbol{\Sigma}_c = \frac{1}{N_c} \sum_i \gamma_{c,i} (\mathbf{x}_i - \boldsymbol{\mu}_c) (\mathbf{x}_i - \boldsymbol{\mu}_c)^T = \frac{1}{N_c} \sum_i \gamma_{c,i} \mathbf{x}_i \mathbf{x}_i^T - \boldsymbol{\mu}_c \boldsymbol{\mu}_c^T$$

The terms

$$\mathbf{S}_c = \sum_i \gamma_{c,i} \mathbf{x}_i \mathbf{x}_i^T$$

are also called **second order statistics**

The weights can be re-estimated as

$$w_c = \frac{N_c}{N}$$

where N is the number of samples $N = \sum_{c=1}^K N_c$

Gaussian Mixture Models

Since we are not given $\gamma_{c,i}$, we can follow the same procedure we used for hard assignments:

- Given θ , we estimate the responsibilities, i.e. the cluster or component posterior probabilities

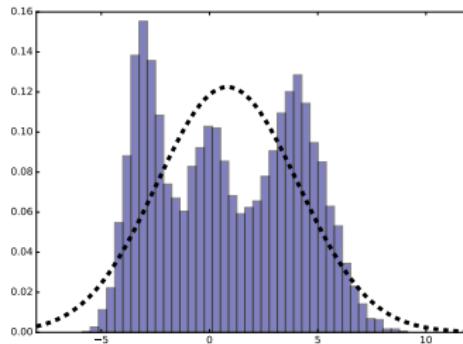
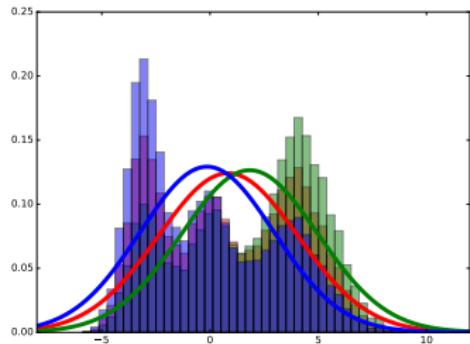
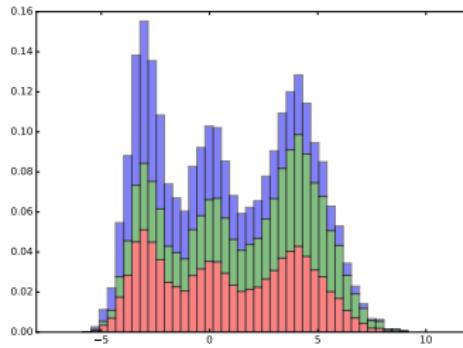
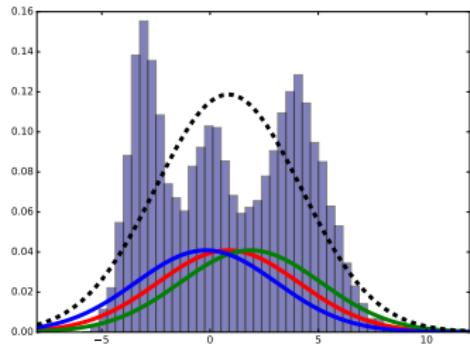
$$\gamma_{c,i} = P(C_i = c | \mathbf{X}_i = \mathbf{x}_i)$$

for each sample of our dataset \mathcal{D}

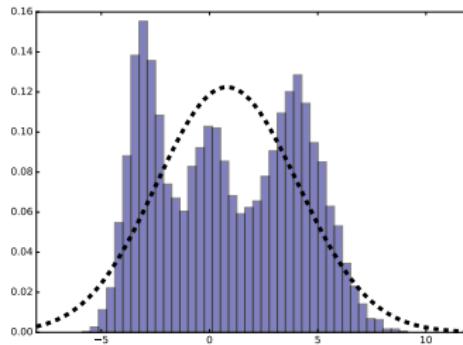
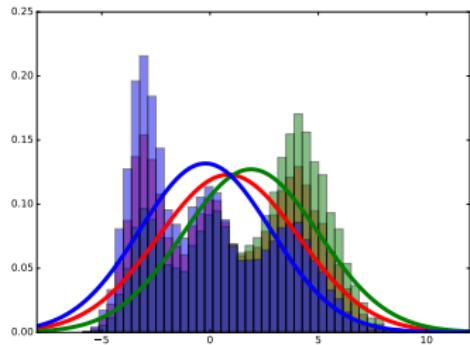
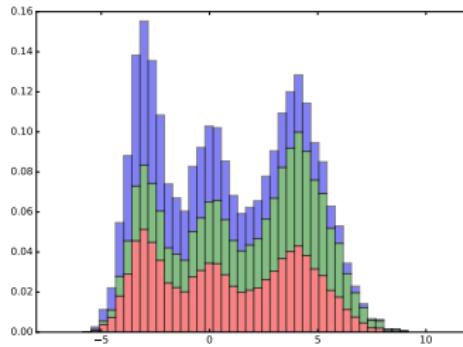
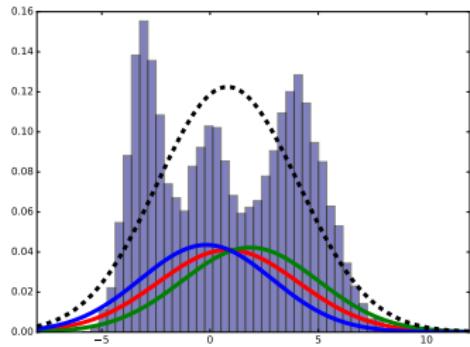
- Given the responsibilities, we re-estimate the GMM parameters θ using the previous expressions

As we will shortly see, this procedure is a particular instance of an algorithm known as **Expectation-Maximization**

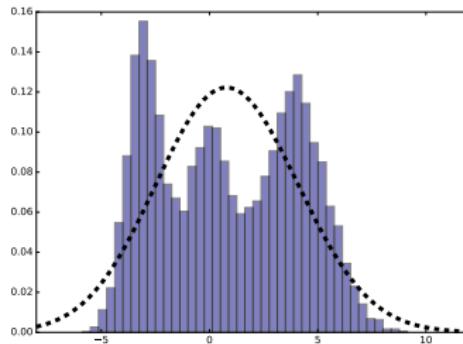
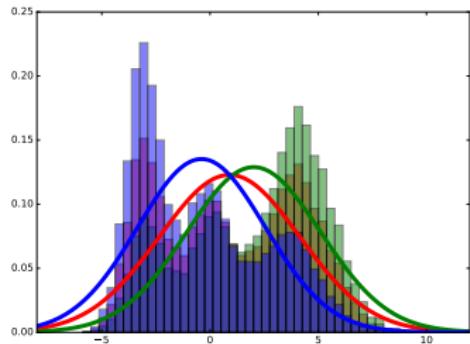
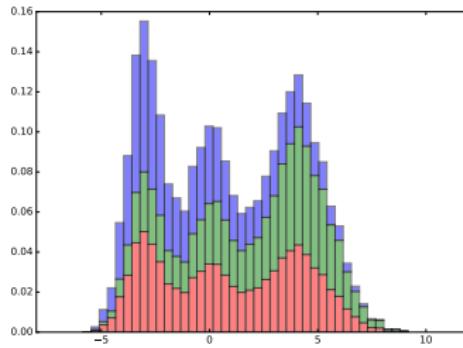
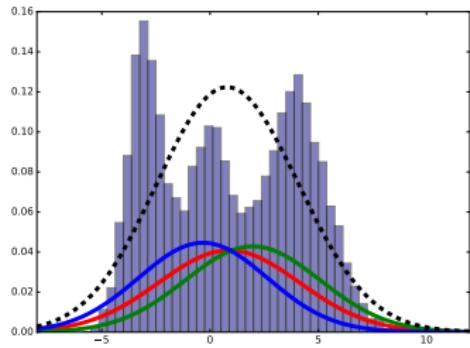
Gaussian Mixture Models



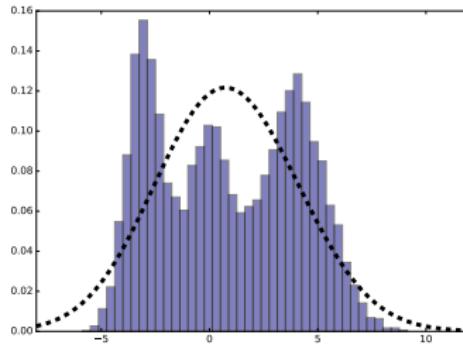
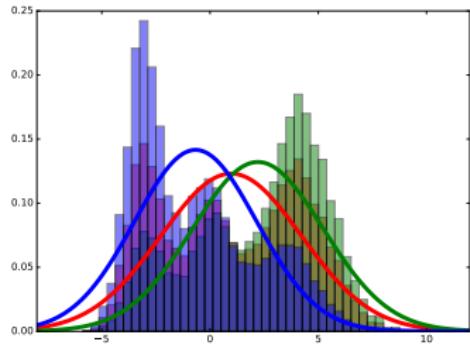
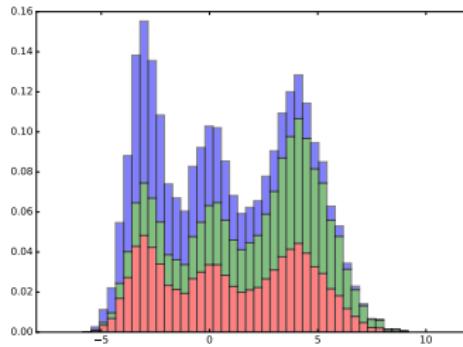
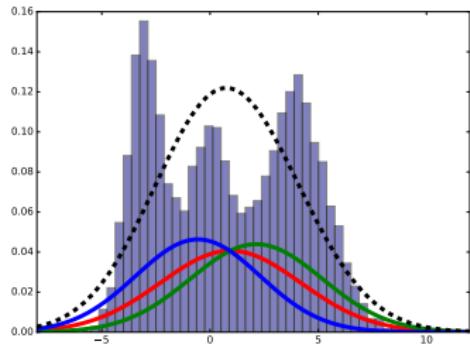
Gaussian Mixture Models



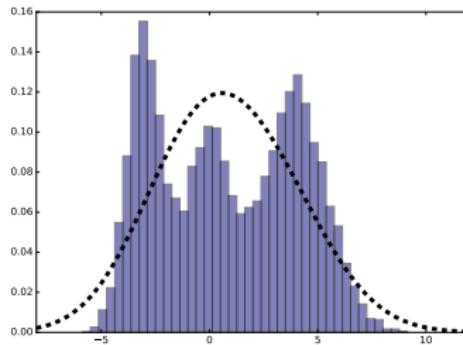
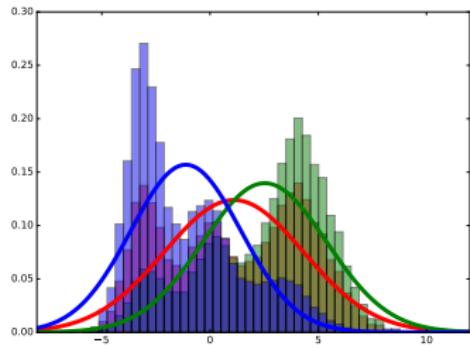
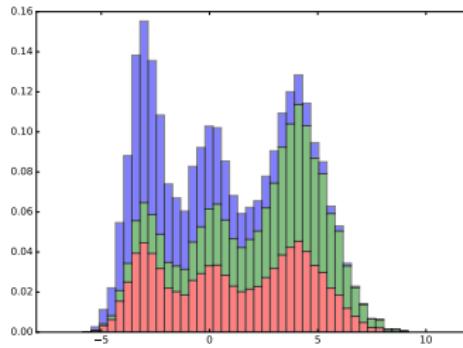
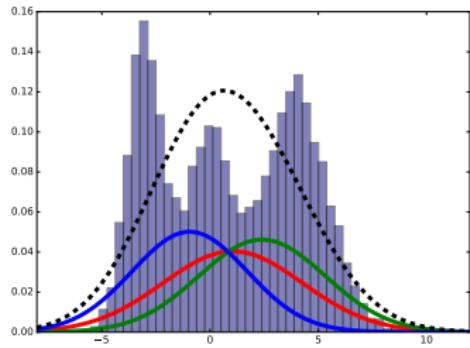
Gaussian Mixture Models



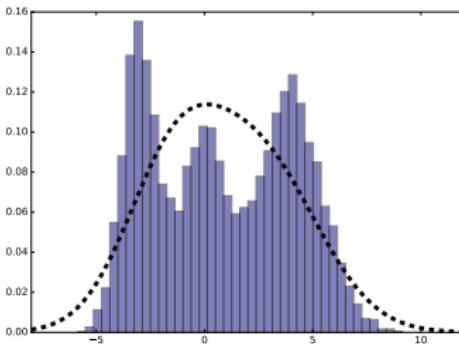
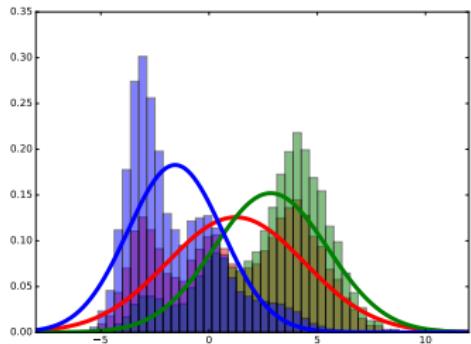
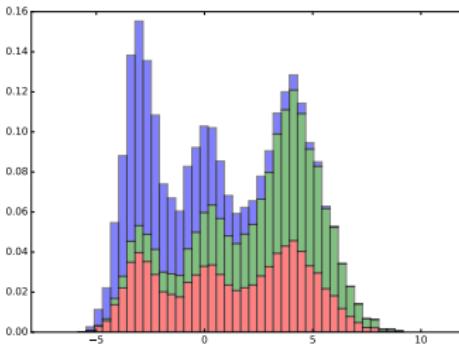
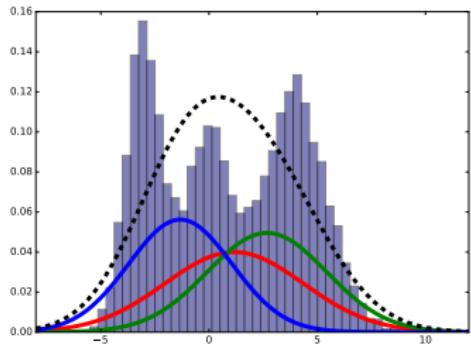
Gaussian Mixture Models



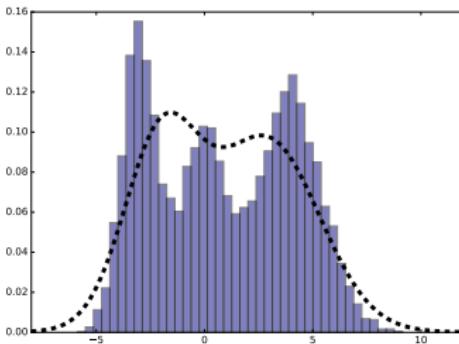
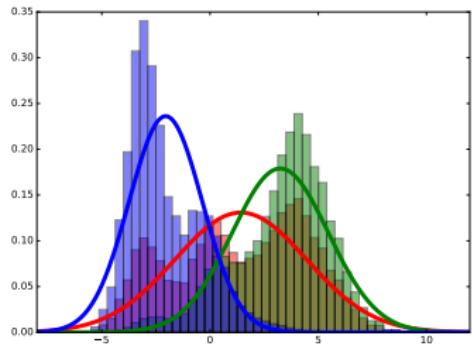
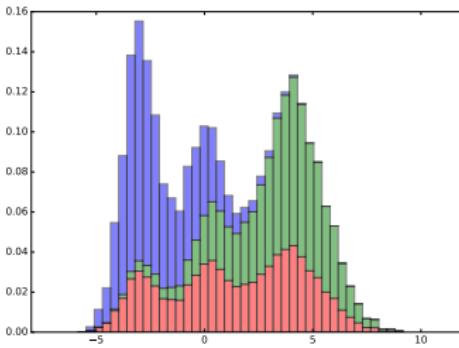
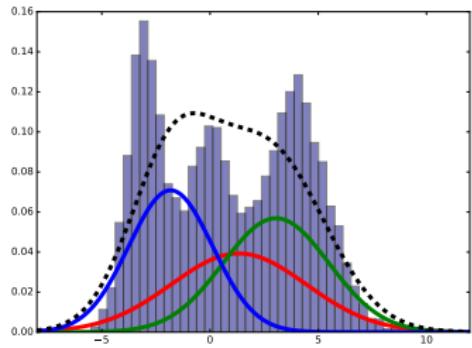
Gaussian Mixture Models



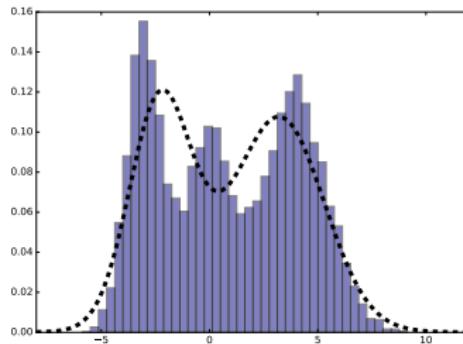
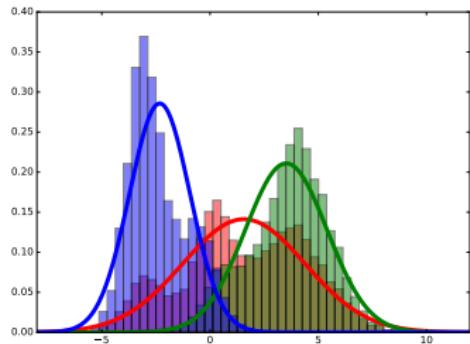
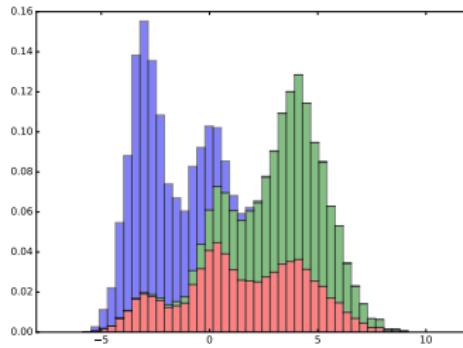
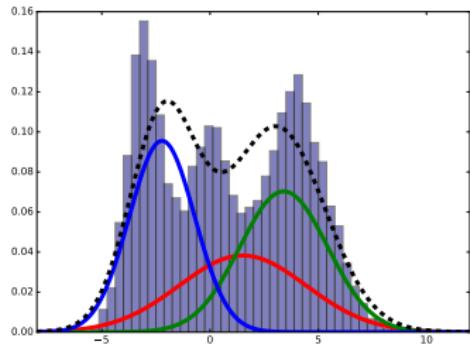
Gaussian Mixture Models



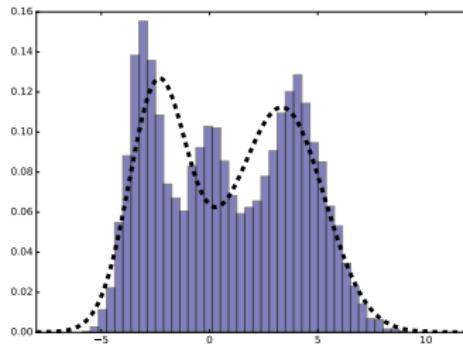
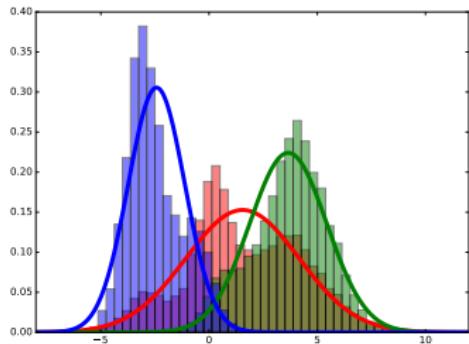
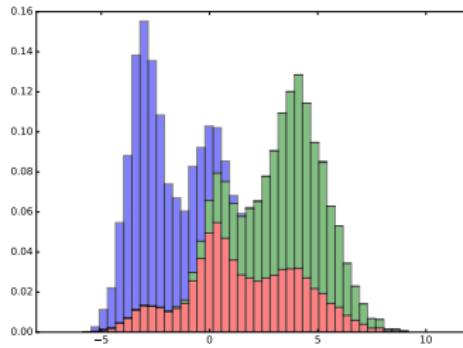
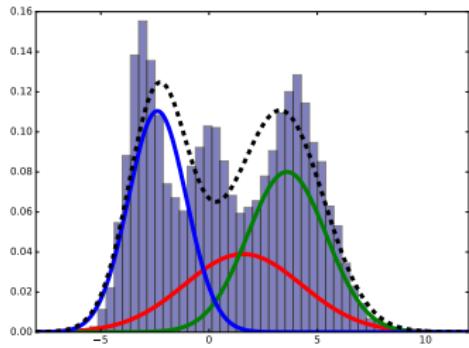
Gaussian Mixture Models



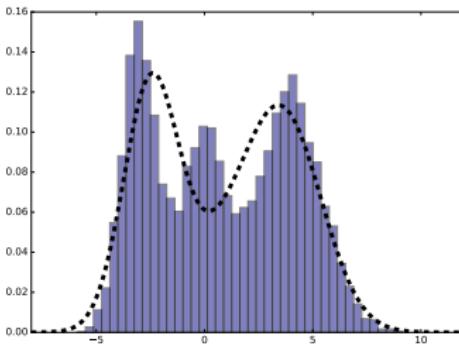
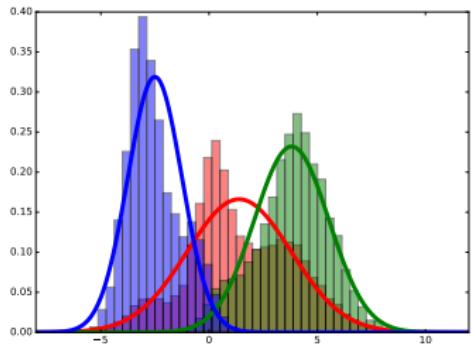
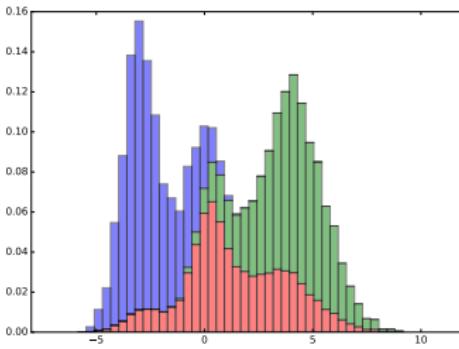
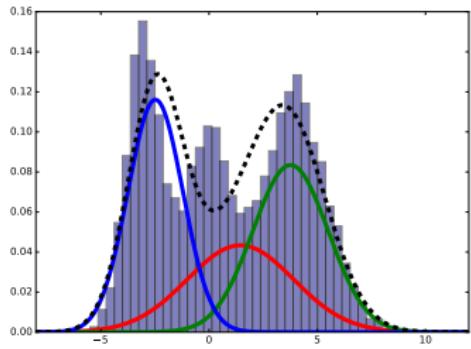
Gaussian Mixture Models



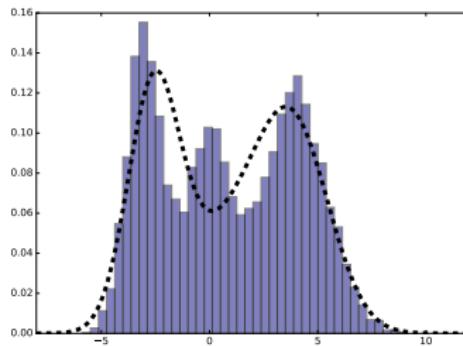
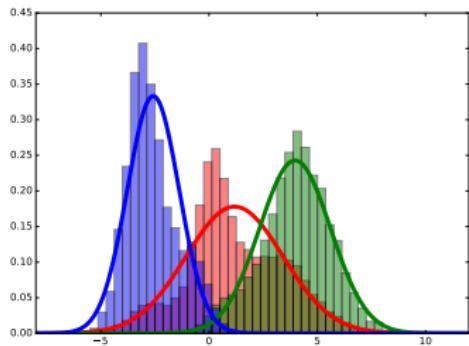
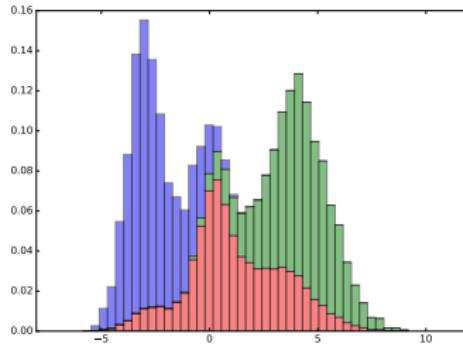
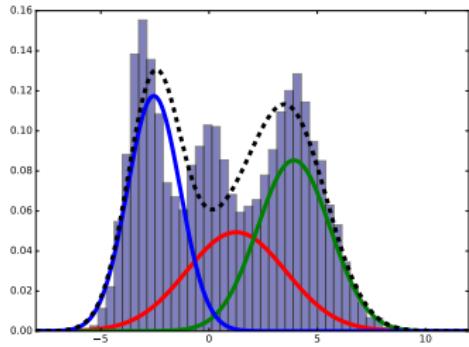
Gaussian Mixture Models



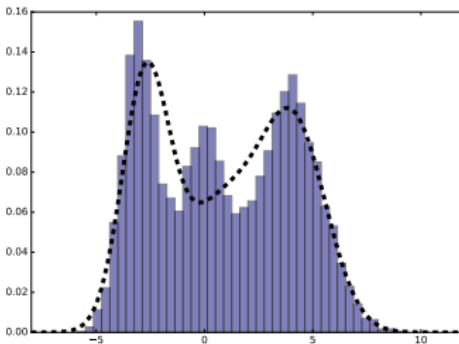
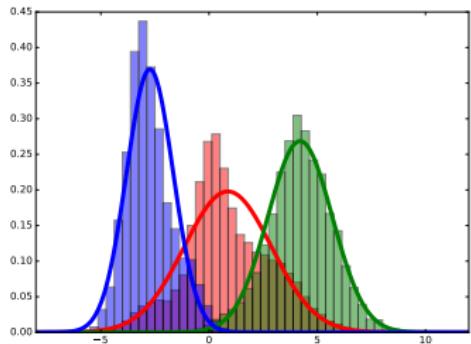
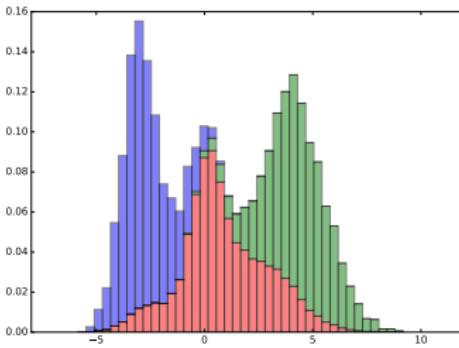
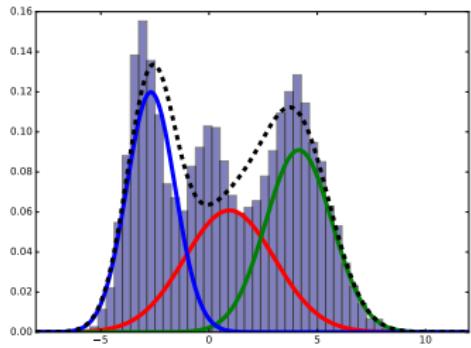
Gaussian Mixture Models



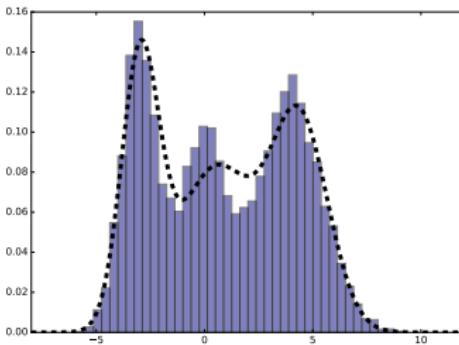
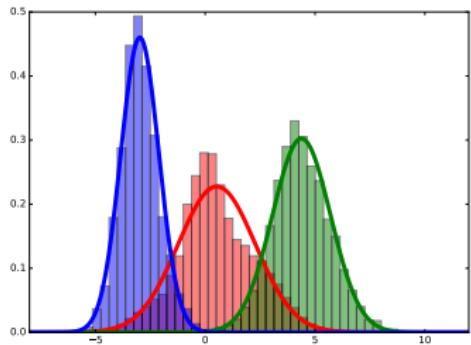
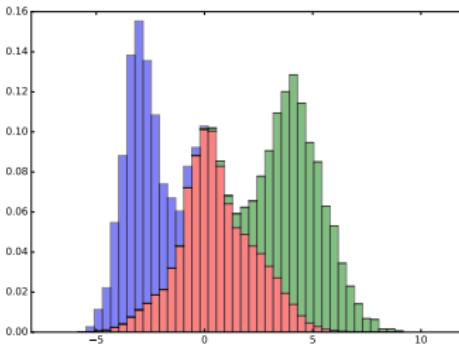
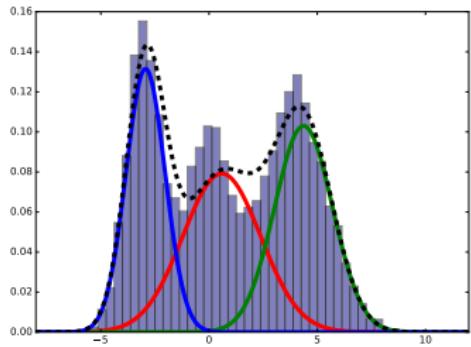
Gaussian Mixture Models



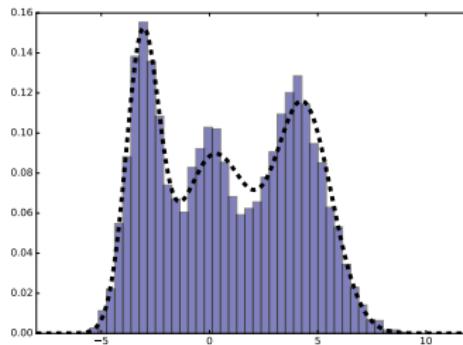
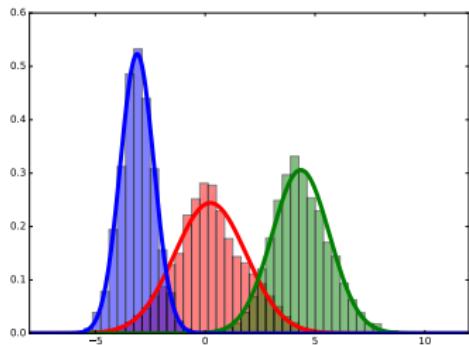
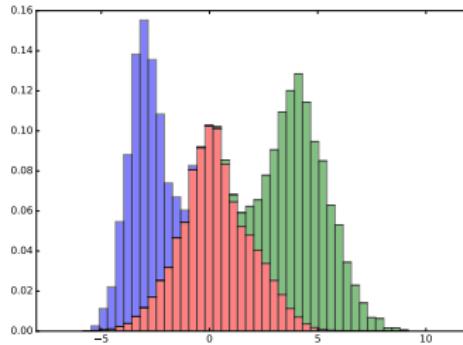
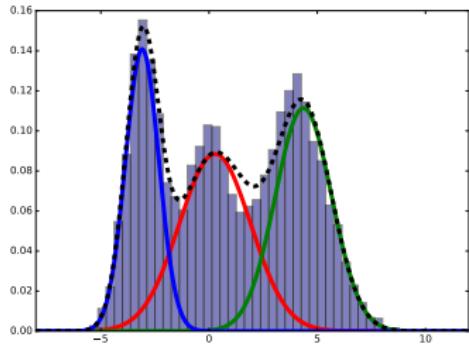
Gaussian Mixture Models



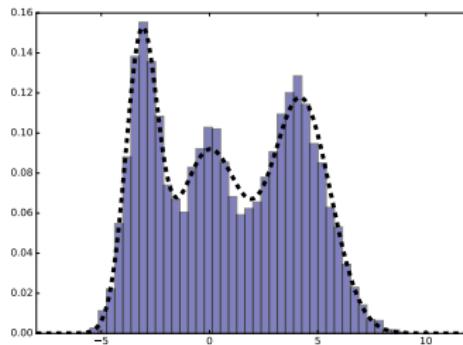
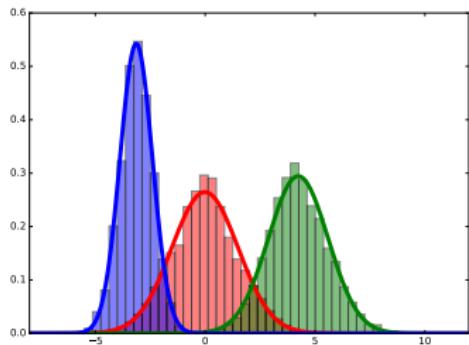
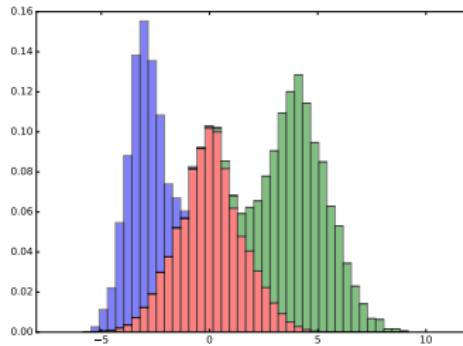
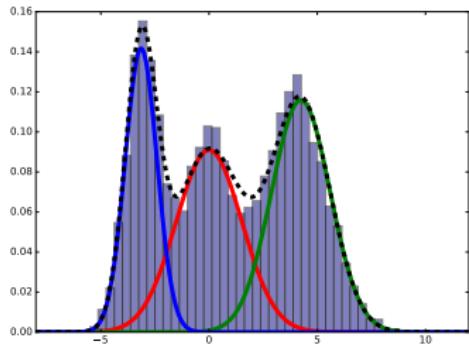
Gaussian Mixture Models



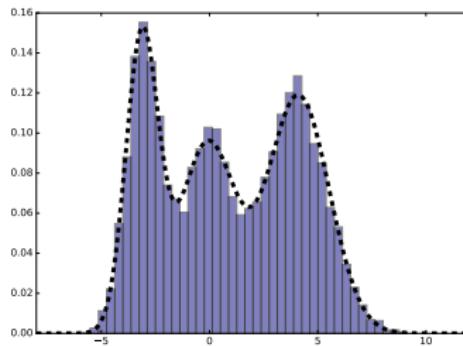
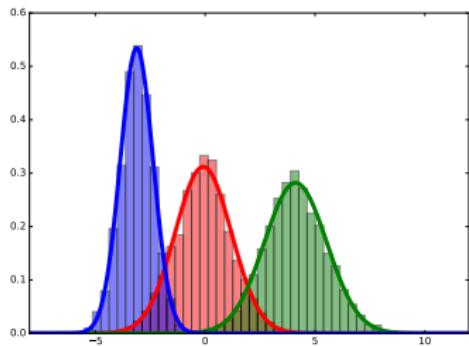
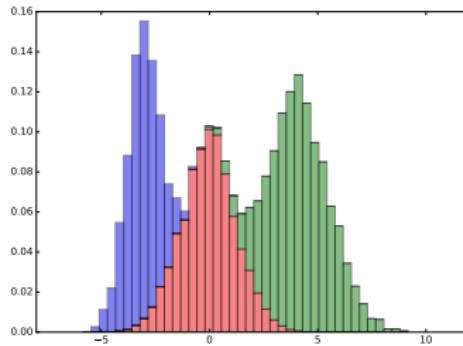
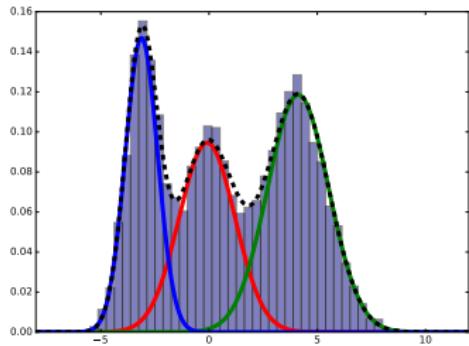
Gaussian Mixture Models



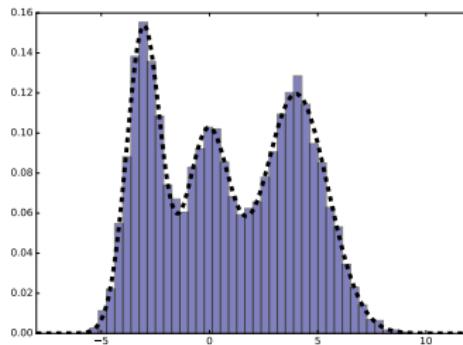
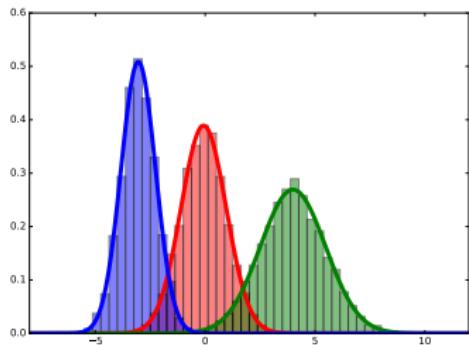
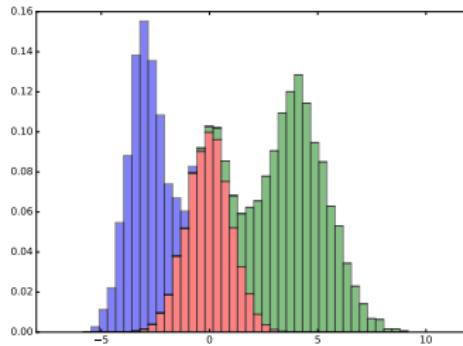
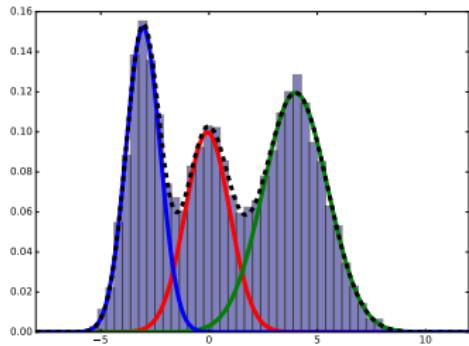
Gaussian Mixture Models



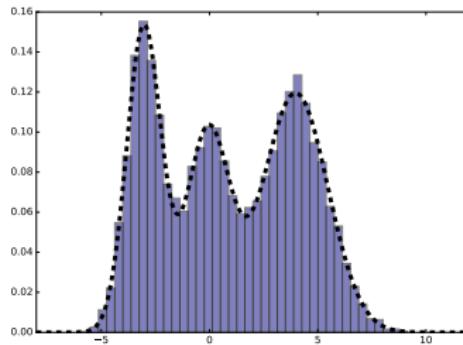
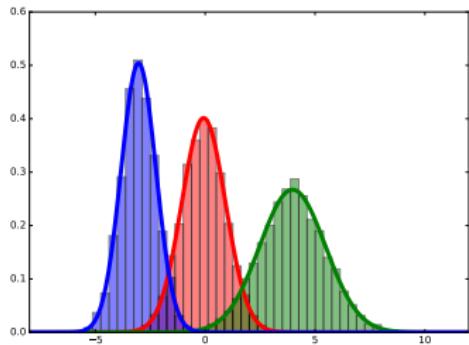
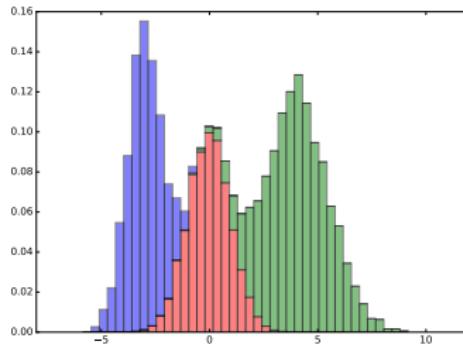
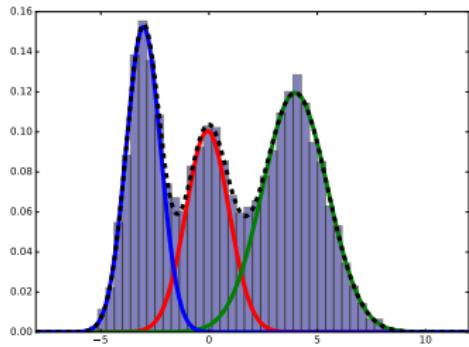
Gaussian Mixture Models



Gaussian Mixture Models



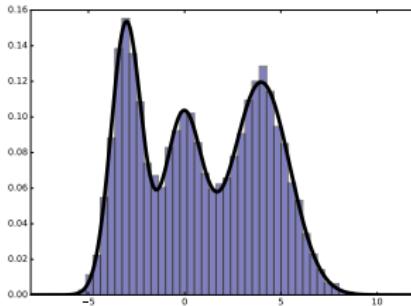
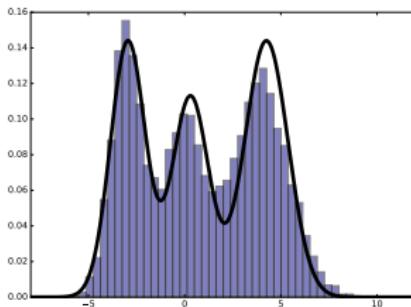
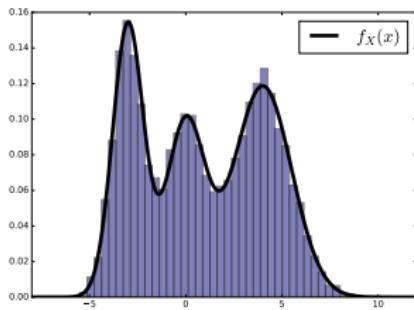
Gaussian Mixture Models



Gaussian Mixture Models

Hard assignment

P.d.f.



Soft assignment

Gaussian Mixture Models

Sampling parameters and estimated values

	π_1	π_2	π_3
P.d.f.	0.25	0.45	0.30
Hard	0.27	0.41	0.32
EM	0.25	0.45	0.30
	μ_1	μ_2	μ_3
P.d.f.	0.00	4.00	-3.00
Hard	0.30	4.25	-2.95
EM	-0.07	3.98	-3.03
	σ_1^2	σ_2^2	σ_3^2
P.d.f.	1.00	2.25	0.64
Hard	0.89	1.27	0.82
EM	0.99	2.23	0.63

Expectation Maximization

Direct maximization of the GMM log-likelihood proved difficult because of the form of the marginal log-density

$$\log f_X(\mathbf{x}|\boldsymbol{\theta}) = \log \sum_{c=1}^K w_c \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

On the contrary, we have seen that the optimization of cluster-conditional likelihoods is straightforward: the cluster-conditional likelihood consists of a sum of normal log-densities

$$\log f_{X|C}(\mathbf{x}|c) = \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

and has a simple expression

Expectation Maximization

The EM is an iterative procedure suited for the ML estimation of the parameters of complex likelihoods² $f_X(x|\theta)$ that can be expressed through marginalization of joint likelihoods $f_{X,H}(x, h|\theta)$:

$$f_X(x) = \int f_{X,H}(x, h) dh = \int f_{X|H}(x|h)f_H(h) dh$$

NOTE: Here and in the following we do not make any assumption on what X represents. We simply assume that it's a random variable or a random vector, for which we **observed realization** x . We shall see later that, for our GMM estimation task, X represents the set of random vectors that describe our dataset \mathcal{D}

²The derivations hold for both continuous and discrete latent variables, replacing integrals with sums

Expectation Maximization

H represents a **latent (or hidden) random variable (or vector)** i.e. a R.V. for which we do not have a **realization** (i.e. we did not observe its value)

As we shall see, the EM transforms the maximization of a log-likelihood $\log f_X(x|\theta)$ into a sequence of optimizations of the joint log-likelihood $\log f_{X,H}(x, h|\theta)$

Expectation Maximization

Let's consider again the marginal log-likelihood

$$\ell(\theta) = \log f_X(x|\theta) = \log \frac{f_{X,H}(x, h|\theta)}{f_{H|X}(h|x, \theta)}$$

Given a density $Q(h)$ with the same support of $f_H(h)$, we can rewrite the log-pdf as

$$\begin{aligned}\log f_X(x|\theta) &= \int Q(h) \log f_X(x|\theta) dh \\ &= \int Q(h) \log \frac{f_{X,H}(x, h|\theta)}{f_{H|X}(h|x, \theta)} dh \\ &= \int Q(h) \log \frac{f_{X,H}(x, h|\theta)}{Q(h)} - \int Q(h) \log \frac{f_{H|X}(h|x, \theta)}{Q(h)} dh\end{aligned}\tag{4}$$

Expectation Maximization

The term

$$\begin{aligned} D_h(Q(h) \parallel f_{H|X}(h|x, \theta)) &= - \int Q(h) \log \frac{f_{H|X}(h|x, \theta)}{Q(h)} dh \\ &= -\mathbb{E}_Q \left[\log \frac{f_{H|X}(h|x, \theta)}{Q(h)} \right] \end{aligned}$$

is called **Kullback-Leibler (KL)** divergence (usually denoted simply as $D(Q \parallel f_{H|X})$)

As we will shortly see, the term

$$\mathcal{L}_h(Q(h), \theta) = \int Q(h) \log \frac{f_{X,H}(x, h|\theta)}{Q(h)} dh = \mathbb{E}_{Q(h)} [f_{X,H}(x, h|\theta)] + \mathcal{H}(Q(h))$$

where $\mathcal{H}(Q(h))$ is the entropy of distribution $Q(h)$, provides a lower bound of the log-likelihood (again, the suffix h is usually omitted, but we keep it to remember we are integrating w.r.t. h)

Expectation Maximization

Let's consider the KL divergence

$$D_h(Q(h) \parallel f_{H|X}(h|x, \theta)) = - \int Q(h) \log \frac{f_{H|X}(h|x, \theta)}{Q(h)} dh$$

Since for every $z > 0$, we have

$$\log z \leq z - 1$$

and $\log z = z - 1$ if and only if $z = 1$, then, for every value of h in the support of Q :

$$-\log \frac{f_{H|X}(h|x, \theta)}{Q(h)} \geq -\frac{f_{H|X}(h|x, \theta)}{Q(h)} + 1$$

with the equality holding if and only if

$$Q(h) = f_{H|X}(h|x, \theta)$$

Expectation Maximization

We thus have

$$-\int Q(h) \log \frac{f_{H|X}(h|x, \theta)}{Q(h)} dh \geq -\int Q(h) \frac{f_{H|X}(h|x, \theta)}{Q(h)} dh + \int Q(h) dh = 0$$

with the quality holding if and only if

$$Q(h) = f_{H|X}(h|x, \theta)$$

almost everywhere (a. e.)³

Therefore

$$D_h(Q(h) \| f_{H|X}(h|x, \theta)) \geq 0$$

and

$$D_h(Q(h) \| f_{H|X}(h|x, \theta)) = 0 \iff Q = f_{H|X} \text{ a. e.}$$

³i.e. over all the domain of H , except for at most a subset of zero measure

Expectation Maximization

We have decomposed the log-likelihood as

$$\log f_X(x|\theta) = \mathcal{L}_h(Q(h), \theta) + D_h(Q(h) \| f_{H|X}(h|x, \theta))$$

Notice that the left hand side of the equation does not depend on the choice of Q

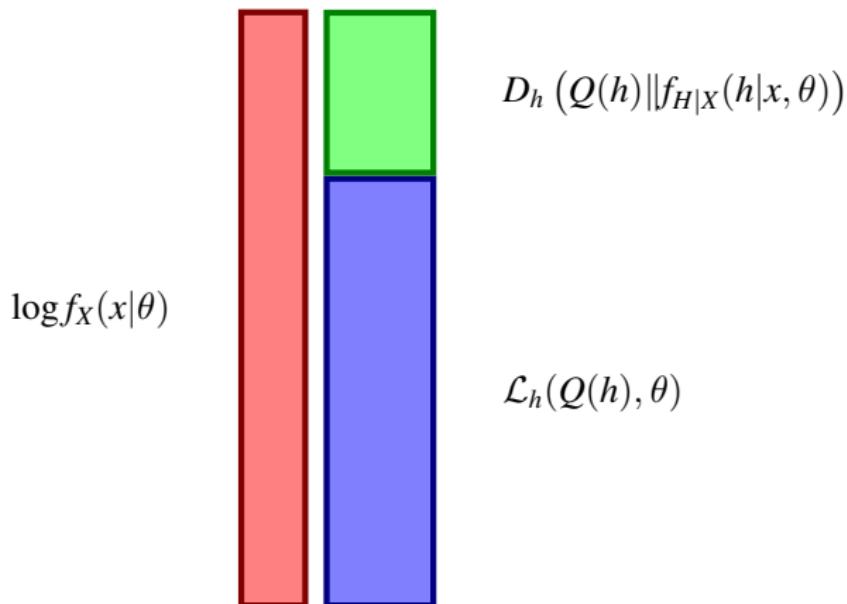
Furthermore,

$$D_h(Q(h) \| f_{H|X}(h|x, \theta)) \geq 0 \implies \mathcal{L}_h(Q(h), \theta) \leq \log f_X(x|\theta)$$

thus $\mathcal{L}_h(Q(h), \theta)$ is a lower bound on the log-likelihood

Expectation Maximization

Decomposition of $\log f_X(x|\theta) = \mathcal{L}_h(Q(h), \theta) + D_h(Q(h)||f_{H|X}(h|x, \theta))$



Expectation Maximization

The EM algorithm optimizes the log-likelihood by iteratively

- Maximizing the lower bound $\mathcal{L}_h(Q(h), \theta)$ with respect to Q
- Maximizing the lower bound $\mathcal{L}_h(Q(h), \theta)$ with respect to θ

From an initial sets of parameters θ_0 :

- $Q_0 = \arg \max_Q \mathcal{L}_h(Q(h), \theta_0)$
- $\theta_1 = \arg \max_\theta \mathcal{L}_h(Q_0(h), \theta)$
- $Q_1 = \arg \max_Q \mathcal{L}_h(Q(h), \theta_1)$
- $\theta_2 = \arg \max_\theta \mathcal{L}_h(Q_1(h), \theta)$
- ...

Expectation Maximization

Let's consider the maximization of the lower bound w.r.t. $Q(h)$, with θ **fixed**: $\theta = \theta^t$

We have shown that

$$\mathcal{L}_h(Q, \theta_t) \leq \log f_X(x|\theta_t)$$

and

$$Q(h) = f_{H|X}(h|x, \theta_t) \implies \mathcal{L}_h(Q(h), \theta_t) = \log f_X(x|\theta_t)$$

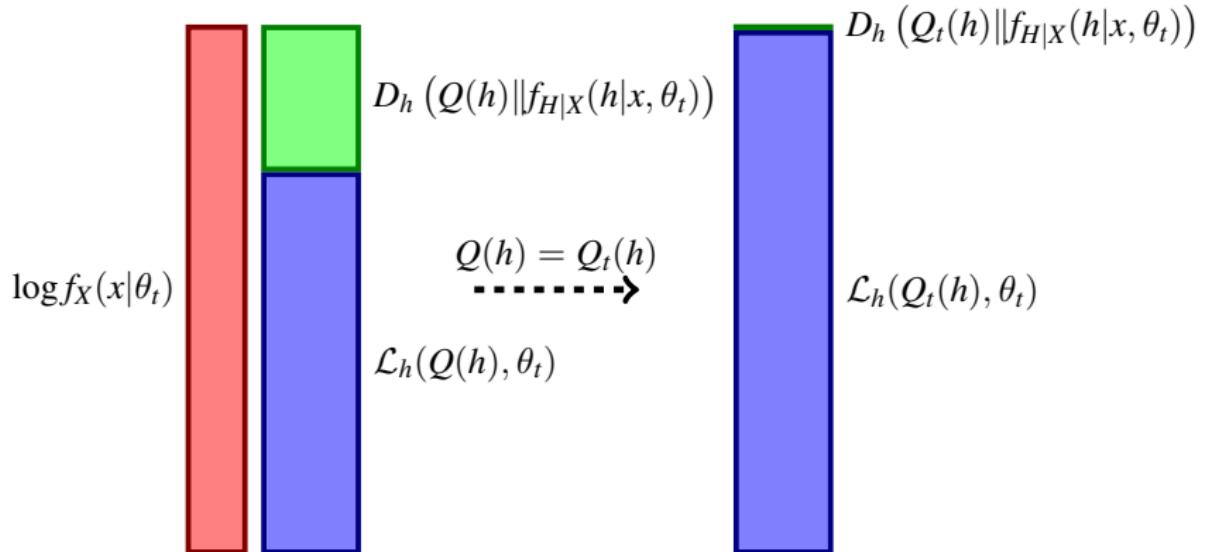
Therefore, we can maximize $\mathcal{L}_h(Q(h), \theta_t)$ w.r.t. Q by simply setting

$$Q_t(h) = f_{H|X}(h|x, \theta_t)$$

i.e., the posterior for H given X and the fixed value θ_t for the parameters

Expectation Maximization

Setting $Q_t(h) = f_{H|X}(h|x, \theta_t)$ does not change the log-likelihood $\log f_X(x|\theta_t)$, but reduces to zero the KL divergence:



Expectation Maximization

We can now maximize $\mathcal{L}_h(Q_t, \theta)$ w.r.t. θ

This requires computing

$$\theta_{t+1} = \arg \max_{\theta} \mathbb{E}_{Q_t(h)} \log f_{X,h}(x, h | \theta)$$

Notice that the distribution we are taking the expectation with respect to does not involve θ anymore:

$$\begin{aligned}\mathbb{E}_{Q_t(h)} \log f_{X,h}(x, h | \theta) &= \int Q_t(h) \log f_{X,H}(x, h | \theta) dh \\ &= \int f_{H|X}(h|x, \theta_t) \log f_{X,H}(x, h | \theta) dh\end{aligned}$$

since the parameters used in the distribution

$$Q_t(h) = f_{H|X}(h|x, \theta_t)$$

are fixed to θ_t ($Q_t(h)$ does not depend on θ , but on θ_t)

Expectation Maximization

We can also rewrite the problem as maximization of

$$\mathbb{E}_{Q_t(h)} \log f_{X,H}(x, h|\theta) = \mathbb{E}_{Q_t(h)} \log f_{X|H}(x|h, \theta) + \mathbb{E}_{Q_t(h)} \log f_H(h|\theta)$$

which corresponds to the expression we used for the GMM

Since we are maximizing $\mathcal{L}_h(Q_t(h), \theta)$, we have

$$\mathcal{L}(Q_t, \theta_{t+1}) \geq \mathcal{L}(Q_t, \theta_t)$$

Expectation Maximization

$\mathcal{L}_h(Q_t, \theta_{t+1})$ is a lower bound of $\log f_X(x|\theta_{t+1})$, thus

$$\log f_X(x|\theta_{t+1}) \geq \mathcal{L}_h(Q_t, \theta_{t+1})$$

Indeed,

$$\log f_X(x|\theta_{t+1}) = \mathcal{L}(Q_t(h), \theta_{t+1}) + D(Q_t, f_{H|X}(h|x, \theta_{t+1}))$$

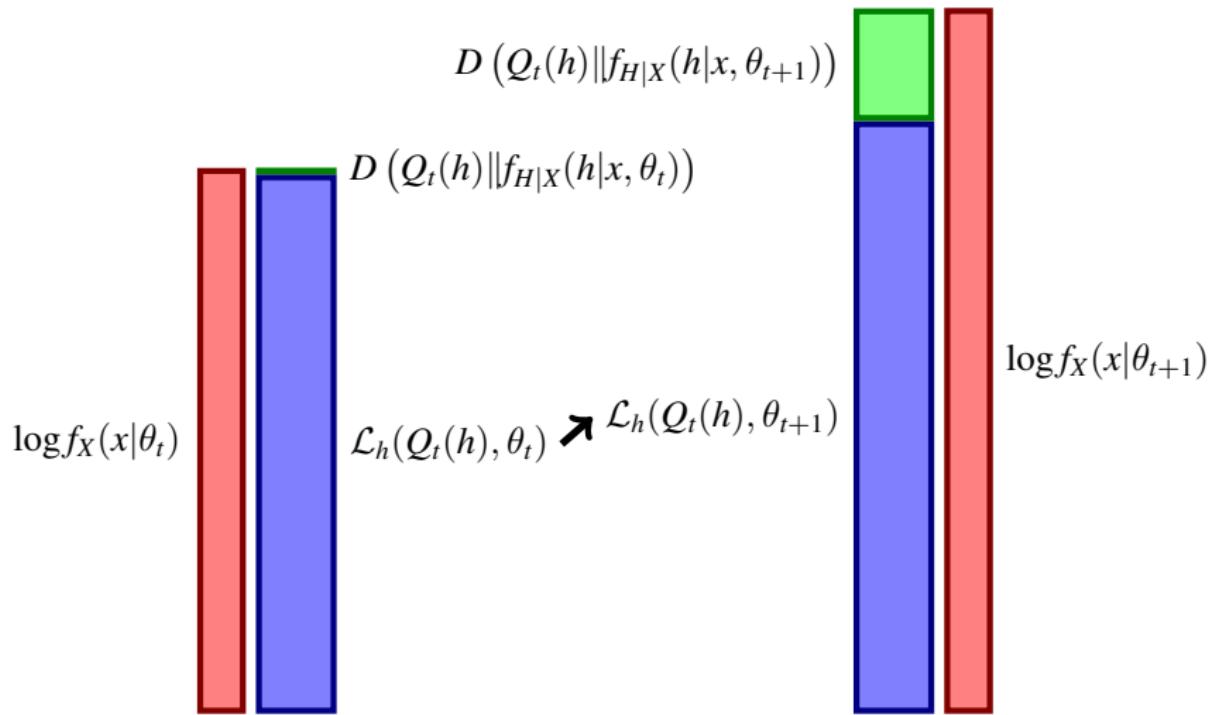
Maximization with respect to θ has increased both \mathcal{L}_h and the KL-divergence, since, in general, $Q_t(h) \neq f_{H|X}(h|x, \theta_{t+1})$ unless we reached convergence

We thus have that

$$\log f_X(x|\theta_{t+1}) \geq \log f_X(x|\theta_t)$$

Expectation Maximization

Maximization w.r.t. θ of $\mathcal{L}_h(Q_t, \theta)$ increases the log-pdf $\log f_X(x|\theta)$



Expectation Maximization

The algorithm iterates between two steps

- **Expectation (E) step:** Compute the posterior distribution $f_{H|X}(h|x, \theta_t)$ and compute the **auxiliary function**:

$$\mathcal{Q}(\theta, \theta_t) = \mathbb{E}_{f_{H|X}(h|x, \theta_t)} [\log f_{X,H}(x, h|\theta)]$$

- **Maximization (M) step:** **Maximize** $\mathcal{Q}(\theta, \theta_t)$ w.r.t. θ to obtain

$$\theta_{t+1} = \arg \max_{\theta} \mathcal{Q}(\theta, \theta_t)$$

Expectation Maximization

Under very weak conditions it can be shown that the EM algorithm converges to a saddle point of the log-likelihood θ^*

Sufficient conditions for θ^* to be a local maximum exist, but are not easy to verify in practice

The saddle point will depend on the initial set of parameters

The choice of a good starting point is important for the estimation of good models

It is sometimes useful to apply several times the EM algorithm with different starting points

Gaussian Mixture Models

Let's apply the algorithm to the GMM

We have a set of n hidden variables $h = (C_1 \dots C_N)$ that represent the cluster assignments, i.e. the assignment of each sample to a component of the GMM

The GMM specifies the joint likelihood for samples and cluster assignments. For a single sample:

$$f_{X_i, C_i}(\mathbf{x}_i, c) = w_c \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

The cluster R.V. C_i has a **Categorical prior distribution**

$$P(C_i = c) = w_c$$

whereas the sample conditional likelihood is

$$f_{X_i | C_i}(\mathbf{x}_i | c) = \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

Gaussian Mixture Models

We assume that samples are independent given the model parameters, so that we can express the log-likelihood for all the training set samples as

$$\log f_{X_1 \dots X_N, C_1 \dots C_N}(\mathbf{x}_1 \dots \mathbf{x}_N, c_1 \dots c_N | \boldsymbol{\theta}) = \sum_{i=1}^N \log f_{X_i, C_i}(\mathbf{x}_i, c_i | \boldsymbol{\theta})$$

The EM algorithm requires computing the posterior for the hidden variables $C_1 \dots C_N | X_1 \dots X_N, \boldsymbol{\theta}$.

Due to the independence assumptions, also the posterior distribution factorizes as

$$f_{C_1 \dots C_N | X_1 \dots X_N}(c_1 \dots c_N | \mathbf{x}_1 \dots \mathbf{x}_N, \boldsymbol{\theta}) = \prod_{i=1}^N P(C_i = c_i | X_i = \mathbf{x}_i, \boldsymbol{\theta})$$

Gaussian Mixture Models

The **E-step** requires computing the auxiliary function

$$\begin{aligned}\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}_t) &= \mathbb{E}_{C_1 \dots C_N | X_1 = \mathbf{x}_1 \dots X_N = \mathbf{x}_N, \boldsymbol{\theta}_t} [\log f_{X_1 \dots X_N, C_1 \dots C_N}(\mathbf{x}_1 \dots \mathbf{x}_N, c_1 \dots c_N | \boldsymbol{\theta})] \\ &= \sum_{i=1}^N \mathbb{E}_{C_1 \dots C_N | X_1 = \mathbf{x}_1 \dots X_N = \mathbf{x}_N, \boldsymbol{\theta}_t} [\log f_{X_i, C_i}(\mathbf{x}_i, c)] \\ &= \sum_{i=1}^N \mathbb{E}_{C_i | X_i = \mathbf{x}_i, \boldsymbol{\theta}_t} [\log f_{X_i, C_i}(\mathbf{x}_i, c)]\end{aligned}$$

Gaussian Mixture Models

The EM algorithm becomes:

E-step: Compute $\gamma_{c,i} = P(C_i = c | X_i = \mathbf{x}_i, \boldsymbol{\theta}^t)$. The auxiliary function is

$$\begin{aligned}\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}_t) &= \sum_{i=1}^N \sum_{c=1}^K P(C_i = c | X_i = \mathbf{x}_i, \boldsymbol{\theta}_t) \log f_{X_i, C_i}(\mathbf{x}_i, c | \boldsymbol{\theta}) \\ &= \sum_{i=1}^N \sum_{c=1}^K \gamma_{c,i} [\log \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) + \gamma_{c,i} \log w_c] \\ &= \sum_{i=1}^N \sum_{c=1}^K \gamma_{c,i} \left(\frac{1}{2} \log |\boldsymbol{\Lambda}_c| - \frac{1}{2} (\mathbf{x}_i - \boldsymbol{\mu}_c)^T \boldsymbol{\Lambda}_c (\mathbf{x}_i - \boldsymbol{\mu}_c) \right) \\ &\quad + \sum_{i=1}^N \sum_{c=1}^K \gamma_{c,i} \log w_c\end{aligned}$$

Gaussian Mixture Models

The EM algorithm becomes:

M-step: Maximize $\mathcal{Q}(\theta, \theta_t)$ w.r.t. $\theta = (\mathbf{M}, \mathcal{S}, \mathbf{w})$:

$$\boldsymbol{\mu}_c^* = \frac{\sum_i \gamma_{c,i} \mathbf{x}_i}{\sum_i \gamma_{c,i}}$$

$$\boldsymbol{\Sigma}_c^* = \frac{\sum_i \gamma_{c,i} (\mathbf{x}_i - \boldsymbol{\mu}_c)(\mathbf{x}_i - \boldsymbol{\mu}_c)^T}{\sum_i \gamma_{c,i}}$$

$$w_c^* = \frac{\sum_i \gamma_{c,i}}{\sum_i \sum_c \gamma_{c,i}}$$

The new estimate of the parameters is $\theta_{t+1} = (\mathbf{M}_{t+1}, \mathcal{S}_{t+1}, \mathbf{w}_{t+1})$:

$$\mathbf{M}_{t+1} = [\boldsymbol{\mu}_1^* \dots \boldsymbol{\mu}_K^*] , \quad \mathcal{S}_{t+1} = [\boldsymbol{\Sigma}_1^* \dots \boldsymbol{\Sigma}_K^*] , \quad \mathbf{w}_{t+1} = [w_1^* \dots w_K^*]$$

Gaussian Mixture Models for classification

Just as we used Gaussian densities for modeling the samples of different classes in a classification task, we can use GMM to model class distributions

We can, for example, assume that the samples of class c are generated by a GMM with parameters $(\mathbf{M}_c, \mathcal{S}_c, \mathbf{w}_c)$

For each class we want to recognize, we can compute the ML estimate of a GMM for the samples of that class. We can then use the estimated densities to compute class conditional log-likelihoods and class posterior distributions or log-likelihood ratios

Gaussian Mixture Models for classification

MVG for classification:

$$X_i|C_i = c \sim \mathcal{N}(\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

$$f_{X_t|C_t}(\mathbf{x}_t|c) = \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$$

$$P(C_t = c|X_t = \mathbf{x}_t) = \frac{\mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)P(C_t = c)}{\sum_c' \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_{c'}, \boldsymbol{\Sigma}_{c'})P(C_t = c')}$$

GMM for classification (we use index k to denote one of the K GMM components):

$$X_i|C_i = c \sim GMM(\mathbf{M}_c, \mathcal{S}_c, \mathbf{w}_c)$$

$$f_{X_t|C_t}(\mathbf{x}_t|c) = \sum_{k=1}^K w_k \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_{c,k}, \boldsymbol{\Sigma}_{c,k})$$

$$P(C_t = c|X_t = \mathbf{x}_t) = \frac{P(C_t = c) \sum_{k=1}^K w_{c,k} \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_{c,k}, \boldsymbol{\Sigma}_{c,k})}{\sum_{c'} P(C_t = c') \sum_{k=1}^K w_{c',k} \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_{c',k}, \boldsymbol{\Sigma}_{c',k})}$$

We can also exploit GMMs clustering capabilities for **open-set** multiclass classification

The difficulty in open-set classification consists in building a robust model for the **none-of-the-others** class

This class is usually very heterogeneous, and explicit modeling its sub-components requires labeled examples of its possible objects

We can partially alleviate the issue using a GMM.

Gaussian Mixture Models for classification

We can assume that samples of known classes can be modeled by MVG distributions

We can collect a large set of **unlabeled** samples for the **none-of-the-others** class

We can model the samples of the **none-of-the-others** class using a GMM

The GMM will find homogeneous clusters (sub-classes) of the none-of-the-others population, and can be used as an estimate for the conditional density of a test sample given that it belongs to the **one-of-the-others** class⁴

⁴Given the complexity of the task, and due to the fact that different amounts of parameters are used for modeling the non-of-the-others class these kind of models often provide class-conditional likelihoods that are not calibrated, and may require further score processing (e.g. score calibration) to obtain good decisions

Gaussian Mixture Models

As we did for MVG, we can train GMM with diagonal covariance matrices to reduce the number of parameters to estimate (reducing overfitting and computational costs).

The solution is again given by the diagonals of Σ_c^* 's we defined before

We may need more components to model more complex distributions

The diagonal covariance assumption **DOES NOT** correspond to the Naive Bayes assumption in this case

The Naive Bayes assumption would correspond to training a **different** GMM (possibly with different number of components) for each set of features that are assumed independent from the others

We can also assume that all components of a GMM have the same covariance matrix (tied GMM).

Gaussian Mixture Models for classification

MNIST — GMM (PCA 50)

Components:	1	2	4	8	16
FullCov	3.6%	3.4%	2.8%	2.3%	2.2%
Diagonal	12.3%	10.1%	8.9%	7.6%	6.2%
Components:	32	64	128	256	
FullCov	2.3%				
Diagonal	5.1%	4.3%	4.3%	4.3%	

Gaussian Mixture Models

Initialization plays an important role in GMM training

Hard-assignment with isotropic covariances → K-means

K-means can be used as initializer

Alternative approach: LBG (can also be used for K-means)

LBG algorithm:

- Split the components of a G -components GMM

$$\mu_c^+ = \mu_c + \varepsilon$$

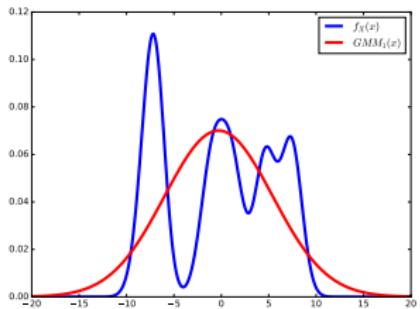
$$\mu_c^- = \mu_c - \varepsilon$$

to obtain an initial $2G$ -components GMM

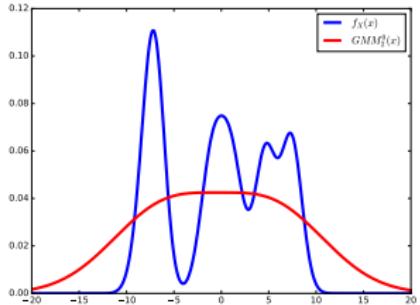
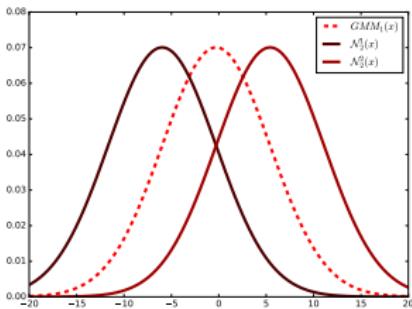
- Run the EM algorithm until convergence for the $2G$ -components GMM
- Iterate until the desired number of Gaussians is reached

A good value for ε can be a displacement along the principal eigenvector of the covariance matrix Σ_c

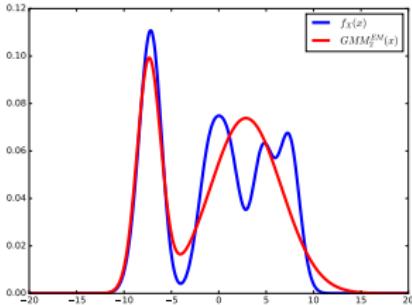
Gaussian Mixture Models



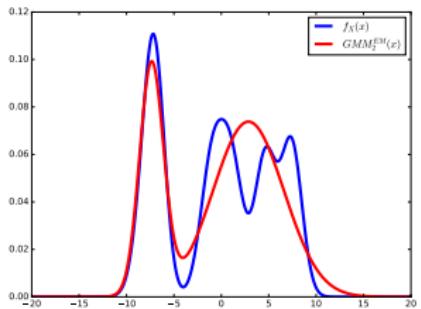
Split →



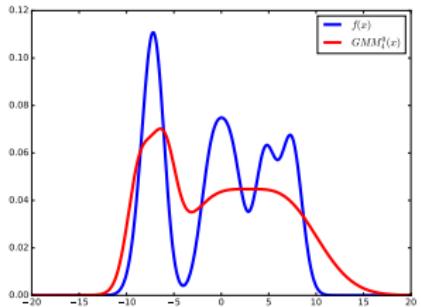
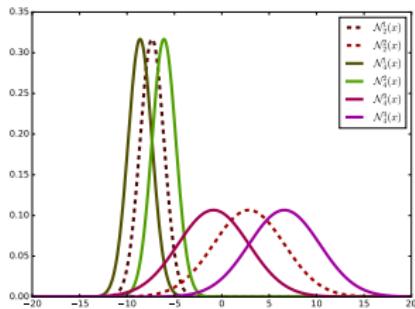
EM →



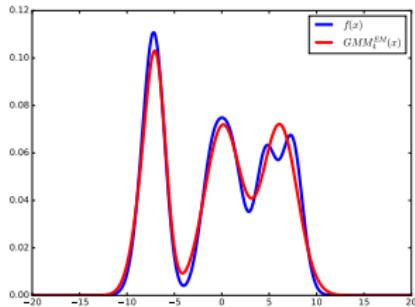
Gaussian Mixture Models



Split →



EM →



Gaussian Mixture Models

Problem: what is the “right” number of Gaussians?

If we increase the number of components, the likelihood will increase

We cannot choose based on likelihood alone

Several criteria, more or less successful, have been proposed (AIC, BIC)

We can also resort to **cross-validation**

Gaussian Mixture Models

We also need to pay attention to degenerate models

As we said at the beginning, the log-likelihood for a GMM, as long as we have at least two components, is unbounded

The EM algorithm will usually find local maxima that are well-behaved, however, especially if we have too many components, we may obtain degenerate models which cause numerical issues

Some heuristics can be used to force models to be well-behaved (e.g. impose minimum values for the eigenvalues of the covariance matrices, tie the covariance of different components)

We can also modify our initialization so that the algorithm may end up in a different local maximum