

MACHINE LEARNING

With
Python



AJIT SINGH

PREFACE

This book is to provide a conceptual introduction to statistical or machine learning (ML) techniques for those that would not normally be exposed to such approaches during their typical required statistical training.

Machine learning can be described as a form of statistical analysis, often even utilizing well-known and familiar techniques, that has bit of a different focus than traditional analytical practice in applied disciplines. The key notion is that flexible, automatic approaches are used to detect patterns within the data, with a primary focus on making predictions on future data. Python versions of the model examples are available here. In addition, Marcio Mourao has provided additional Python examples.

As for prerequisite knowledge, I will assume a basic familiarity with regression analyses typically presented in applied disciplines. Regarding programming, none is really required to follow most of the content here. Note that I will not do much explaining of the code, as I will be more concerned with getting to a result than clearly detailing the path to it.

This book is written for sophomore students in computer science, technology, engineering, or mathematics (STEM), assuming that they know algebra and calculus. Readers should have already solved some problems using computer programs. More specifically, the book takes a task-based approach to machine learning, with almost 200 self-contained solutions (you can copy and paste the code and it will run) for the most common tasks a data scientist or machine learning engineer building a model will run into.

The book discusses many methods that have their bases in different fields: statistics, pattern recognition, neural networks, artificial intelligence, signal processing, control, and data mining. In the past, research in these different communities followed different paths with different emphases. In this book, the aim is to incorporate them together to give a unified treatment of the problems and the proposed solutions to them.

This is an introductory textbook, intended for senior undergraduate and graduate-level courses on machine learning, as well as engineers working in the industry who are interested in the application of these methods. The prerequisites are courses on computer programming, probability, calculus, and linear algebra. The aim is to have all learning algorithms sufficiently explained so it will be a small step from the equations given in the book to a computer program. For some cases, pseudocodes of algorithms are also included to make this task easier.

The book can be used for a one-semester course by sampling from the chapters. I very much enjoyed writing this book; I hope you will enjoy reading it.

Note: External sources of text and images as a contribution for this book are clearly mentioned inline along with the respective text and images.

ABOUT AUTHOR

Ajit Singh

Assistant Professor (Ad-hoc)
Department of Computer Application
Patna Women's College, Patna, Bihar.



World Record Title(s):

1. Online World Record (OWR).
2. Future Kalam's Book of Records.

20+ Years of strong teaching experience for Under Graduate and Post Graduate courses of Computer Science across several colleges of Patna University and NIT Patna, Bihar, IND.

[Amazon's Author Profile]

www.amazon.com/author/ajitsingh

[Contact]

URL: <http://www.ajitvoice.in>

Email: ajit_singh24@yahoo.com

[Memberships]

1. InternetSociety (2168607) - Delhi/Trivandrum Chapters
2. IEEE (95539159)
3. International Association of Engineers (IAENG-233408)
4. Eurasia Research STRA-M19371
5. Member – IoT Council
5. ORCID <https://orcid.org/0000-0002-6093-3457>
6. Python Software Foundation
7. Data Science Foundation
8. Non Fiction Authors Association (NFAA-21979)

Copyright © 2019 Ajit Singh

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the contents presented.

CONTENTS

1	Introduction to Machine Learning Machine learning within data science IT/computing science tools Statistics and applied mathematics Data analysis methodology	7
2	Python language Set up your programming environment using Anaconda Import libraries Data types Math Comparisons and Boolean operations Conditional statements Lists Tuples Strings Dictionaries Sets Functions Loops List comprehensions Exceptions handling Basic operating system interfaces (os) Object Oriented Programming (OOP) Exercises	12
3	Numpy: arrays and matrices Create arrays Stack arrays Selection Vector zed operations Exercises	24
4	Pandas: data manipulation Create DataFrame Concatenate DataFrame Join DataFrame Summarizing Columns selection Rows selection Rows selection / filtering. Sorting Reshaping by pivoting Quality control: duplicate data Quality control: missing data Rename values Dealing with outliers Groupby File I/O Exercises	27
5	Matplotlib: Plotting Preamble about the F-distribution Basic plots Scatter (2D) plots Saving Figures	33

Exploring data (with seaborn)
Density plot with one figure containing multiple axes

6	Univariate statistics	43
	Estimators of the main statistical measures	
	Main distributions	
	Testing pairwise associations	
	Non-parametric test of pairwise associations	
	Linear model	
	Linear model with stats models	
	Multiple comparisons	
	Exercise	
7	Multivariate statistics	65
	Linear Algebra	
	Mean vector	
	Covariance matrix	
	Precision matrix	
	Mahalanobis distance	
	Multivariate normal distribution	
	Exercises	
8	Dimension reduction and feature extraction	73
	Introduction	
	Singular value decomposition (SVD)	
	Principal components analysis (PCA)	
	Multi-dimensional Scaling (MDS)	
	Nonlinear dimensionality reduction	
9	Clustering	84
	K-means clustering	
	Hierarchical clustering	
	Gaussian mixture models	
	Model selection	
10	Linear methods for regression	91
	Ordinary least squares	
	Linear regression with scikit-learn	
	Overfitting	
	Ridge regression (ℓ_2 -regularization)	
	Lasso regression (ℓ_1 -regularization)	
	Elastic-net regression ($\ell_2 - \ell_1$ -regularization)	
11	Linear classification	102
	Fisher's linear discriminant with equal class covariance	
	Linear discriminant analysis (LDA)	
	Logistic regression	
	Overfitting	
	Ridge Fisher's linear classification (L2-regularization)	
	Ridge logistic regression (L2-regularization)	
	Lasso logistic regression (L1-regularization)	
	Ridge linear Support Vector Machine (L2-regularization)	
	Lasso linear Support Vector Machine (L1-regularization)	
	Exercise	
	Elastic-net classification (L2-L1-regularization)	
	Metrics of classification performance evaluation	
	Imbalanced classes	

12 Non linear learning algorithms	116
Support Vector Machines (SVM)	
Random forest	
13 Resampling Methods	118
Left out samples validation	
Cross-Validation (CV)	
CV for model selection: setting the hyper parameters	
Random Permutations	
Bootstrapping	
14 Scikit-learn processing pipelines	126
Data pre-processing	
Scikit-learn pipelines	
Regression pipelines with CV for parameters selection	
Classification pipelines with CV for parameters selection	
15 Case studies of ML	134
Default of credit card clients Data Set	

**CHAPTER
ONE**

INTRODUCTION TO MACHINE LEARNING

The learning that is being done is always based on some sort of observations or data, such as examples (the most common case in this book), direct experience, or instruction. So in general, machine learning is about learning to do better in the future based on what was experienced in the past.

The emphasis of machine learning is on *automatic* methods. In other words, the goal is to devise learning algorithms that do the learning automatically without human intervention or assistance. The machine learning paradigm can be viewed as programming by example." Often we have a specific task in mind, such as spam filtering. But rather than program the computer to solve the task directly, in machine learning, we seek methods by which the computer will come up with its own program based on examples that we provide.

In 1959, Arthur Samuel defined machine learning as a "Field of study that gives computers the ability to learn without being explicitly programmed". Tom M. Mitchell provided a widely quoted, more formal definition: "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". This definition is notable for its defining machine learning in fundamentally operational rather than cognitive terms, thus following Alan Turing's proposal in his paper "Computing Machinery and Intelligence" that the question "Can machines think?" be replaced with the question "Can machines do what we (as thinking entities) can do?"

Machine learning is a core subarea of artificial intelligence. It is very unlikely that we will be able to build any kind of intelligent system capable of any of the facilities that we associate with intelligence, such as language or vision, without using learning to get there. These tasks are otherwise simply too difficult to solve. Further, we would not consider a system to be truly intelligent if it were incapable of learning since learning is at the core of intelligence.

Although a subarea of AI, machine learning also intersects broadly with other fields, especially statistics, but also mathematics, physics, theoretical computer science and more.

Examples of Machine Learning Problems

There are many examples of machine learning problems. Much of this course will focus on classification problems in which the goal is to categorize objects into a fixed set of categories. Here are several examples:

Optical character recognition: categorize images of handwritten characters by the letters represented.

Face detection: find faces in images (or indicate if a face is present) spam

Filtering: identify email messages as spam or non-spam.

Topic spotting: categorize news articles (say) as to whether they are about politics, sports, entertainment, etc.

Spoken language understanding: within the context of a limited domain, determine the meaning of something uttered by a speaker to the extent that it can be classified into one of a fixed set of categories.

Medical diagnosis: diagnose a patient.

Customer segmentation: predict, for instance, which customers will respond to a particular promotion.

Fraud detection: identify credit card transactions (for instance) which may be fraudulent in nature.

Weather prediction: predict, for instance, whether or not it will rain tomorrow.

Although much of what we will talk about will be about classification problems, there are other important learning problems. In classification, we want to categorize objects into fixed categories. In regression, on the other hand, we are trying to predict a real value. For instance, we may wish to predict *how much* it will rain tomorrow. Or, we might want to predict how much a house will sell for.

A richer learning scenario is one in which the goal is actually to behave intelligently, or to make intelligent decisions. For instance, a robot needs to learn to navigate through its environment without colliding with anything. To use machine learning to make money on the stock market, we might treat investment as a classification problem (will the stock go up or down) or a regression problem (how much will the stock go up), or, dispensing with these intermediate goals, we might want the computer to learn directly how to decide to make investments so as to maximize wealth.

Goals of Machine Learning Research

The primary goal of machine learning research is to develop general purpose algorithms of practical value. Such algorithms should be efficient. As usual, as computer scientists, we care about time and space efficiency. But in the context of learning, we also care a great deal about another precious resource, namely, the amount of data that is required by the learning algorithm.

Learning algorithms should also be as general purpose as possible. We are looking for algorithms that can be easily applied to a broad class of learning problems, such as those listed above.

Of course, we want the result of learning to be a prediction rule that is as accurate as possible in the predictions that it makes.

Occasionally, we may also be interested in the interpretability of the prediction rules produced by learning. In other words, in some contexts (such as medical diagnosis), we want the computer to do prediction rules that are easily understandable by human experts.

As mentioned above, machine learning can be thought of as programming by example." What is the advantage of machine learning over direct programming? First, the results of using machine learning are often more accurate than what can be created through direct programming. The reason is that machine learning algorithms are data driven, and are able to examine large amounts of data. On the other hand, a human expert is likely to be guided by imprecise impressions or perhaps an examination of only a relatively small number of examples.

Also, humans often have trouble expressing what they know, but have no difficulty labelling items. For instance, it is easy for all of us to label images of letters by the character represented, but we would have a great deal of trouble explaining how we do it in precise terms.

Another reason to study machine learning is the hope that it will provide insights into the general phenomenon of learning. Some of the questions that might be answered include:

1. What are the intrinsic properties of a given learning problem that make it hard or easy to solve?
2. How much do you need to know ahead of time about what is being learned in order to be able to learn it effectively?
3. Why are "simpler" hypotheses better?

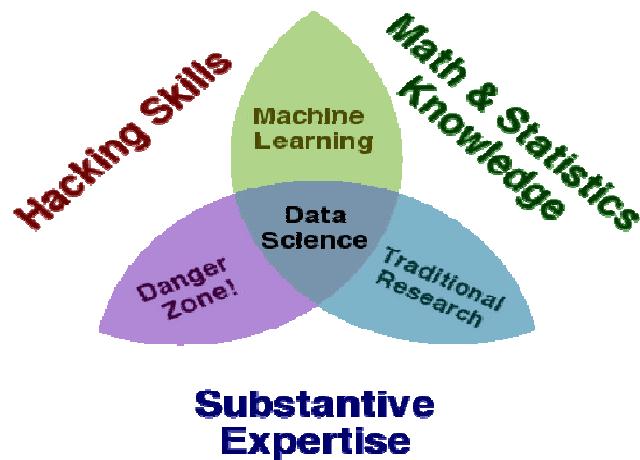
Types of problems and tasks

Machine learning tasks are typically classified into three broad categories, depending on the nature of the learning “signal” or “feedback” available to a learning system.

These are:

- Supervised learning: The computer is presented with example inputs and their desired outputs, given by a “teacher”, and the goal is to learn a general rule that maps inputs to outputs.
- Unsupervised learning: No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end.
- Reinforcement learning: A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle), without a teacher explicitly telling it whether it has come close to its goal or not. Another example is learning to play a game by playing against an opponent.

Machine learning within data science



Machine learning covers two main types of data analysis:

1. Exploratory analysis: Unsupervised learning. Discover the structure within the data. E.g.: Experience (in years in a company) and salary are correlated.
2. Predictive analysis: Supervised learning. This is sometimes described as to “learn from the past to predict the future”. Scenario: a company wants to detect potential future clients among a base of prospect. Retrospective data analysis: given the base of prospected company (with their characteristics: size, domain, localization, etc.) some became clients, some do not. Is it possible to learn to predict those that are more likely to become clients from their company characteristics? The training data consist of a set of n training samples. Each sample, x_i , is a vector of p input features (company characteristics) and a target feature ($y_i \in \{ \text{Yes}, \text{No} \}$) (whether they became a client or not).

IT/Computing Science tools

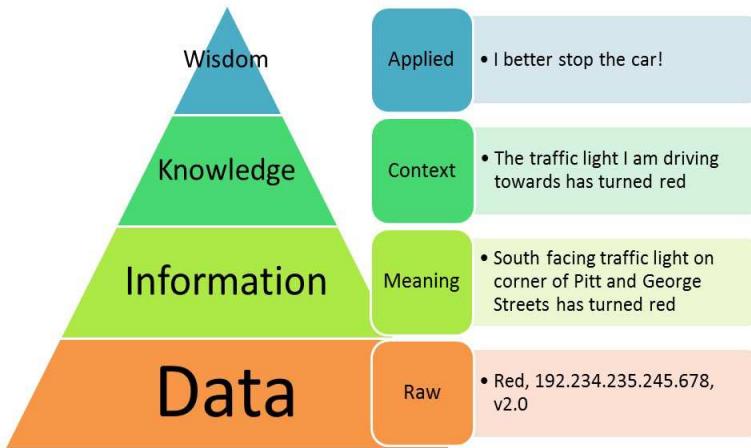
- Python: the language
- Numpy: raw numerical data
- Pandas: structured data

Statistics and applied mathematics

- Linear model
- Non parametric statistics
- Linear algebra: matrix operations, inversion, Eigen values.

Data analysis methodology

DIKW Pyramid: Data, Information, Knowledge, and Wisdom



Methodology

1. Discuss with your customer:
 - Understand his needs.
 - Formalize his needs into a learning problem.
 - Define with your customer the learning dataset required for the project.
 - Goto 1. Until convergence of both sides (you and the customer).
2. In a document formalize (i) the project objectives; (ii) the required learning dataset; more specifically the input data and the target variables. (iii) The conditions that define the acquisition of the dataset. In this document warn the customer that the learned algorithms may not work on new data acquired under different condition.
3. Read your learning dataset (level D of the pyramid) provided by the customer.
4. Clean your data (QC: Quality Control) (reach level I of the pyramid).
5. Explore data (visualization, PCA) and perform basics univariate statistics (reach level K of the pyramid).
7. Perform more complex multivariate-machine learning.
8. Model validation. First deliverable: the predictive model with performance on training dataset.
9. Apply on new data (level W of the pyramid).

CHAPTER TWO

PYTHON LANGUAGE

Set up your programming environment using Anaconda

1. Download anaconda 2019.03 for Linux Installer <https://www.anaconda.com/distribution/>
2. Install it, on Linux:

Python 2.7:

```
bash Anaconda2-2.7.0-Linux-x86_64.sh
```

Python 3.7:

```
bash Anaconda3-3.7.0-Linux-x86_64.sh
```

3. Add anaconda path in your PATH variable in your .bashrc file:

Python 2.7:

```
export PATH="${HOME}/anaconda2/bin:$PATH"
```

Python 3.7:

```
export PATH="${HOME}/anaconda3/bin:$PATH"
```

4. Optional: install additional packages:

Using conda:

```
conda install seaborn
```

Using pip:

```
pip install -U --user seaborn
```

Optional:

```
pip install -U --user nibabel pip install -U --  
user nilearn
```

5. Python editor spyder:

- Consoles/Open IPython consol.
- Left panel text editor
- Right panel ipython consol
- F9 run selection or current line (in recent version of spyder)

6. Python interpreter python or ipython same as python with many useful features.

Import libraries

```
# 'generic import' of math module  
import math math.sqrt(25)  
# import a function from math  
import sqrt sqrt(25)# no longer have to reference the module  
# import multiple functions at once  
from math import cos, floor  
# import all functions in a module (generally discouraged) from os import *  
# define an alias  
import numpy as np  
# show all functions in math module  
content = dir(math)
```

Data types

```
# determine the type of an object
type(2)      # returns 'int'
type(2.0)    # returns 'float'
type('two')   # returns 'str'
type(True)    # returns 'bool'
type(None)   # returns 'NoneType'
# check if an object is of a given type
isinstance(2.0, int)      # returns False
isinstance(2.0, (int, float)) # returns True
# convert an object to a given type
float(2)
int(2.9)
str(2.9)
# zero, None, and empty containers are converted to False
bool(0)
bool(None)
bool("") # empty string bool([])
          # empty list bool({})
#empty dictionary
# non-empty containers and non-zeros are converted to True
bool(2)
bool('two')
bool([2])
True
```

Math

```
# basic operations
10 + 4           # add (returns 14)
10 - 4           # subtract (returns 6)
10 * 4           # multiply (returns 40)
10 ** 4          # exponent (returns 10000)
10 / 4           # divide (returns 2 because both types are 'int')
10 / float(4)    # divide (returns 2.5)
5 % 4            # modulo (returns 1) - also known as the remainder
# force '/' in Python 2.x to perform 'true division' (unnecessary in Python 3.x)
from __future__ import division
10 / 4           # true division (returns 2.5)
10 // 4          # floor division (returns 2)
2
```

Comparisons and boolean operations

```
# comparisons (these return True)
5 > 3
5 >= 3
5 != 3
5 == 5
# boolean operations (these return True)
5 > 3 and 6 > 3 5 > 3 or 5 < 3 not False False or not False and True # evaluation order: not, and, or
True
```

Conditional statements

```
x = 3
# if statement
if x > 0:
    print('positive')
```

```

# if/else statement
if x > 0:
    print('positive')
else:
    print('zero or negative')
# if/elif/else statement
if x > 0:
    print('positive')
elif x == 0:
    print('zero')
else:
    print('negative')
# single-line if statement (sometimes discouraged)
if x > 0: print('positive')
# single-line if/else statement (sometimes discouraged)
# known as a 'ternary operator' 'positive'
if x > 0 else 'zero or negative'

```

positive	positive	positive
positive		

'positive'

Lists

```

## properties: ordered, iterable, mutable, can contain multiple data types
# create an empty list (two ways)
empty_list = []
empty_list = list()
# create a list
simpsons = ['homer', 'marge', 'bart']
# examine a list
simpsons[0]      # print element 0 ('homer')
len(simpsons) # returns the length (3)
# modify a list (does not return the list)
simpsons.append('lisa')                      # append element to end
simpsons.extend(['itchy', 'scratchy']) # append multiple elements to end
simpsons.insert(0, 'maggie')      # insert element at index 0 (shifts everything right)
simpsons.remove('bart')    # searches for first instance and removes it
simpsons.pop(0) # removes element 0 and returns it
del simpsons[0] # removes element 0 (does not return it)
simpsons[0] = 'krusty'     # replace element 0

```

```

# concatenate lists (slower than 'extend' method)
neighbors = simpsons + ['ned','rod','todd']
# find elements in a list
simpsons.count('lisa')      # counts the number of instances
simpsons.index('itchy')     # returns index of first instance
# list slicing [start:end:stride]
weekdays = ['mon','tues','wed','thurs','fri']
weekdays[0]      # element 0
weekdays[0:3]    # elements 0, 1, 2
weekdays[:3]     # elements 0, 1, 2
weekdays[3:]     # elements 3, 4
weekdays[-1]     # last element (element 4)
weekdays[::-2]   # every 2nd element (0, 2, 4)
weekdays[::-1]   # backwards (4, 3, 2, 1, 0)
# alternative method for returning the list backwards
list(reversed(weekdays))
# sort a list in place (modifies but does not return the list)
simpsons.sort()
simpsons.sort(reverse=True)      # sort in reverse
simpsons.sort(key=len)          # sort by a key
# return a sorted list (but does not modify the original list)
sorted(simpsons)
sorted(simpsons, reverse=True)
sorted(simpsons, key=len)
# create a second reference to the same list
num = [1, 2, 3]
same_num = num
same_num[0] = 0 # modifies both 'num' and 'same_num'
# copy a list (three ways)
new_num = num.copy()
new_num = num[:]
new_num = list(num)
# examine objects
id(num) == id(same_num) # returns True
id(num) == id(new_num) # returns False
num is same_num         # returns True
num is new_num # returns False
num == same_num         # returns True
num == new_num           # returns True (their contents are equivalent)
# concatenate +, replicate *
[1, 2, 3] + [4, 5, 6]
["a"] * 2 + ["b"] * 3
['a', 'a', 'b', 'b', 'b']

```

Tuples

Like lists, but they don't change size properties: ordered, iterable, immutable, can contain multiple data types

```

# create a tuple
digits = (0, 1, 'two')      # create a tuple directly
digits = tuple([0, 1, 'two']) # create a tuple from a list zero = (0,) # trailing comma is required to indicate it's a tuple
# examine a tuple
digits[2] # returns 'two'
len(digits)      # returns 3
digits.count(0)  # counts the number of instances of that value (1)
digits.index(1)  # returns the index of the first instance of that value (1)
# elements of a tuple cannot be modified # digits[2] = 2
# throws an error
# concatenate tuples
digits = digits + (3, 4)
# create a single tuple with elements repeated (also works with lists)
(3, 4) * 2        # returns (3, 4, 3, 4)
# tuple unpacking

```

```
bart = ('male', 10, 'simpson') # create a tuple
```

Strings

Properties: iterable, immutable

```
from __future__ import print_function
# create a string
s = str(42) # convert another data type into a string s = 'I like you'
# examine a string
s[0] # returns 'I'
len(s) # returns 10
# string slicing like lists
s[6] # returns 'I like'
s[7:] # returns 'you'
s[1] # returns 'u'
# basic string methods (does not modify the original string)
s.lower() # returns 'i like you'
s.upper() # returns 'I LIKE YOU'
s.startswith('I') # returns True
s.endswith('you') # returns True
s.isdigit() # returns False (returns True if every character in the string is a digit)
s.find('like') # returns index of first occurrence (2), but doesn't support regex
s.find('hate') # returns -1 since not found
s.replace('like','love') # replaces all instances of 'like' with 'love'

# split a string into a list of substrings separated by a delimiter
s.split(' ') # returns ['I', 'like', 'you']
s.split() # same thing
s2 = 'a, an, the'
s2.split(',') # returns ['a', 'an', 'the']
# join a list of strings into one string using a delimiter
stooges = ['larry','curly','moe']
''.join(stooges) # returns 'larry curly moe'
# concatenate strings
s3 = 'The meaning of life is'
s4 = '42' s3 + '' + s4 # returns 'The meaning of life is 42'
s3 + '' + str(42) # same thing
# remove whitespace from start and end of a string
s5 = ' ham and cheese '
s5.strip() # returns 'ham and cheese'
# string substitutions: all of these return 'raining cats and dogs'
'raining %s and %s' % ('cats','dogs') # old way
'raining {} and {}'.format('cats','dogs') # new way
'raining {arg1} and {arg2}'.format(arg1='cats',arg2='dogs') # named arguments
# string formatting
# more examples: http://mkaz.com/2012/10/10/python-string-format/
'pi is {:.2f}'.format(3.14159) # returns 'pi is 3.14'
```

```
# normal strings versus raw strings print('first line\nsecond line')      # normal strings allow for escaped
characters
print(r'first line\nfirst line') # raw strings treat backslashes as literal characters
```

first line second line first
linenfirst line

Dictionaries

Properties: unordered, iterable, mutable, can contain multiple data types made up of key-value pairs keys must be unique, and can be strings, numbers, or tuples values can be any type

```
# create an empty dictionary (two ways)
empty_dict = {}
empty_dict = dict()

# create a dictionary (two ways)
family = {'dad':'homer', 'mom':'marge', 'size':6}
family = dict(dad='homer', mom='marge', size=6)

# convert a list of tuples into a dictionary
list_of_tuples = [('dad','homer'), ('mom','marge'), ('size', 6)]
family = dict(list_of_tuples)

# examine a dictionary
family['dad']      # returns 'homer'
len(family)        # returns 3
family.keys()      # returns list: ['dad', 'mom', 'size']
family.values()    # returns list: ['homer', 'marge', 6]
family.items()     # returns list of tuples:
                  #[('dad', 'homer'), ('mom', 'marge'), ('size', 6)]
'mom' in family   # returns True
'marge' in family  # returns False (only checks keys)
# modify a dictionary (does not return the dictionary)
family['cat'] = 'snowball'    # add a new entry
family['cat'] = 'snowball ii' # edit an existing entry
del family['cat']           # delete an entry
family['kids'] = ['bart', 'lisa'] # value can be a list
family.pop('dad')          # removes an entry and returns the value ('homer')
family.update({'baby':'maggie', 'grandpa':'abe'}) # add multiple entries
# accessing values more safely with 'get'
family['mom']              # returns 'marge'
family.get('mom')          # same thing
try:
    family['grandma'] # throws an error
except KeyError as e:
    print("Error", e)
family.get('grandma')       # returns None
family.get('grandma', 'not found') # returns 'not found' (the default)
# accessing a list element within a dictionary
family['kids'][0]           # returns 'bart'
family['kids'].remove('lisa') # removes 'lisa'
# string substitution using a dictionary
'youngest child is %(baby)s' % family           # returns 'youngest child is maggie'
```

Sets

Like dictionaries, but with keys only (no values) properties: unordered, iterable, mutable, can contain multiple data types made up of unique elements (strings, numbers, or tuples)

```
# create an empty set
empty_set = set()

# create a set
languages = {'python', 'r', 'java'}           # create a set directly
```

```

snakes = set(['cobra', 'viper', 'python'])           # create a set from a list
# examine a set
len(languages)                                     # returns 3
'python' in languages                            # returns True
# set operations
languages & snakes # returns intersection: {'python'}
languages | snakes      # returns union: {'cobra', 'r', 'java', 'viper', 'python'}
languages - snakes # returns set difference: {'r', 'java'}
snakes - languages      # returns set difference: {'cobra', 'viper'}
# modify a set (does not return the set)
languages.add('sql')      # add a new element
languages.add('r') # try to add an existing element (ignored, no error)
languages.remove('java') # remove an element
try:
    languages.remove('c') # try to remove a non-existing element (throws an error)
except KeyError as e:
    print("Error", e)
languages.discard('c')   # removes an element if present, but ignored otherwise
languages.pop() # removes and returns an arbitrary element
languages.clear() # removes all elements
languages.update('go', 'spark') # add multiple elements (can also pass a list or set)
# get a sorted list of unique elements from a list
sorted(set([9, 0, 2, 1, 0])) # returns [0, 1, 2, 9]

```

Error 'c'
[0, 1, 2, 9]

Functions

```

# define a function with no arguments and no return values
def print_text(): print('this is text')
# call the function
print_text()
# define a function with one argument and no return values
def print_this(x): print(x)
# call the function
print_this(3)      #prints 3
n = print_this(3) # prints 3, but doesn't assign 3 to n
                  # because the function has no return statement
# define a function with one argument and one return value
def square_this(x):
    return x ** 2
# include an optional docstring to describe the effect of a function
def square_this(x):
    """Return the square of a number."""
# call the function
square_this(3)      # prints 9
var = square_this(3)      # assigns 9 to var, but does not print 9
# default arguments
def power_this(x, power=2):
    return x ** power
power_this(2)      # 4
power_this(2, 3) # 8
# use 'pass' as a placeholder if you haven't written the function body
def stub():
    pass
# return two values from a single function
def min_max(nums):
    return min(nums), max(nums)
# return values can be assigned to a single variable as a tuple
nums = [1, 2, 3]
min_max_num = min_max(nums) # min_max_num = (1, 3)
# return values can be assigned into multiple variables using tuple unpacking

```

```
min_num, max_num = min_max(nums)      # min_num = 1, max_num = 3
```

this is text 3

3

Loops

```
# range returns a list of integers
range(0, 3)      # returns [0, 1, 2]: includes first value but excludes second value
range(3) # same thing: starting at zero is the default
range(0, 5, 2) # returns [0, 2, 4]: third argument specifies the 'stride'
# for loop (not recommended)
fruits = ['apple', 'banana', 'cherry']
for i in range(len(fruits)):
    print(fruits[i].upper())
# alternative for loop (recommended style)
for fruit in fruits:
    print(fruit.upper())
# use range when iterating over a large sequence to avoid actually creating the integer list in memory
for i in range(10**6):
    pass
# iterate through two things at once (using tuple unpacking)
family = {'dad':'homer', 'mom':'marge', 'size':6}
for key, value in family.items():
    print(key, value)
# use enumerate if you need to access the index value within the loop
for index, fruit in enumerate(fruits):
    print(index, fruit)
# for/else loop
for fruit in fruits:
    if fruit == 'banana':
        print("Found the banana!")
        break # exit the loop and skip the 'else' block
else:
    # this block executes ONLY if the for loop completes without hitting 'break'
    print("Can't find the banana")
# while loop
count = 0
while count < 5:
    print("This will print 5 times")
    count += 1 # equivalent to 'count = count + 1'
```

```

APPLE
BANANA
CHERRY
APPLE
BANANA
CHERRY
mom marge
dad homer
size 6 0
apple
1 banana
2 cherry
Found the banana!
This will print 5 times

```

List comprehensions

```

# for loop to create a list of cubes
nums = [1, 2, 3, 4, 5]
cubes = [] for num in nums:
    cubes.append(num**3)
# equivalent list comprehension
cubes = [num**3 for num in nums] #[1, 8, 27, 64, 125]
# for loop to create a list of cubes of even numbers
cubes_of_even = [] for num in nums:
    if num % 2 == 0:
        cubes_of_even.append(num**3)
# equivalent list comprehension
# syntax: [expression for variable in iterable if condition]
cubes_of_even = [num**3 for num in nums if num % 2 == 0] #[8, 64]
# for loop to cube even numbers and square odd numbers
cubes_and_squares = [] for num in nums:
    if num % 2 == 0:
        cubes_and_squares.append(num**3)
    else:
        cubes_and_squares.append(num**2)
# equivalent list comprehension (using a ternary expression)
# syntax: [true_condition if condition else false_condition for variable in iterable]
cubes_and_squares = [num**3 if num % 2 == 0 else num**2 for num in nums] #[1, 8, 9, 64, 25]
# for loop to flatten a 2d-matrix
matrix = [[1, 2], [3, 4]]
items = [] for row in matrix:
    for item in row:
        items.append(item)
# equivalent list comprehension
items = [item for row in matrix
            for item in row] #[1, 2, 3, 4]
# set comprehension
fruits = ['apple', 'banana', 'cherry']
unique_lengths = {len(fruit) for fruit in fruits} # {5, 6}
# dictionary comprehension
fruit_lengths = {fruit:len(fruit) for fruit in fruits} # {'apple': 5, 'banana': 6, 'cherry': 6}

```

Exceptions handling

```
dct = dict(a=[1, 2], b=[4, 5])
key = 'c'
try:
    dct[key]
except:
    print("Key %s is missing. Add it with empty value" % key)
    dct['c'] = []
print(dct)
```

Key c is missing. Add it with empty value
{'c': [], 'b': [4, 5], 'a': [1, 2]}

Basic operating system interfaces (os)

```
import os
import tempfile
tmpdir = tempfile.gettempdir()
# list containing the names of the entries in the directory given by path.
os.listdir(tmpdir)
# Change the current working directory to path.
os.chdir(tmpdir)
# Get current working directory.
print('Working dir:', os.getcwd())
# Join paths mytmpdir = os.path.join(tmpdir, "foobar")
# Create a directory if not
os.path.exists(mytmpdir):
os.mkdir(mytmpdir)
filename = os.path.join(mytmpdir, "myfile.txt")
print(filename)
# Write
lines = ["Dans python tout est bon", "Enfin, presque"]
## write line by line
fd = open(filename, "w")
fd.write(lines[0] + "\n")
fd.write(lines[1] + "\n")
fd.close()
## use a context manager to automatically close your file
with open(filename, 'w') as f:
    for line in lines:
        f.write(line + '\n')
# Read
## read one line at a time (entire file does not have to fit into memory)
f = open(filename, "r")
f.readline()           # one string per line (including newlines)
f.readline()           # next line
f.close()
## read one line at a time (entire file does not have to fit into memory)
f = open(filename, 'r')
f.readline()           # one string per line (including newlines)
f.readline()           # next line
f.close()
## read the whole file at once, return a list of lines
f = open(filename, 'r')
f.readlines() # one list, each line is one string
f.close()
```

Basic operating system interfaces (os)

```
## use list comprehension to duplicate readlines without reading entire file at once
f = open(filename, 'r')
[line for line in f]
f.close()
## use a context manager to automatically close your file with
open(filename, 'r') as f:
lines = [line for line in f]
```

Working dir: /tmp

```
/tmp/foobar/myfile.txt
```

Object Oriented Programming (OOP)

Sources

- http://python-textbook.readthedocs.org/en/latest/Object_Oriented_Programming.html

Principles

- Encapsulate data (attributes) and code (methods) into objects.
- Class = template or blueprint that can be used to create objects.
- An object is a specific instance of a class.
- Inheritance: OOP allows classes to inherit commonly used state and behaviour from other classes. Reduce code duplication
- Polymorphism: (usually obtained through polymorphism) calling code is agnostic as to whether an object belongs to a parent class or one of its descendants (abstraction, modularity). The same method called on 2 objects of 2 different classes will behave differently.

```
import math
class Shape2D:
    def area(self):
        raise NotImplementedError()
# __init__ is a special method called the constructor
# Inheritance + Encapsulation
class Square(Shape2D):
    def __init__(self, width):
        self.width = width
    def area(self): return self.width ** 2
class Disk(Shape2D):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * self.radius ** 2
shapes = [Square(2), Disk(3)]

# Polymorphism print([s.area())
for s in shapes]:
    s = Shape2D() try:
        s.area()
    except
        NotImplementedError as e: print("NotImplementedError")
[4, 28.274333882308138]
NotImplementedError
```

Exercises

Exercise 1: functions

Create a function that acts as a simple calculator If the operation is not specified, default to addition If the operation is misspecified, return an error message Ex: calc(3,5) returns 8 Ex: calc(1,2,"something") returns error message

Exercise 2: functions + list + loop

Given a list of numbers, return a list where all adjacent duplicate elements have been reduced to a single element. Ex:

[1,2,2,3,2] returns [1,2,3,2]. You may create a new list or modify the passed in list.

Remove all duplicate values (adjacent or not) Ex: [1,2,2,3,2] returns [1,2,3]

Exercise 3: File I/O

Copy/past the bsd 4 clause license into a text file. Read, the file (assuming this file could be huge) and could not the occurrences of each word within the file. Words are separated by whitespace or new line characters.

Exercise 4: OOP

1. Create a class Employee with 2 attributes provided in the constructor: name, years_of_service. With one method salary with is obtained by $1500 + 100 * \text{years_of_service}$.
2. Create a subclass Manager which redefine salary method $2500 + 120 * \text{years_of_service}$.
3. Create a small dictionnary-based database where the key is the employee's name. Populate the database with: samples = Employee('lucy', 3), Employee('john', 1), Manager('julie', 10), Manager('paul', 3)
4. Return a table of made name, salary rows, ie. a list of list [[name, salary]]
5. Compute the average salary

CHAPTER THREE

NUMPY: ARRAYS AND MATRICES

NumPy is an extension to the Python programming language, adding support for large, multi-dimensional (numerical) arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

Sources: Kevin Markham: <https://github.com/justmarkham>

```
from __future__ import print_function
import numpy as np
```

Create arrays

```
# create ndarrays from lists
# note: every element must be the same type (will be converted if possible)
data1 = [1, 2, 3, 4, 5]      # list
arr1 = np.array(data1)        # 1d array
data2 = [range(1, 5), range(5, 9)] # list of lists
arr2 = np.array(data2)        # 2d array
arr2.tolist()                # convert array back to list
# examining arrays
arr1.dtype                  # float64
arr2.dtype                  # int32
arr2.ndim                   # 2
arr2.shape                   # (2, 4) - axis 0 is rows, axis 1 is columns
arr2.size # 8 - total number of elements
len(arr2) # 2 - size of first dimension (aka axis)
# create special arrays
np.zeros(10)
np.zeros((3, 6))
np.ones(10)
np.linspace(0, 1, 5)         # 0 to 1 (inclusive) with 5 points
np.logspace(0, 3, 4)         # 10^0 to 10^3 (inclusive) with 4 points
# arange is like range, except it returns an array (not a list)
int_array = np.arange(5)
float_array = int_array.astype(float)
```

Reshaping

```
matrix= np.arange(10,dtype=float).reshape((2,5)) print(matrix.shape)
print(matrix.reshape(5, 2))
# Add an axis
a = np.array([0, 1])
a_col = a[:, np.newaxis]
# array([[0],
#        [1]])
# Transpose
a_col.T
#array([[0, 1]])
```

Stack arrays

Stack flat arrays in columns

Selection

Single item

```
arr1[0]           # 0th element (slices like a list)
arr2[0, 3]         # row 0, column 3: returns 4
arr2[0][3]         # alternative syntax
```

Slicing

```
arr2[0, :]          # row 0: returns 1d array ([1, 2, 3, 4])
arr2[:, 0]          # column 0: returns 1d array ([1, 5])
arr2[:, :2]          # columns strictly before index 2 (2 first columns)
arr2[:, 2:]          # columns after index 2 included
arr2[:, 1:4]         # columns between index 1 (included) and 4 (excluded)
```

Views and copies

```
arr = np.arange(10)
arr[5:8]             # returns [5, 6, 7]
arr[5:8] = 12         # all three values are overwritten (would give error on a list)

arr_view = arr[5:8]      # creates a "view" on arr, not a copy
arr_view[:] = 13         # modifies arr_view AND arr
arr_copy = arr[5:8].copy()  # makes a copy instead
arr_copy[:] = 14         # only modifies arr_copy
```

using boolean arrays

```
arr[arr > 5]
```

Boolean selection return a view which authorizes the modification of the original array

```

arr[arr > 5] = 0
print(arr)
names = np.array(['Bob', 'Joe', 'Will', 'Bob'])
names == 'Bob' # returns a boolean array
names[names != 'Bob'] # logical selection
(names == 'Bob') | (names == 'Will') # keywords "and/or" don't work with boolean arrays
names[names != 'Bob'] = 'Joe' # assign based on a logical selection
np.unique(names) # set function

```

Vectorized operations

```

nums = np.arange(5)
nums * 10 # multiply each element by 10
nums = np.sqrt(nums) # square root of each element
np.ceil(nums) # also floor, rint (round to nearest int)
np.isnan(nums) # checks for NaN
nums +
np.arange(5) # add element-wise
np.maximum(nums, np.array([1, -2, 1, -4, 5])) # compare element-wise
# Compute Euclidean distance between 2 vectors
vec1 = np.random.randn(10)
vec2 = np.random.randn(10)
dist = np.sqrt(np.sum((vec1 - vec2) ** 2))
# math and stats
rnd = np.random.randn(4, 2) # random normals in 4x2 array
rnd.mean()
rnd.std()
rnd.argmin() # index of minimum element
rnd.sum()
rnd.sum(axis=0) # sum of columns
rnd.sum(axis=1) # sum of rows
# methods for boolean arrays
(rnd > 0).sum() # counts number of positive values
(rnd > 0).any() # checks if any value is True
(rnd > 0).all() # checks if all values are True
# reshape, transpose, flatten
nums = np.arange(32).reshape(8, 4) # creates 8x4 array
nums.T # transpose
nums.flatten() # flatten
# random numbers
np.random.seed(1234) # Set the seed
np.random.rand(2, 3) # 2 x 3 matrix in [0, 1]
np.random.randn(10) # random normals (mean 0, sd 1)
np.random.randint(0, 2, 10) # 10 randomly picked 0 or 1

```

Exercises

Given the array:

```
X = np.random.randn(4, 2) # random normals in 4x2 array
```

- For each column find the row index of the minimum value.
- Write a function standardize(X) that return an array whose columns are centered and scaled (by std-dev).

¹ Vectorized operations

CHAPTER FOUR

PANDAS: DATA MANIPULATION

It is often said that 80% of data analysis is spent on the cleaning and preparing data. To get a handle on the problem, this chapter focuses on a small, but important, aspect of data manipulation and cleaning with Pandas.

Sources:

- Kevin Markham: <https://github.com/justmarkham>
- Pandas doc: <http://pandas.pydata.org/pandas-docs/stable/index.html>

Data structures

- Series is a one-dimensional labelled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call `pd.Series([1,3,5,np.nan,6,8])`
- DataFrame is a 2-dimensional labelled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It stems from the *R* `data.frame()` object.

```
from __future__ import print_function
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Create DataFrame

```
columns = ['name', 'age', 'gender', 'job']
user1 = pd.DataFrame([{"alice": 19, "F", "student"}, {"john": 26, "M", "student"}], columns=columns)
user2 = pd.DataFrame([{"eric": 22, "M", "student"}, {"paul": 58, "F", "manager"}], columns=columns)
user3 = pd.DataFrame(dict(name=['peter', 'julie'], age=[33, 44],
                           gender=['M','F'], job=['engineer', 'scientist']))
```

Concatenate DataFrame

```
user1.append(user2)
users = pd.concat([user1, user2, user3])
print(users)
```

#	age	gender	job	name
#0	19	F	student	alice
#1	26	M	student	john
#0	22	M	student	eric
#1	58	F	manager	paul
#0	33	M	engineer	peter
#1	44	F	scientist	julie

Join DataFrame

```
user4 = pd.DataFrame(dict(name=['alice', 'john', 'eric', 'julie'], height=[165, 180, 175, 171]))
print(user4)
#   height    name
#0    165  alice
#1    180   john
#2    175   eric
#3    171  julie
# Use intersection of keys from both frames
merge_inter = pd.merge(users, user4, on="name")
print(merge_inter)
#   age  gender     job    name  height
#0  19      F  student  alice    165
#1  26      M  student   john    180
#2  22      M  student    eric    175
#3  44      F scientist  julie    171
# Use union of keys from both frames
users = pd.merge(users, user4, on="name", how='outer')
print(users)
#   age  gender     job    name  height
#0  19      F  student  alice    165
#1  26      M  student   john    180
#2  22      M  student    eric    175
#3  58      F  manager   paul    NaN
#4  33      M  engineer  peter    NaN
#5  44      F scientist  julie    171
```

Summarizing

Columns selection

```
users['gender']    # select one column
type(users['gender'])    # Series
users.gender      # select one column using the DataFrame
# select multiple columns
users[['age', 'gender']]    # select two columns

my_cols = ['age', 'gender']          # or, create a list...
users[my_cols]                    # ...and use that list to select columns
type(users[my_cols])            # DataFrame
```

Rows selection

```
# iloc is strictly integer position based
df = users.copy()
df.iloc[0] # first row
df.iloc[0, 0] # first item of first row
df.iloc[0, 0] = 55
for i in range(users.shape[0]):
    row = df.iloc[i]
    row.age *= 100 # setting a copy, and not the original frame data.
print(df) # df is not modified
# ix supports mixed integer and label based access.
df = users.copy()
df.ix[0]          # first row
df.ix[0, "age"] # first item of first row
df.ix[0, "age"] = 55
for i in range(df.shape[0]):
    df.ix[i, "age"] *= 10
print(df) # df is modified
```

Rows selection / filtering

```
# simple logical filtering
users[users.age < 20]
# only show users with age < 20
young_bool = users.age < 20
# or, create a Series of booleans...
users[young_bool]
# ...and use that Series to filter rows
users[users.age < 20].job
# select one column from the filtered results
# advanced logical filtering
users[users.age < 20][['age', 'job']]
# select multiple columns
users[(users.age > 20) & (users.gender=='M')]
# use multiple conditions
users[users.job.isin(['student', 'engineer'])]
# filter specific values
```

Sorting

```
df = users.copy()
df.age.sort_values()      # only works for a Series
df.sort_values(by='age')  # sort rows by a specific column
df.sort_values(by='age', ascending=False) # use descending order instead
df.sort_values(by=['job', 'age'])       # sort by multiple columns
df.sort_values(by=['job', 'age'], inplace=True) # modify df
```

Reshaping by pivoting

```
# "Unpivots" a DataFrame from wide format to long (stacked) format,
staked = pd.melt(users, id_vars="name", var_name="variable", value_name="value")
print(staked)
#      name variable     value
#0    alice     age      19
#1    john     age      26
#2    eric     age      22
#3    paul     age      58
#4   peter     age      33
#5   julie     age      44
#6    alice    gender      F
#
#11 julie    gender      F
#12 alice    job    student
#
#17 julie    job    scientist
#18 alice    height     165
#
#23 julie    height     171
# "pivots" a DataFrame from long (stacked) format to wide format,
print(staked.pivot(index='name', columns='variable', values='value'))
#variable age gender height          job
#name
#alice    19      F    165  student
#eric     22      M    175  student
#john     26      M    180  student
#julie    44      F    171  scientist
#paul     58      F    NaN  manager
#peter    33      M    NaN  engineer
```

Quality control: duplicate data

```
df = users.append(df.iloc[0], ignore_index=True)
print(df.duplicated())                                # Series of booleans
# (True if a row is identical to a previous row)
df.duplicated().sum()      # count of duplicates
df[df.duplicated()] # only show duplicates
df.age.duplicated()        # check a single column for duplicates
df.duplicated(['age', 'gender']).sum() # specify columns for finding duplicates
df = df.drop_duplicates() # drop duplicate rows
```

Quality control: missing data

```
# missing values are often just excluded df =
users.copy()
df.describe(include='all')                           # excludes missing values
# find missing values in a Series
df.height.isnull() # True if NaN, False otherwise
df.height.notnull() # False if NaN, True otherwise
df[df.height.notnull()] # only show rows where age is not NaN
df.height.isnull().sum() # count the missing values
# find missing values in a DataFrame
df.isnull() # DataFrame of booleans
df.isnull().sum() # calculate the sum of each column
# Strategy 1: drop missing values
df.dropna() # drop a row if ANY values are missing
df.dropna(how='all') # drop a row only if ALL values are missing
# Strategy2: fill in missing values
df.height.mean()
df = users.copy()
df.ix[df.height.isnull(), "height"] = df["height"].mean()
```

Rename values

```
df = users.copy()
print(df.columns) df.columns = ['age', 'genre', 'travail', 'nom', 'taille']
df.travail = df.travail.map({'student':'etudiant', 'manager':'manager',
                            'engineer':'ingenieur', 'scientist':'scientific'})
assert df.travail.isnull().sum() == 0
```

Dealing with outliers

```
size = pd.Series(np.random.normal(loc=175, size=20, scale=10))
# Corrupt the first 3 measures
size[:3] += 500
```

Based on parametric statistics: use the mean

Assume random variable follows the normal distribution Exclude data outside 3 standard-deviations: - Probability that a sample lies within 1

sd: 68.27% - Probability that a sample lies within 3

sd: 99.73% ($68.27 + 2 * 15.73$)

https://fr.wikipedia.org/wiki/Loi_normale#/media/File:Boxplot_vs_PDF.svg

```
size_outlr_mean = size.copy()
size_outlr_mean[((size - size.mean()).abs() > 3 * size.std())] = size.mean()
print(size_outlr_mean.mean())
```

Based on non-parametric statistics: use the median

Median absolute deviation (MAD), based on the median, is a robust non-parametric statistics.
https://en.wikipedia.org/wiki/Median_absolute_deviation

```
mad = 1.4826 * np.median(np.abs(size - size.median()))
size_outlr_mad = size.copy()
size_outlr_mad[((size - size.median()).abs() > 3 * mad)] = size.median()
print(size_outlr_mad.mean(), size_outlr_mad.median())
```

Groupby

```
for grp, data in users.groupby("job"):
    print(grp, data)
```

File I/O

csv

```
import tempfile, os.path
tmpdir = tempfile.gettempdir()
csv_filename = os.path.join(tmpdir, "users.csv")
users.to_csv(csv_filename, index=False)
other = pd.read_csv(csv_filename)
```

Read csv from url

```
url = 'https://raw.github.com/neurospin/pystatsml/master/data/salary_table.csv'
salary = pd.read_csv(url)
```

Excel

```
xls_filename = os.path.join(tmpdir, "users.xlsx")
users.to_excel(xls_filename, sheet_name='users', index=False)
pd.read_excel(xls_filename, sheetname='users')
# Multiple sheets
with pd.ExcelWriter(xls_filename) as writer:
    users.to_excel(writer, sheet_name='users', index=False)
    df.to_excel(writer, sheet_name='salary', index=False)
```

Groupby

```
pd.read_excel(xls_filename, sheetname='users')
pd.read_excel(xls_filename, sheetname='salary')
```

Exercises

Data Frame

1. Read the iris dataset at '<https://raw.github.com/neurospin/pystatsml/master/data/iris.csv>'
2. Print column names
3. Get numerical columns
4. For each species compute the mean of numerical columns and store it in a stats table like:

	species	sepal_length	sepal_width	petal_length	petal_width
0	Setosa	5.006	3.428	1.462	0.246
1	Versicolor	5.936	2.770	4.260	1.326
2	Virginica	6.588	2.974	5.552	2.026

Missing data

Add some missing data to the previous table users:

```
df = users.copy()
df.ix[[0, 2], "age"] = None
df.ix[[1, 3], "gender"] = None
```

1. Write a function `fillmissing_with_mean(df)` that fill all missing value of numerical column with the mean of the current columns.

- Save the original users and “imputed” frame in a single excel file “users.xlsx” with 2 sheets: original, imputed.

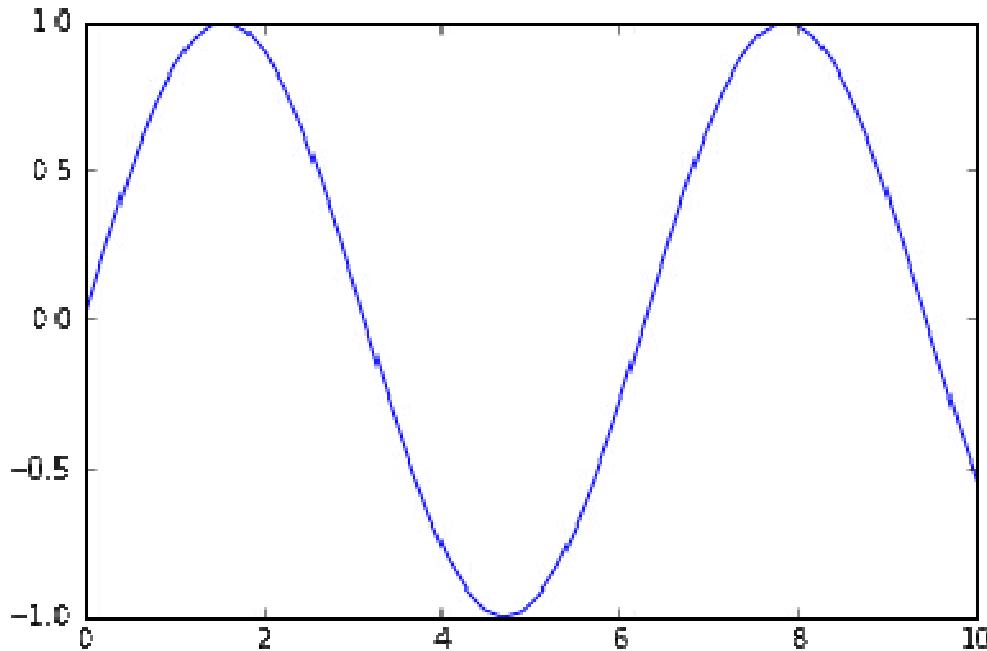
CHAPTER FIVE

MATPLOTLIB: PLOTTING

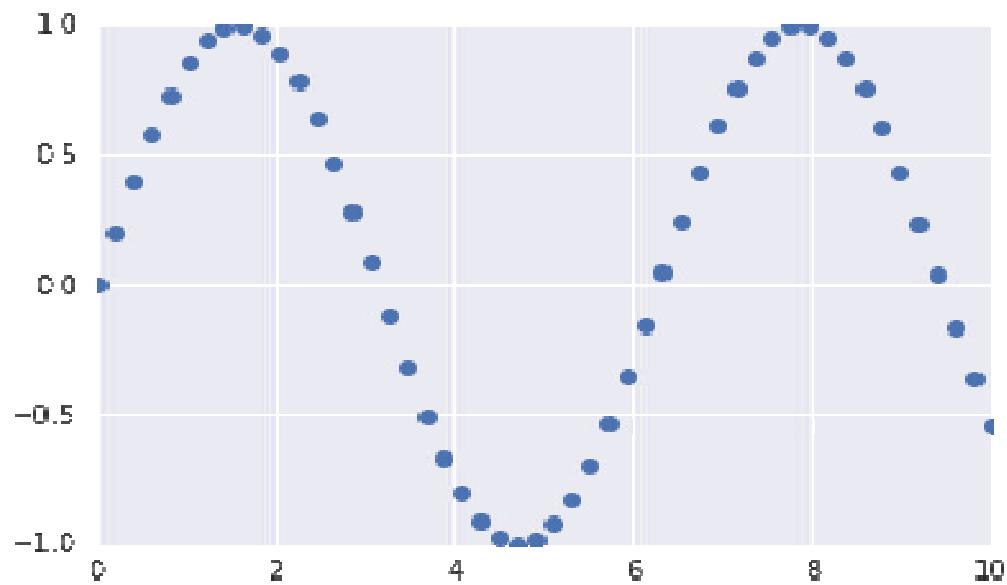
Sources - Nicolas P. Rougier:<http://www.labri.fr/perso/nrougier/teaching/matplotlib>
<https://www.kaggle.com/benhamner/d/uciml/iris/python-data-visualizations>

Preamble about the F-distribution Basic plots

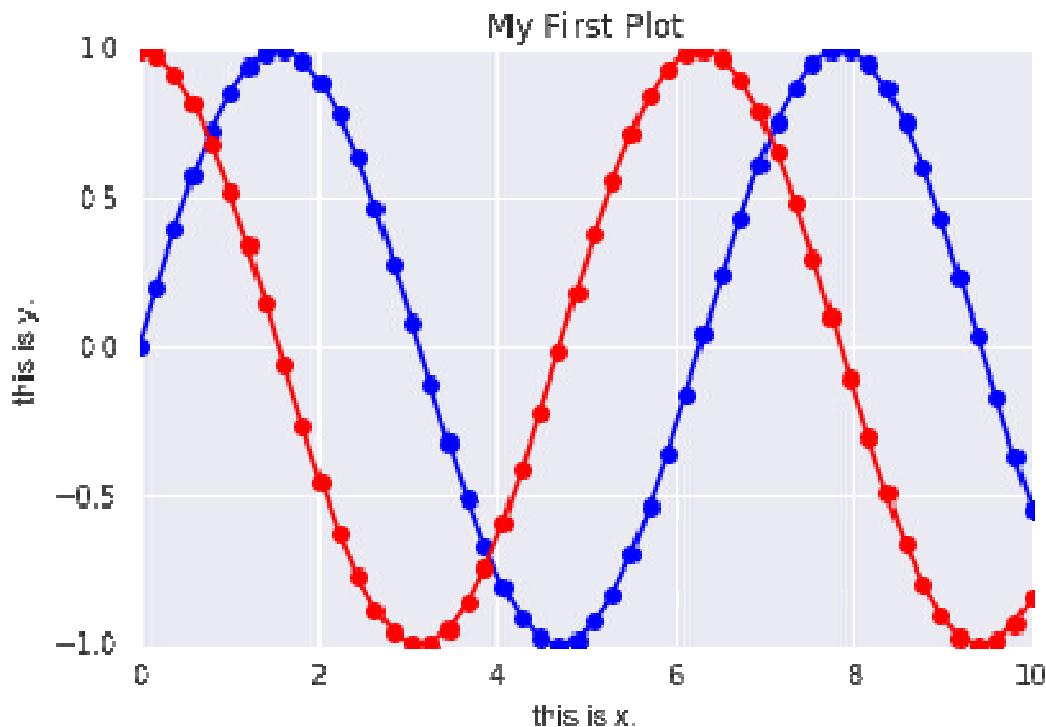
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
x = np.linspace(0, 10, 50)
sinus = np.sin(x)
plt.plot(x, sinus)
plt.show()
```



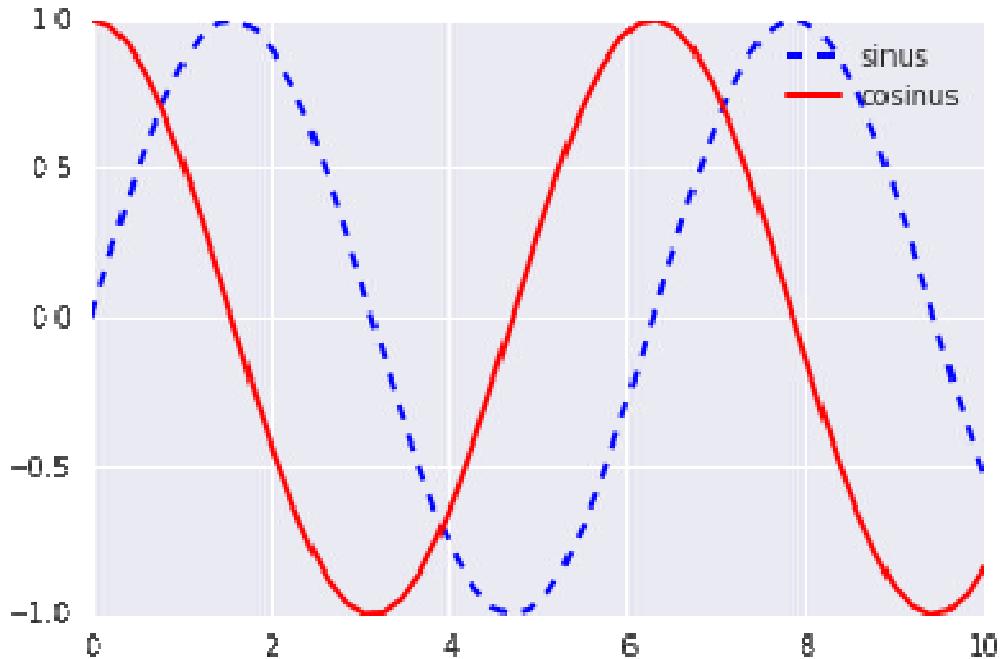
```
plt.plot(x, sinus, "o")
plt.show()
# use plt.plot to get color / marker abbreviations
```



```
cosinus = np.cos(x)
plt.plot(x, sinus, "-b", x, sinus, "ob", x, cosinus, "-r", x, cosinus, "or")
plt.xlabel('this is x!')
plt.ylabel('this is y!')
plt.title('My First Plot')
plt.show()
```



```
# Step by step
plt.plot(x, sinus, label='sinus', color='blue', linestyle='--', linewidth=2)
plt.plot(x, cosinus, label='cosinus', color='red', linestyle='-', linewidth=2)
plt.legend()
plt.show()
```



Scatter (2D) plots

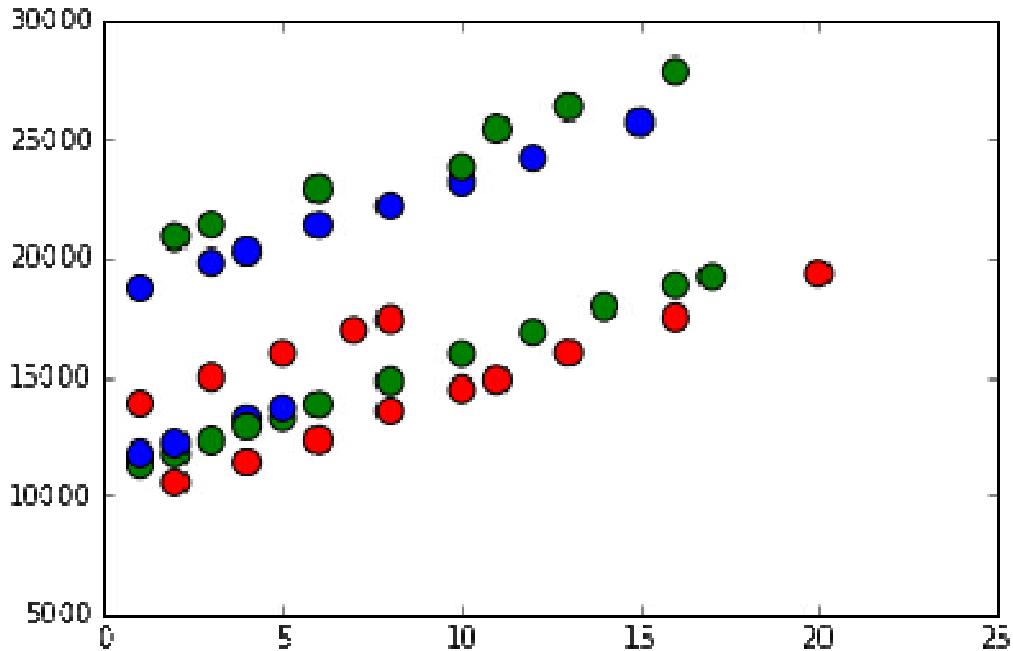
Load dataset

```
import pandas as pd
try:
    salary = pd.read_csv("../data/salary_table.csv")
except:
    url = 'https://raw.github.com/duchesnay/pylearn-doc/master/data/salary_table.csv'
    salary = pd.read_csv(url)
df = salary
```

Simple scatter with colors

```
colors = colors_edu = {'Bachelor':'r', 'Master':'g', 'Ph.D':'blue'}
plt.scatter(df['experience'], df['salary'], c=df['education'].apply(lambda x: colors[x]), s=100)
```

```
<matplotlib.collections.PathCollection at 0x7fc113f387f0>
```



Scatter plot with colors and symbols

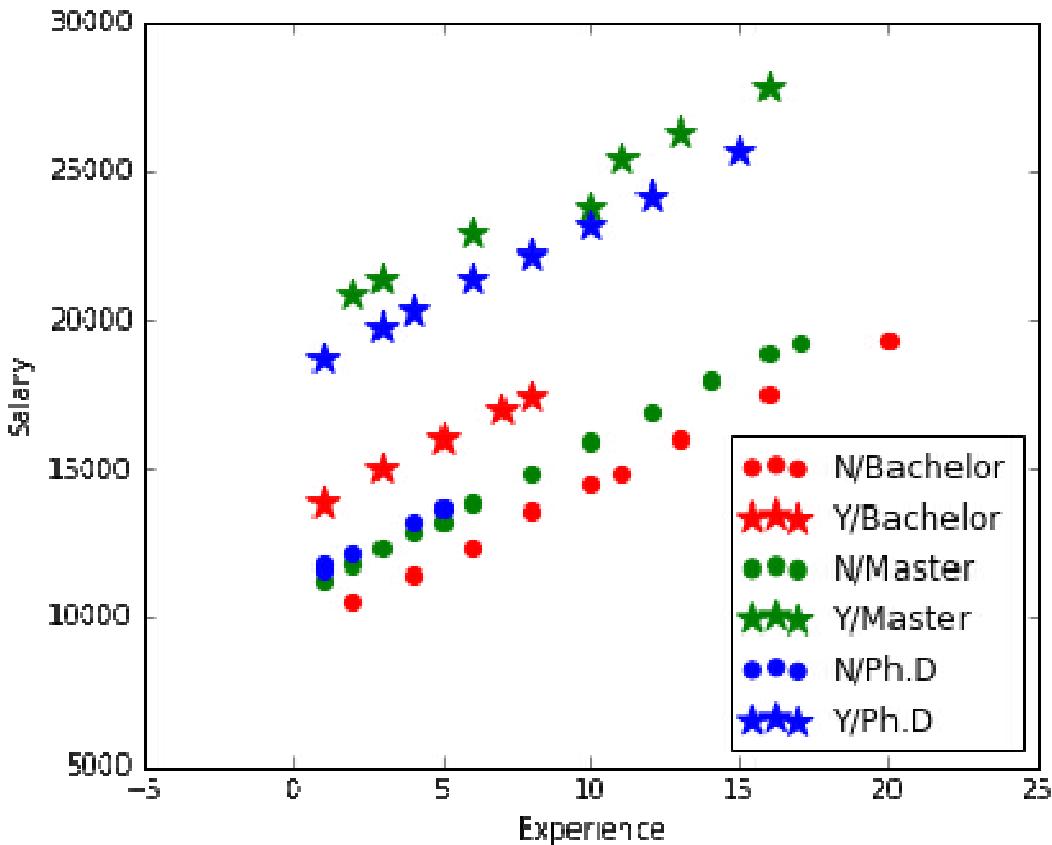
```

## Figure size
plt.figure(figsize=(6,5))
## Define colors / symbols manually
symbols_manag = dict(Y='*', N='.')
colors_edu = {'Bachelor':r, 'Master':g, 'Ph.D':blue}

## group by education x management => 6 groups
for values, d in salary.groupby(['education','management']):
    edu, manager = values
    plt.scatter(d['experience'], d['salary'], marker=symbols_manag[manager],
                color=colors_edu[edu], s=150, label=manager+"/"+edu)

## Set labels
plt.xlabel('Experience')
plt.ylabel('Salary')
plt.legend(loc=4) # lower right
plt.show()

```



Saving Figures

```
#### bitmap format
plt.plot(x,sinus)
plt.savefig("sinus.png")
plt.close()
# Prefer vectorial format (SVG: Scalable Vector Graphics) can be edited with
# Inkscape, Adobe Illustrator, Blender, etc.
plt.plot(x, sinus)
plt.savefig("sinus.svg")
```

Saving Figures

```
plt.close()
# Or pdf
plt.plot(x, sinus)
plt.savefig("sinus.pdf")
plt.close()
```

Exploring data (with seaborn)

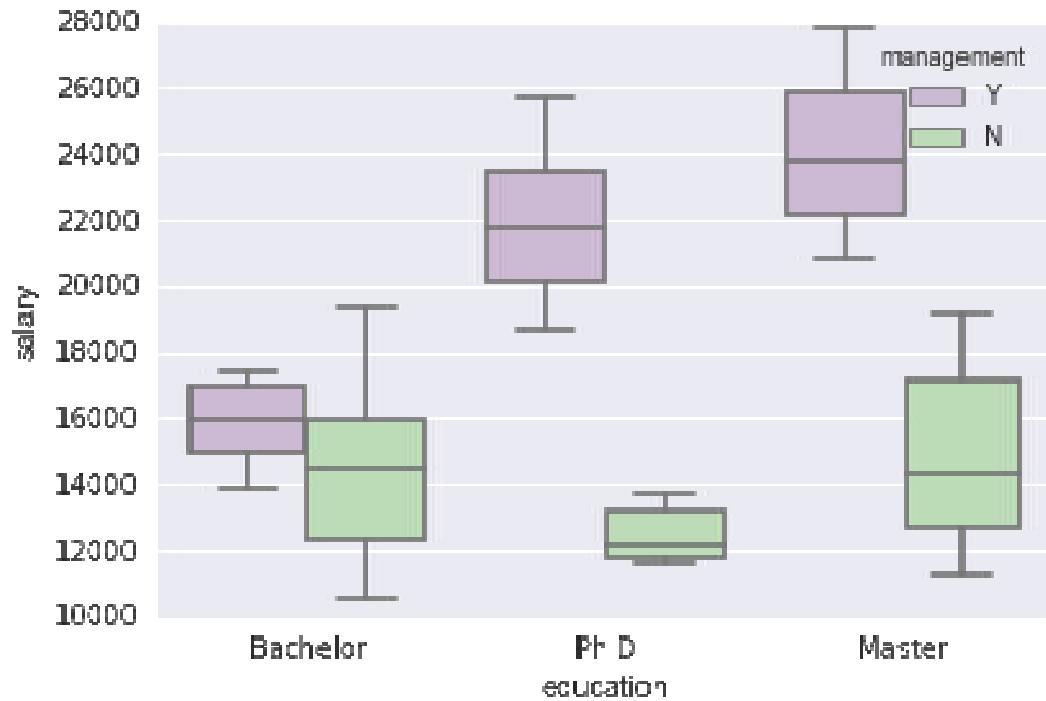
Sources: <http://stanford.edu/~mwaskom/software/seaborn>

Install using: pip install -U --user seaborn

Boxplot

Box plots are non-parametric: they display variation in samples of a statistical population without making any assumptions of the underlying statistical distribution.

```
import seaborn as sns
sns.boxplot(x="education", y="salary", hue="management", data=salary, palette="PRGn")
<matplotlib.axes._subplots.AxesSubplot at 0x7f3fd161b38>
```



```
sns.boxplot(x="management", y="salary", hue="education", data=salary, palette="PRGn")
```

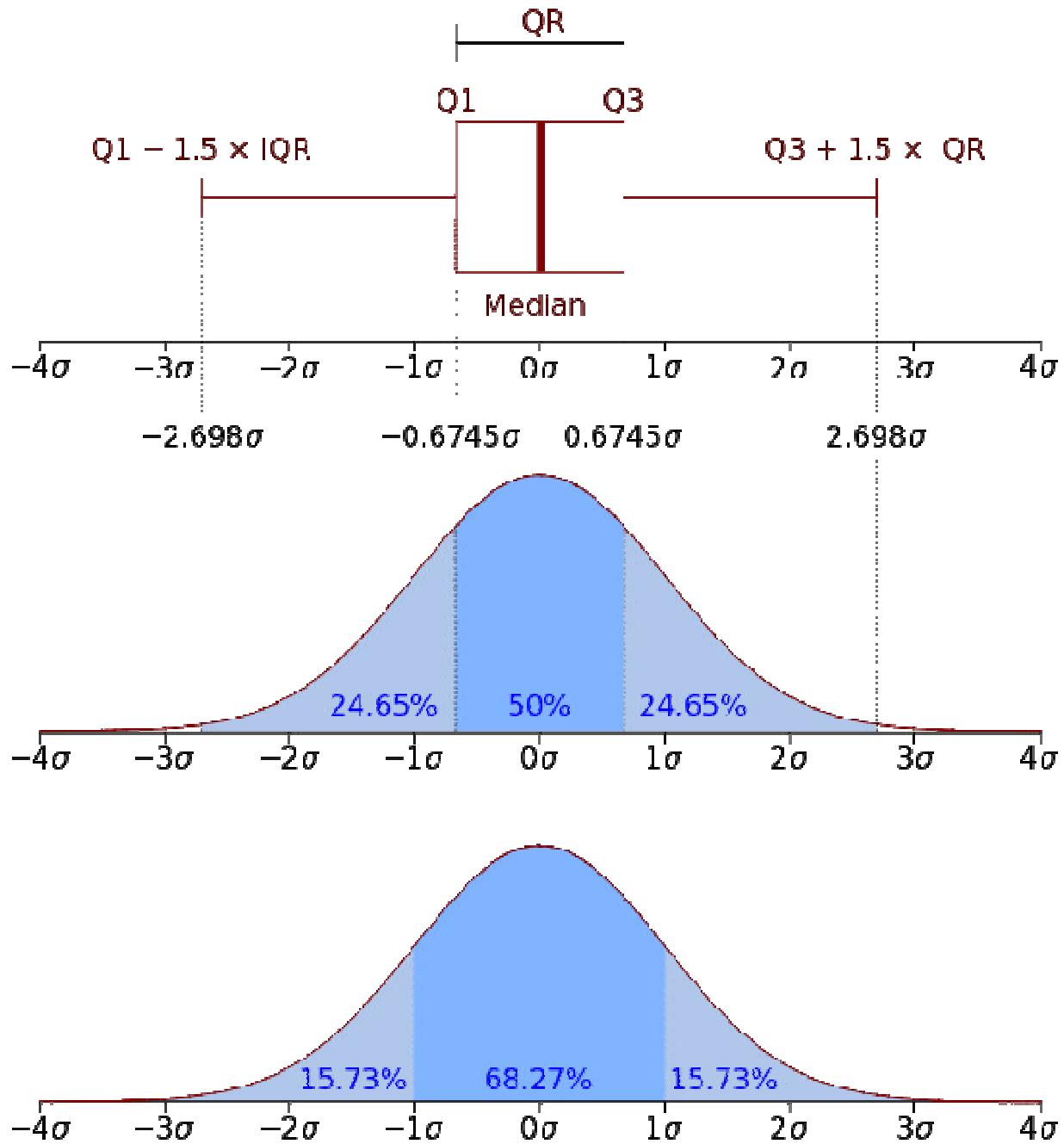
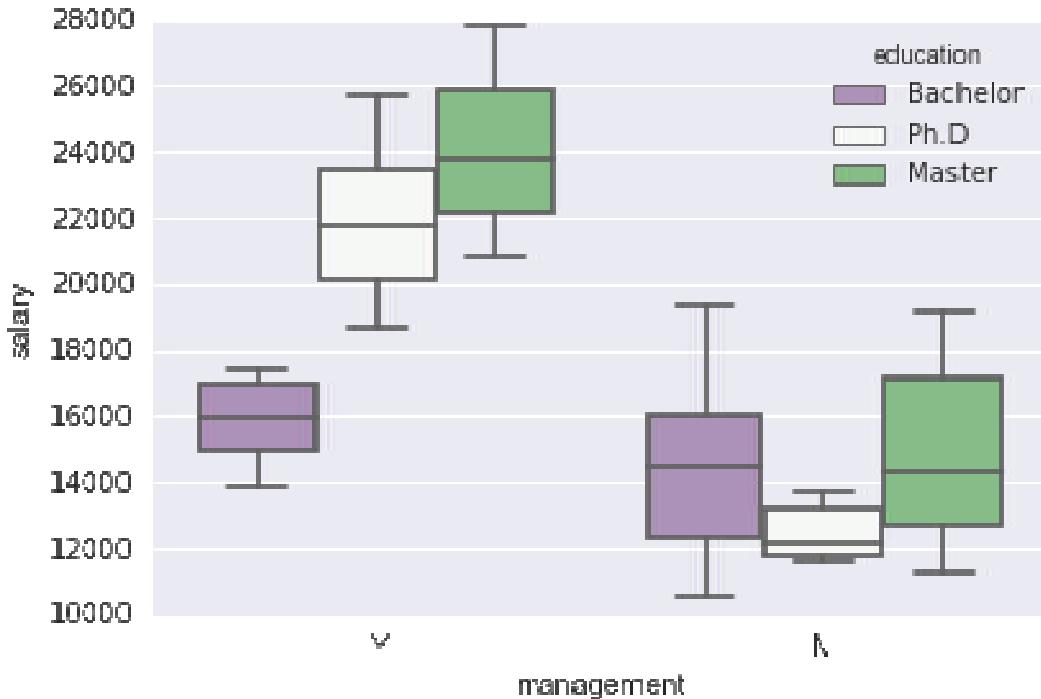


Fig. 5.1

Exploring data (with seaborn)

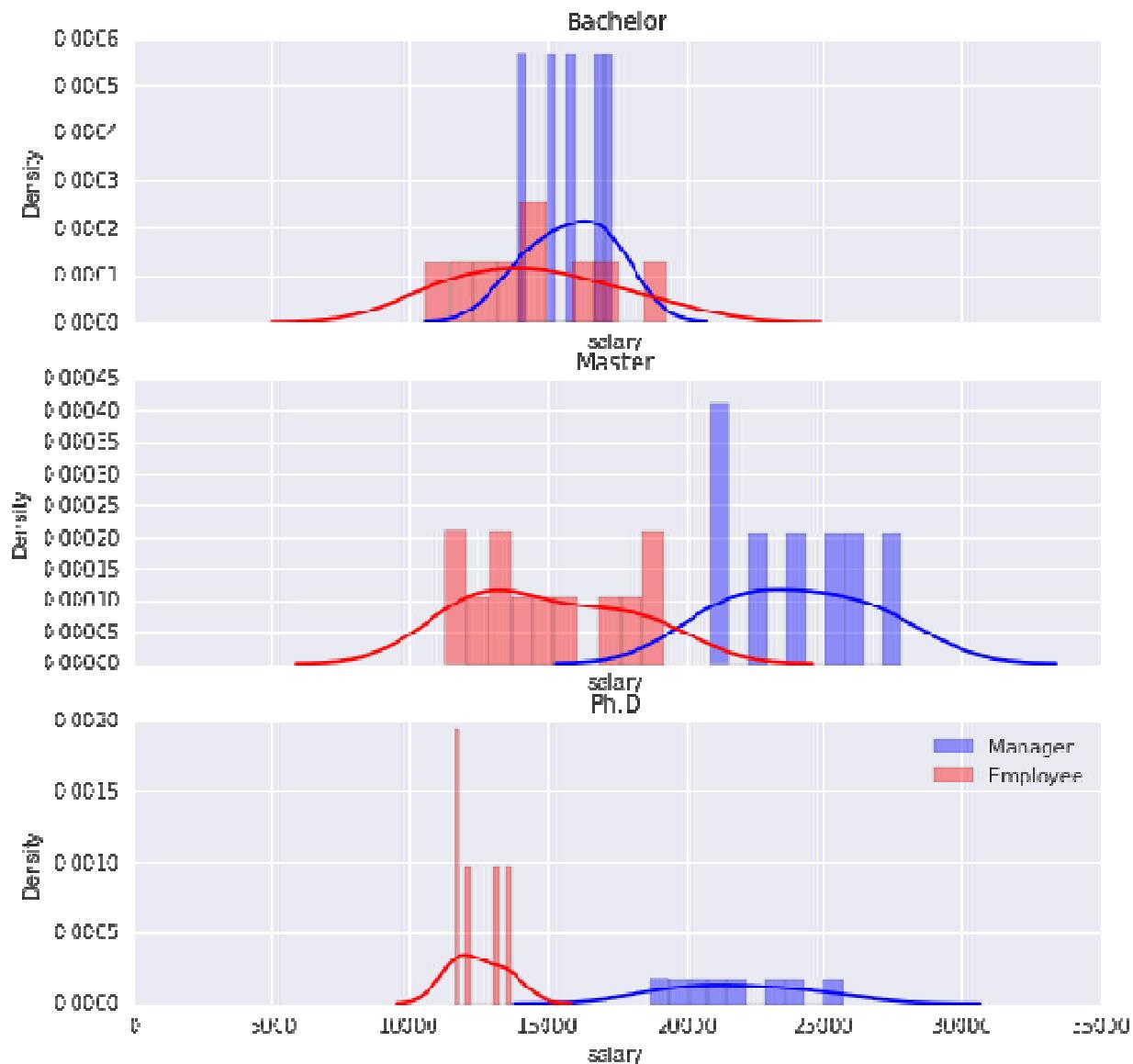
<matplotlib.axes._subplots.AxesSubplot at 0x7f3fd19d630>



Density plot with one figure containing multiple axis

One figure can contain several axis, whose contain the graphic elements

```
# Set up the matplotlib figure: 3 x 1 axis
f, axes = plt.subplots(3, 1, figsize=(9, 9), sharex=True)
i = 0
for edu, d in salary.groupby(['education']):
    sns.distplot(d.salary[d.management == "Y"], color="b", bins=10, label="Manager", ax=axes[i])
    sns.distplot(d.salary[d.management == "N"], color="r", bins=10, label="Employee", ax=axes[i])
    axes[i].set_title(edu)
    axes[i].set_ylabel('Density')
    i += 1
plt.legend()
<matplotlib.legend.Legend at 0x7f3fd9c70470>
```



```
g = sns.PairGrid(salary, hue="management")
```

```
g.map_diag(plt.hist)
```

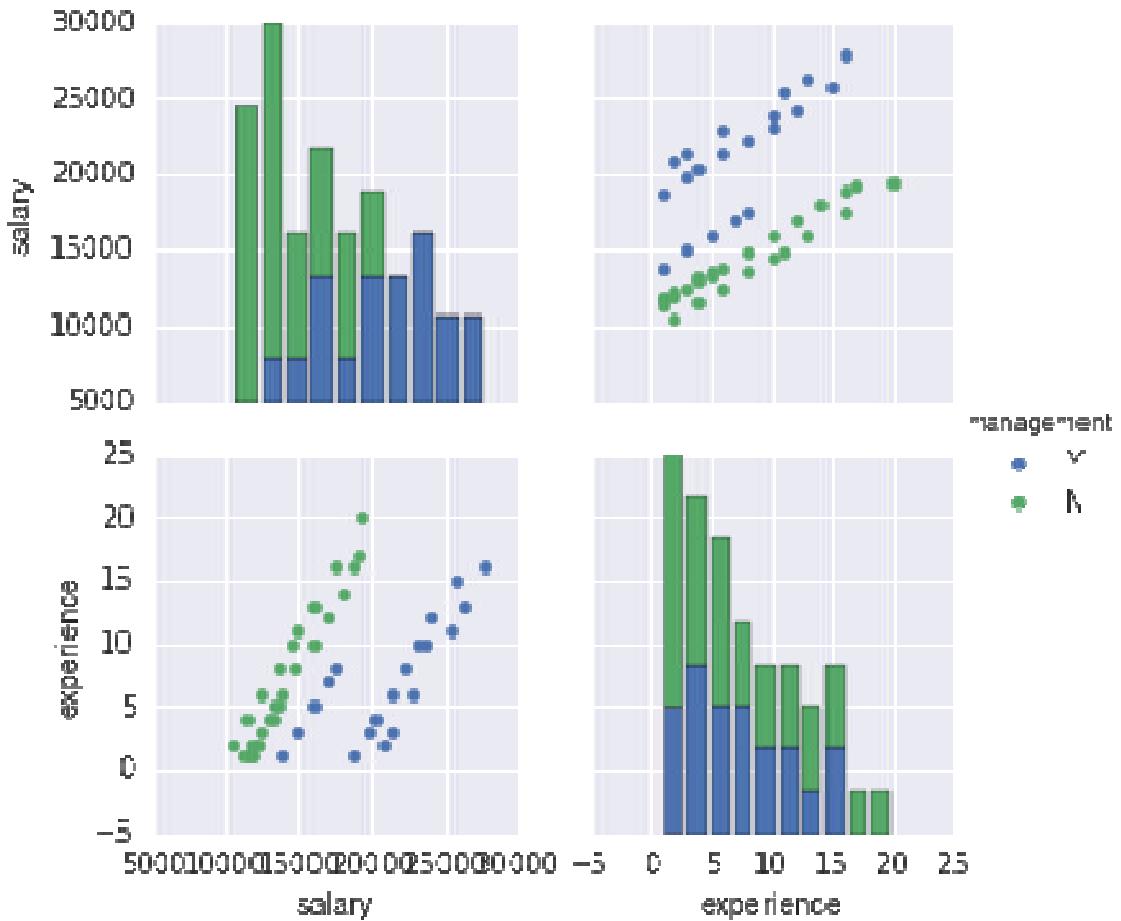
```
g.map_offdiag(plt.scatter)
```

```
g.add_legend()
```

```
/home/ed203246/anaconda3/lib/python3.7/site-packages/matplotlib/__init__.py:
```

```
892:UserWarning: axes.color_cycle is deprecated and replaced with axes.prop_cycle; please use the latter.  
warnings.warn(self.msg_depr % (key, alt_key))
```

```
<seaborn.axisgrid.PairGrid at 0x7f3fd195da0>
```



CHAPTER SIX

UNIVARIATE STATISTICS

Basics univariate statistics are required to explore dataset:

- Discover associations between a variable of interest and potential predictors. It is strongly recommended to start with simple univariate methods before moving to complex multivariate predictors.
- Assess the prediction performances of machine learning predictors.
- Most of the univariate statistics are based on the linear model which is one of the main model in machine learning.

Estimators of the main statistical measures

Mean

Properties of the expected value operator $E(\cdot)$ of a random variable X

$$E(X + c) = E(X) + c \quad (6.1)$$

$$E(X + Y) = E(X) + E(Y) \quad (6.2)$$

$$E(aX) = aE(X) \quad (6.3)$$

The estimator \bar{x} on a sample of size n : $x = x_1, \dots, x_n$ is given by

$$\bar{x} = \frac{1}{n} \sum_i x_i$$

\bar{x} is itself a random variable with properties:

- $E(\bar{x}) = \bar{x}$,
- $Var(\bar{x}) = \frac{Var(X)}{n}$.

Variance

$$Var(X) = E((X - E(X))^2) = E(X^2) - (E(X))^2$$

The estimator is

$$\sigma_x^2 = \frac{1}{n-1} \sum_i (x_i - \bar{x})^2$$

Note here the subtracted 1 degree of freedom (df) in the divisor. In standard statistical practice, $df = 1$ provides an unbiased estimator of the variance of a hypothetical infinite population. With $df = 0$ it instead provides a maximum likelihood estimate of the variance for normally distributed variables.

Standard deviation

$$Std(X) = \sqrt{Var(X)}$$

The estimator is simply $\sigma_x = \sqrt{\sigma_x^2}$.

Covariance

$$Cov(X, Y) = E((X - E(X))(Y - E(Y))) = E(XY) - E(X)E(Y).$$

Properties:

$$Cov(X, X) = Var(X)$$

$$Cov(X, Y) = Cov(Y, X)$$

$$\begin{aligned}\text{Cov}(cX, Y) &= c\text{Cov}(X, Y) \\ \text{Cov}(X + c, Y) &= \text{Cov}(X, Y)\end{aligned}$$

The estimator with $df = 1$ is

$$\sigma_{xy} = \frac{1}{n-1} \sum_i (x_i - \bar{x})(y_i - \bar{y})$$

Correlation

$$Cor(X, Y) = \frac{\text{Cov}(X, Y)}{\text{Std}(X)\text{Std}(Y)}$$

The estimator is

$$\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Standard Error (SE)

The standard error (SE) is the standard deviation (of the sampling distribution) of a statistic:

$$SE(X) = \frac{\text{Std}(X)}{\sqrt{n}}$$

Exercises

- Generate 2 random samples: $x \sim N(1.78, 0.1)$ and $y \sim N(1.66, 0.1)$, both of size 10.

Main distributions

Normal distribution

The normal distribution is useful because of the central limit theorem (CLT) which states that, given certain conditions, the arithmetic mean of a sufficiently large number of iterates of independent random variables, each with a well-defined expected value and well-defined variance, will be approximately normally distributed, regardless of the underlying distribution. Parameters: μ mean (location) and $\sigma^2 > 0$ variance. Estimators: \bar{x} and σ_x .

The Chi-Square distribution

- The squared standard $Z^2 \sim \chi^2_1$ (one df).
- The distribution of sum of squares of n normal random variables: $\sum_i^n Z_i^2 \sim \chi^2_n$

The sum of two χ^2 RV with p and q df is a χ^2 RV with $p + q$ df. This is useful when summing/subtracting sum of squares.

The Fisher's F-distribution

The F -distribution, $F_{n,p}$, with n and p degrees of freedom is the ratio of two independant χ^2 variables. Let $X \sim \chi^2_n$ and $Y \sim \chi^2_p$, then:

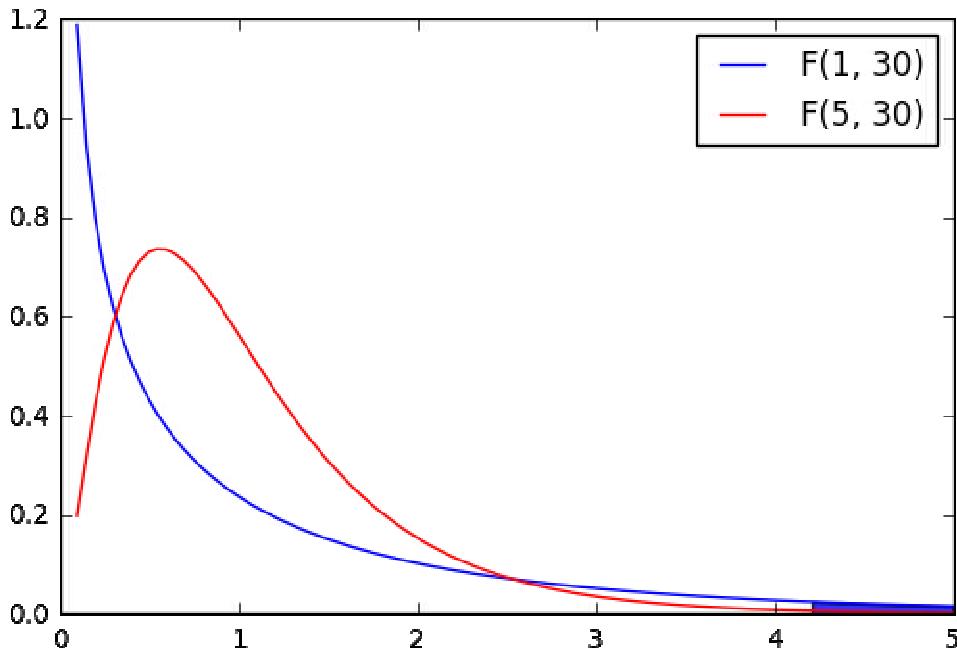
$$F_{n,p} = \frac{X/n}{Y/p}$$

The F -distribution plays a central role in hypothesis testing answering the question: Are two variances equals?

```

import numpy as np from scipy.stats
import f import matplotlib.pyplot as plt
%matplotlib inline
fvalues = np.linspace(.1, 5, 100)
# pdf(x, df1, df2): Probability density function at x of F.
plt.plot(fvalues, f.pdf(fvalues, 1, 30), 'b-', label="F(1, 30)")
plt.plot(fvalues, f.pdf(fvalues, 5, 30), 'r-', label="F(5, 30)")
plt.legend()
# cdf(x, df1, df2): Cumulative distribution function of F.
# ie. proba_at_f_inf_3 = f.cdf(3, 1, 30) # P(F(1,30) < 3)
# ppf(q, df1, df2): Percent point function (inverse of cdf) at q of F.
f_at_proba_inf_95 = f.ppf(.95, 1, 30) # q such P(F(1,30) < .95)
assert
f.cdf(f_at_proba_inf_95, 1, 30) == .95
# sf(x, df1, df2): Survival function (1 - cdf) at x of F.
proba_at_f_sup_3 = f.sf(3, 1, 30) # P(F(1,30) > 3)
assert
proba_at_f_inf_3 + proba_at_f_sup_3 == 1
# p-value: P(F(1, 30)) < 0.05
low_proba_fvalues = fvalues[fvalues > f_at_proba_inf_95]
plt.fill_between(low_proba_fvalues, 0, f.pdf(low_proba_fvalues, 1, 30), alpha=.8, label="P < 0.05")
plt.show()

```



The Student's *t*-distribution

Let $M \sim \mathcal{N}(0,1)$ and $V \sim \chi^2_n$. The *t*-distribution, T_n , with n degrees of freedom is the ratio:

$$T_n = \frac{M}{\sqrt{V/n}}$$

The distribution of the difference between an estimated parameter and its true (or assumed) value divided by the standard deviation of the estimated parameter (standard error) follow a *t*-distribution. Is this parameters different from a given value?

Testing pairwise associations

Mass univariate statistical analysis: explore association between pairs of variable.

- In statistics, a categorical variable or factor is a variable that can take on one of a limited, and usually fixed, number of possible values, thus assigning each individual to a particular group or “category”. The levels are the possible values of the variable. Number of levels =

2: binomial; Number of levels > 2: multinomial. There is no intrinsic ordering to the categories. For example, gender is a categorical variable having two categories (male and female) and there is no intrinsic ordering to the categories. Hair color is also a categorical variable having a number of categories (blonde, brown, brunette, red, etc.) and again, there is no agreed way to order these from highest to lowest. A purely categorical variable is one that simply allows you to assign categories but you cannot clearly order the variables. If the variable has a clear ordering, then that variable would be an ordinal variable, as described below.

- An ordinal variable is similar to a categorical variable. The difference between the two is that there is a clear ordering of the variables. For example, suppose you have a variable, economic status, with three categories (low, medium and high). In addition to being able to classify people into these three categories, you can order the categories as low, medium and high.
- A continuous or quantitative variable $x \in \mathbb{R}$ is one that can take any value in a range of possible values, possibly infinite. E.g.: Salary, Experience in years.

What statistical test should I use? See: http://www.ats.ucla.edu/stat/mult_pkg/whatstat/

Pearson correlation test (quantitative ~ quantitative)

Test the correlation coefficient of two quantitative variables. The test calculates a Pearson correlation coefficient and the p -value for testing non-correlation.

```
import numpy as np
import
scipy.stats as stats n = 50
x = np.random.normal(size=n)
y = 2 * x + np.random.normal(size=n)
# Compute with scipy
cor, pval = stats.pearsonr(x, y)
```

Although the parent population does not need to be normally distributed, the distribution of the population of sample means, x , is assumed to be normal. By the central limit theorem, if the sampling of the parent population is independent then the sample means will be approximately normal.

Exercise

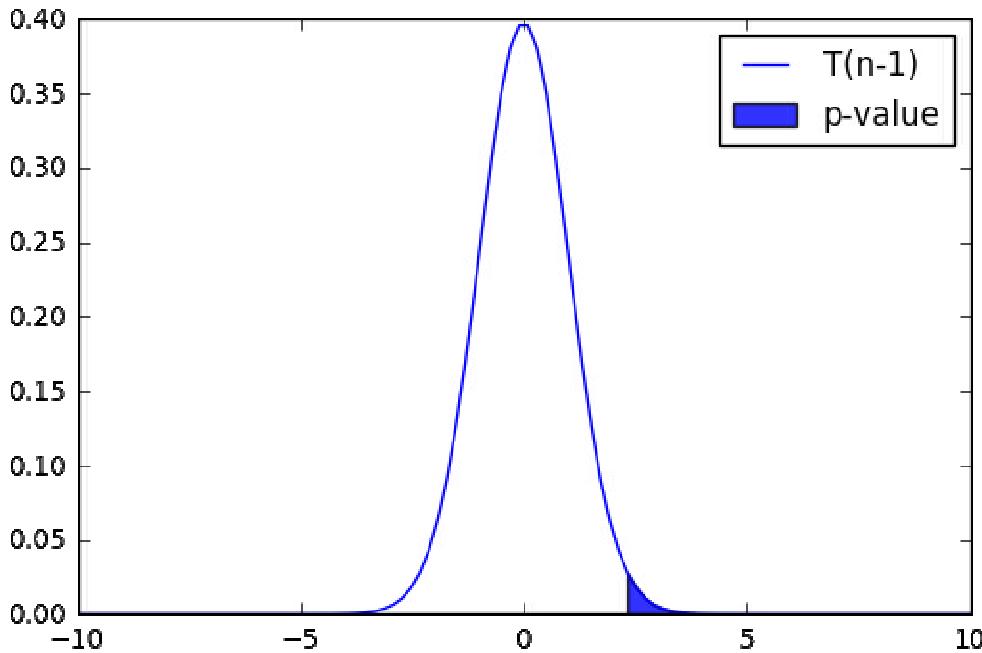
- Given the following samples, test whether its true mean is 1.75.

Warning, when computing the std or the variance, set ddof=1. The default value, ddof=0, leads to the biased estimator of the variance.

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
np.random.seed(seed=42) # make example reproducible
n = 100
x = np.random.normal(loc=1.78, scale=.1, size=n)
```

- Compute the t -value (tval)
- Plot the $T(n-1)$ distribution for 100 tvalues values within [0,10]. Draw $P(T(n-1)>tval)$ i.e. color the surface defined by x values larger than tval below the $T(n-1)$. Use the code.

```
# compute with scipy
tval, pval = stats.ttest_1samp(x, 1.75)
#tval = 2.1598800019529265 # assume the t-value
tvalues = np.linspace(-10, 10, 100)
plt.plot(tvalues, stats.t.pdf(tvalues, n-1), 'b-', label="T(n-1)")
upper_tval_tvalues = tvalues[tvalues > tval]
plt.fill_between(upper_tval_tvalues, 0, stats.t.pdf(upper_tval_tvalues, n-1), alpha=.8, label="p-value")
plt.legend()
```



- Compute the *p*-value: $P(T(n-1) > tval)$.
- The *p*-value is one-sided: a two-sided test would test $P(T(n-1) > tval)$ and $P(T(n-1) < -tval)$. What would the two-sided *p*-value be?
- Compare the two-sided *p*-value with the one obtained by `stats.ttest_1samp` using `assert /np.allclose(arr1,arr2)`.

Equal or unequal sample sizes, equal variance

This test is used only when it can be assumed that the two distributions have the same variance. (When this assumption is violated, see below.) The *t* statistic, that is used to test whether the means are different, can be calculated as follows:

$$t = \frac{\bar{x} - \bar{y}}{\sigma \cdot \sqrt{\frac{1}{n_x} + \frac{1}{n_y}}} \text{ where}$$

$$\sigma = \sqrt{\frac{\sigma_x^2(n_x - 1) + \sigma_y^2(n_y - 1)}{n_x + n_y - 2}}$$

is an estimator of the common standard deviation of the two samples: it is defined in this way so that its square is an unbiased estimator of the common variance whether or not the population means are the same.

Equal or unequal sample sizes, unequal variances (Welch's *t*-test)

Welch's *t*-test defines the *t* statistic as

$$t = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{\sigma_x^2}{n_x} + \frac{\sigma_y^2}{n_y}}}$$

To compute the p -value one needs the degrees of freedom associated with this variance estimate. It is approximated using the Welch–Satterthwaite equation:

$$\nu \approx \frac{\left(\frac{\sigma_x^2}{n_x} + \frac{\sigma_y^2}{n_y} \right)^2}{\frac{\sigma_x^4}{n_x^2(n_x-1)} + \frac{\sigma_y^4}{n_y^2(n_y-1)}}.$$

Exercise

Given the following two samples, test whether their means are equal using the standard t-test, assuming equal variance.

```
import scipy.stats as stats
nx, ny = 50, 25
x = np.random.normal(loc=1.76, scale=0.1, size=nx)
y = np.random.normal(loc=1.70, scale=0.12, size=ny)
# Compute with scipy
tval, pval = stats.ttest_ind(x, y, equal_var=True)
```

- Compute the t -value.
- Compute the p -value.
- The p -value is one-sided: a two-sided test would test $P(T > tval)$ and $P(T < -tval)$. What would the two sided p -value be?
- Compare the two-sided p -value with the one obtained by `stats.ttest_ind` using `assert np.allclose(arr1,arr2)`.

ANOVA F-test (quantitative ~ categorial (>2 levels))

Analysis of variance (ANOVA) provides a statistical test of whether or not the means of several groups are equal, and therefore generalizes the t -test to more than two groups. ANOVAs are useful for comparing (testing) three or more means (groups or variables) for statistical significance. It is conceptually similar to multiple two-sample t -tests, but is less conservative. Here we will consider one-way ANOVA with one independent variable.

1. Model the data

A company has applied three marketing strategies to three samples of customers in order increase their business volume. The marketing is asking whether the strategies led to different increases of business volume. Let y_1, y_2 and y_3 be the three samples of business volume increase.

Here we assume that the three populations were sampled from three random variables that are normally distributed. I.e., $Y_1 \sim N(\mu_1, \sigma_1)$, $Y_2 \sim N(\mu_2, \sigma_2)$ and $Y_3 \sim N(\mu_3, \sigma_3)$.

2. Fit: estimate the model parameters

Estimate means and variances: $\bar{y}_i, \sigma_i, \forall i \in \{1,2,3\}$.

3. F-test

Source: <https://en.wikipedia.org/wiki/F-test>

The ANOVA F-test can be used to assess whether any of the strategies is on average superior, or inferior, to the others versus the null hypothesis that all four strategies yield the same mean response (increase of business volume). This is an example of an “omnibus” test, meaning that a single test is performed to detect any of several possible differences. Alternatively, we could carry out pair-wise tests among the strategies. The advantage of the ANOVA F-test is that we do not need to pre-specify which strategies are to be compared, and we do not need to adjust for making multiple comparisons. The disadvantage of the ANOVA F-test is that if we reject the null hypothesis, we do not know which strategies can be said to be significantly different from the others. The formula for the one-way ANOVA F-test statistic is

$$F = \frac{\text{explained variance}}{\text{unexplained variance}}$$

or

$$F = \frac{\text{between-group variability}}{\text{within-group variability}}$$

The “explained variance”, or “between-group variability” is

$$\sum_i n_i (Y_{-i} - Y_{\bar{i}})^2 / (K - 1),$$

where Y_{-i} denotes the sample mean in the i th group, n_i is the number of observations in the i th group, $Y_{\bar{i}}$ denotes the overall mean of the data, and K denotes the number of groups.

The “unexplained variance”, or “within-group variability” is

$$\sum_{ij} (Y_{ij} - Y_{-i})^2 / (N - K)$$

where Y_{ij} is the j th observation in the i th out of K groups and N is the overall sample size. This F -statistic follows the F -distribution with $K - 1$ and $N - K$ degrees of freedom under the null hypothesis. The statistic will be large if the between-group variability is large relative to the within-group variability, which is unlikely to happen if the population means of the groups all have the same value.

Note that when there are only two groups for the one-way ANOVA F-test, $F = t^2$ where t is the Student's t statistic.

Exercise

Perform an ANOVA on the following dataset

- Compute between and within variances
- Compute F -value: fval
- Compare the p -value with the one obtained by stats.f_oneway using assert np.allclose(arr1,arr2)

```
# dataset
mu_k = np.array([1, 2, 3]) # means of 3 samples
sd_k = np.array([1, 1, 1]) # sd of 3 samples
n_k = np.array([10, 20, 30]) # sizes of 3 samples
grp = [0, 1, 2] # group labels
n = np.sum(n_k)
label = np.hstack([[k] * n_k[k] for k in range(3)])
for k in [0, 1, 2]):
    y = np.zeros(n)
    for k in grp:
        y[label == k] = np.random.normal(mu_k[k], sd_k[k], n_k[k])
# Compute with scipy
fval, pval = stats.f_oneway(y[label == 0], y[label == 1], y[label == 2])
```

Chi-square, χ^2 (categorical ~ categorical)

Computes the chi-square, χ^2 , statistic and p -value for the hypothesis test of independence of frequencies in the observed contingency table. The observed frequencies are tested against an expected contingency table obtained by computing expected frequencies based on the marginal sums under the assumption of independence.

```

import numpy as np
import pandas as pd
import scipy.stats as stats
# Dataset:
# 15 samples:
# 10 first with canalar tumor, 5 last without
canalar_tumor = np.array([1] * 10 + [0] * 5)
# 8 first with metastasis, 6 without, the last with.
meta = np.array([1] * 8 + [0] * 6 + [1])
crosstab = pd.crosstab(canalar_tumor, meta, rownames=['canalar_tumor'], colnames=['meta'])
print("Observed table:")
print("-----")
print(crosstab)
chi2, pval, dof, expected = stats.chi2_contingency(crosstab)

print("Statistics:")
print("-----")
print("Chi2 = %f, pval = %f" % (chi2, pval))
print("Expected table:")
print("-----")
print(expected)

```

Observed table: -----

	0	1
meta		
canalar_tumor	1	4 1
	2	2 8

Statistics:

Chi2 = 2.812500, pval = 0.093533

Expected table:

[2. 3.]
[4. 6.]

Computing expected cross-table

```

# Compute expected cross-table based on proportion
meta_marg = crosstab.sum(axis=0)
meta_freq = meta_marg / meta_marg.sum()
canalar_tumor_marg = crosstab.sum(axis=1)
canalar_tumor_freq = canalar_tumor_marg / canalar_tumor_marg.sum()
print('Canalar tumor frequency? Yes: %.2f % canalar_tumor_freq[0], No: %.2f % canalar_tumor_freq[1]')
print('Metastasis frequency? Yes: %.2f % meta_freq[0], No: %.2f % meta_freq[1]')
print('Expected frequencies:')
print(np.outer(canalar_tumor_freq, meta_freq))
print('Expected cross-table')
print(np.outer(canalar_tumor_freq, meta_freq) * len(canalar_tumor))

```

Canalar tumor frequency? Yes: 0.33 No: 0.67

Metastasis frequency? Yes: 0.40 No: 0.60 Expected frequencies:

[0.13333333 0.2]
[0.26666667 0.4]

Expected cross-table:

[2. 3.]
[4. 6.]

Exercise

Write a function `univar_stat(df,target,variables)` that computes the parametric statistics and p -values between the target variable (provided as string) and all variables (provided as a list of string) of the pandas DataFrame `df`. The target is a quantitative variable but variables may be quantitative or qualitative. The function returns a DataFrame with four columns: `variable`, `test`, `value`, `p_value`.

Apply it to the salary dataset available at

https://raw.github.com/neurospin/pystatsml/master/data/salary_table.csv, with target being S: salaries for IT staff in a corporation.

Non-parametric test of pairwise associations

Spearman rank-order correlation (quantitative ~ quantitative)

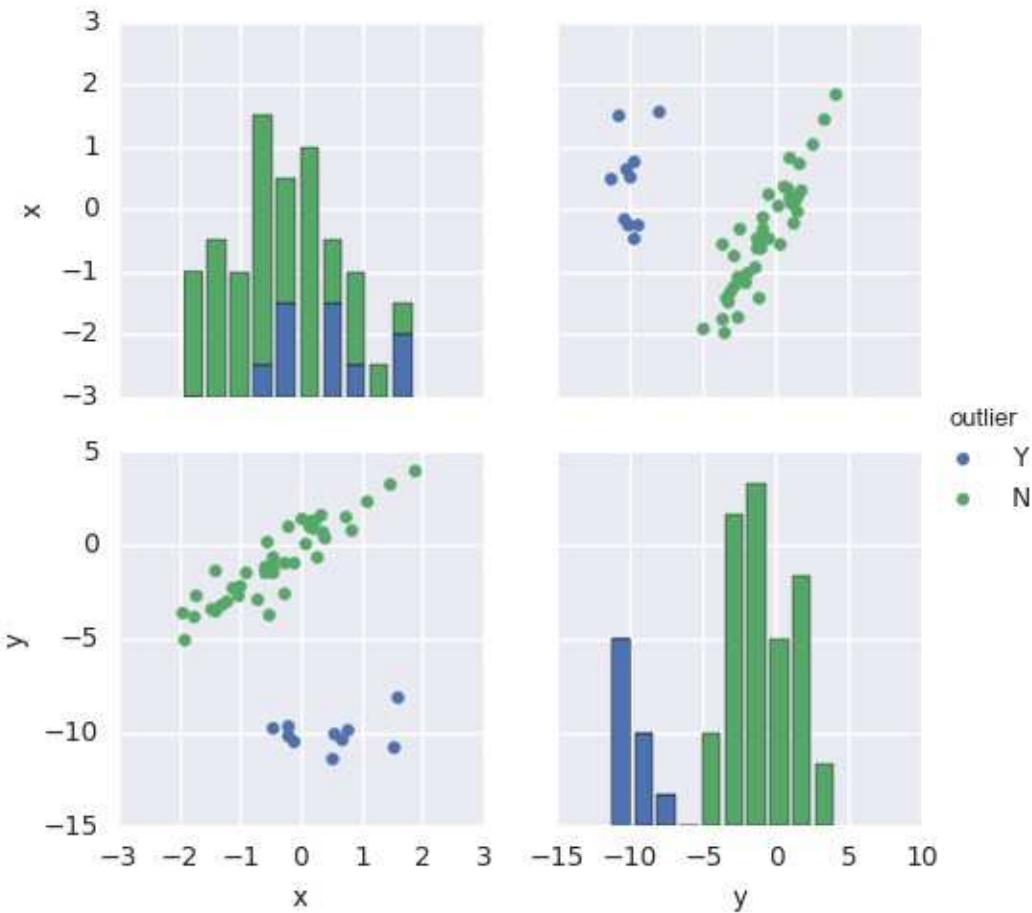
The Spearman correlation is a non-parametric measure of the monotonicity of the relationship between two datasets. Unlike the Pearson correlation, the Spearman correlation does not assume that both datasets are normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact monotonic relationship. Positive correlations imply that as x increases, so does y . Negative correlations imply that as x increases, y decreases.

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
np.random.seed(seed=42) # make example reproducible
n = 50
noutliers = 10 x = np.random.normal(size=n) y = 2 * x + np.random.normal(size=n)
y[:noutliers] = np.random.normal(loc=-10, size=noutliers) # Add 40 outliers
outlier = np.array(["N"] * n)
outlier[:noutliers] = "Y"
# Compute with scipy
cor, pval = stats.spearmanr(x, y)
print("Non-Parametric Spearman cor test, cor: %.4f, pval: %.4f" % (cor, pval))
# Plot distribution + pairwise scatter plot df =
pd.DataFrame(dict(x=x, y=y, outlier=outlier))
g = sns.PairGrid(df, hue="outlier")
g.map_diag(plt.hist)
g.map_offdiag(plt.scatter)
g = g.add_legend()
# Compute the parametric Pearsonw cor test
cor, pval = stats.pearsonr(x, y)
print("Parametric Pearson cor test: cor: %.4f, pval: %.4f" % (cor, pval))
```

Non-Parametric Spearman cor test, cor: 0.2996, pval: 0.0345

```
/home/ed203246/anaconda3/lib/python3.5/site-packages/matplotlib/__init__.py:898:userWarning:
axes.color_cycle is deprecated and replaced with axes.prop_cycle; please use the latter.
warnings.warn(self.msg_depr % (key, alt_key))
```

Parametric Pearson cor test: cor: 0.0426, pval: 0.7687



Wilcoxon signed-rank test (quantitative ~ cte)

Source: https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ (i.e. it is a paired difference test). It is equivalent to one-sample test of the difference of paired samples.

It can be used as an alternative to the paired Student's t -test, t -test for matched pairs, or the t -test for dependent samples when the population cannot be assumed to be normally distributed.

When to use it? Observe the data distribution: - presence of outliers - the distribution of the residuals is not Gaussian

It has a lower sensitivity compared to t -test. May be problematic to use when the sample size is small.

Null hypothesis H_0 : difference between the pairs follows a symmetric distribution around zero.

```
import scipy.stats as stats
n = 20
# Business Volume time 0
bv0 = np.random.normal(loc=3, scale=.1, size=n)
# Business Volume time 1
bv1 = bv0 + 0.1 + np.random.normal(loc=0, scale=.1, size=n)
# create an outlier bv1[0] -= 10
```

Non-parametric test of pairwise associations

```
# Paired t-test print(stats.ttest_rel(bv0, bv1))
```

```
# Wilcoxon
print(stats.wilcoxon(bv0, bv1))
Ttest_relResult(statistic=0.77854266212061596, pvalue=0.44583743298063039)
WilcoxonResult(statistic=26.0, pvalue=0.0031851432686599455)
```

Mann–Whitney U test (quantitative ~ categorial (2 levels))

In statistics, the Mann–Whitney U test (also called the Mann–Whitney–Wilcoxon, Wilcoxon rank-sum test or Wilcoxon–Mann–Whitney test) is a nonparametric test of the null hypothesis that two samples come from the same population against an alternative hypothesis, especially that a particular population tends to have larger values than the other.

It can be applied on unknown distributions contrary to e.g. a t -test that has to be applied only on normal distributions, and it is nearly as efficient as the t -test on normal distributions.

```
import scipy.stats as stats
n = 20
# Buismess Volume group 0
bv0 = np.random.normal(loc=1, scale=.1, size=n)
# Buismess Volume group 1
bv1 = np.random.normal(loc=1.2, scale=.1, size=n)
# create an outlier
bv1[0] -= 10
# Two-samples t-test
print(stats.ttest_ind(bv0, bv1))
# Wilcoxon
print(stats.mannwhitneyu(bv0, bv1))
Ttest_indResult(statistic=0.62748520384004158, pvalue=0.53409388734462837)
MannwhitneyuResult(statistic=43.0, pvalue=2.3024709881112628e-05)
```

Linear model

For the regression case, the statistical model is as follows. Given a (random) sample $(y_i, x_i^1, \dots, x_i^p)$, $i = 1, \dots, n$ the relation between the observations y_i and the independent variables x_i^p is formulated as

$$y_i = \beta_0 + \beta_1 x_i^1 + \dots + \beta_p x_i^p + \varepsilon_i \quad i = 1, \dots, n$$

- An independent variable (IV) is exactly what it sounds like. It is a variable that stands alone and is not changed by the other variables you are trying to measure. For example, someone's age might be an independent variable. Other factors (such as what they eat, how much they go to school, how much television they watch) aren't going to change a person's age. In fact, when you are looking for some kind of relationship between variables you are trying to see if the independent variable causes some kind of change in the other variables, or dependent variables. In Machine Learning, these variables are also called the predictors.
- A dependent variable is exactly what it sounds like. It is something that depends on other factors. For example, a test score could be a dependent variable because it could change depending on several factors such as how much you studied, how much sleep you got the night before you took the test, or even how hungry you were when you took it. Usually when you are looking for a relationship between two things you are trying to find out what makes the dependent variable change the way it does. In Machine Learning this variable is called a target variable.

Simple linear regression (one continuous independant variable (IV))

Using the dataset "salary", explore the association between the dependant variable (e.g. Salary) and the independant variable (e.g.: Experience is quantitative).

```
import pandas as pd
```

```

import matplotlib.pyplot as plt %matplotlib inline
url= 'https://raw.github.com/neurospin/pystatsml/master/data/salary_table.csv'
salary = pd.read_csv(url)

```

1. Model the data

Model the data on some hypothesis e.g.: salary is a linear function of the experience.

$$\text{salary}_i = \beta \text{experience}_i + \beta_0 + \epsilon_i,$$

more generally

$$y_i = \beta x_i + \beta_0 + \epsilon_i$$

- β : the slope or coefficient or parameter of the model, • β_0 : the intercept or bias is the second parameter of the model,
- ϵ_i is the i th error, or residual with.

This model is similar to a correlation. $\epsilon \sim \mathcal{N}(0, \sigma^2)$

2. Fit: estimate the model parameters

The goal it so estimate β , β_0 and σ^2 .

Minimizes the mean squared error (MSE) or the Sum squared error (SSE). The so-called

$$SSE = \sum_i \epsilon_i^2$$

$$SSE = \sum_i (y_i - \beta x_i - \beta_0)^2$$

Ordinary Least Squares (OLS) finds β, β_0 that minimizes the

Recall from calculus that an extreme point can be found by computing where the derivative is zero, i.e. to find the intercelt, we perform the steps:

$$\frac{\partial SSE}{\partial \beta_0} = \sum_i (y_i - \beta x_i - \beta_0) = 0$$

$$\sum_i y_i = \beta \sum_i x_i + n \beta_0$$

$$n \bar{y} = n \beta \bar{x} + n \beta_0$$

$$\beta_0 = \bar{y} - \beta \bar{x}$$

To find the regression coefficient, we perform the steps:

$$\frac{\partial SSE}{\partial \beta} = \sum_i x_i (y_i - \beta x_i - \beta_0) = 0$$

Plug in β_0 :

$$\sum_i x_i (y_i - \beta x_i - \bar{y} + \beta \bar{x}) = 0$$

$$\sum_i x_i y_i - \bar{y} \sum_i x_i = \beta \sum_i (x_i - \bar{x})$$

Divide both sides by n :

$$\frac{1}{n} \sum_i x_i y_i - \bar{y} \bar{x} = \frac{1}{n} \beta \sum_i (x_i - \bar{x})$$

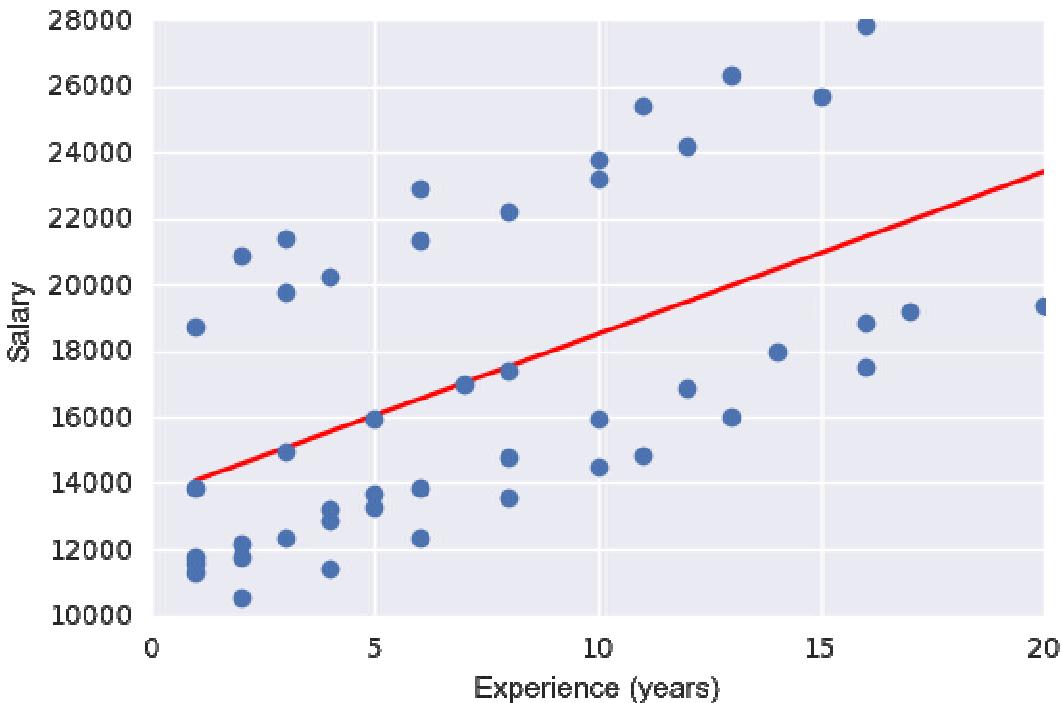
$$\beta = \frac{\frac{1}{n} \sum_i x_i y_i - \bar{y} \bar{x}}{\frac{1}{n} \sum_i (x_i - \bar{x})} = \frac{Cov(x, y)}{Var(x)}$$

```

from scipy import stats
import numpy as np
y, x = salary.salary, salary.experience
beta, beta0, r_value, p_value, std_err =
stats.linregress(x,y)
print("y = %f x + %f, r: %f, r-squared: %f, p-value: %f, std_err: %f" % (beta,
beta0, r_value, r_value**2, p_value, std_err))
# plotting the line
yhat = beta * x + beta0 # regression line
plt.plot(x, yhat, 'r-', x, y, 'o')
plt.xlabel('Experience (years)')
plt.ylabel('Salary')
plt.show()

```

y = 491.486913 x + 13584.043803, r: 0.538886, r-squared: 0.290398, p-value: 0.000112, std_err: 115.823381



3. F-Test

3.1 Goodness of fit

The goodness of fit of a statistical model describes how well it fits a set of observations. Measures of goodness of fit typically summarize the discrepancy between observed values and the values expected under the model in question. We will consider the explained variance also known as the coefficient of determination, denoted R^2 pronounced R-squared.

The total sum of squares, SS_{tot} is the sum of the sum of squares explained by the regression, SS_{reg} , plus the sum of squares of residuals unexplained by the regression, SS_{res} , also called the SSE, i.e. such that

$$SS_{\text{tot}} = SS_{\text{reg}} + SS_{\text{res}}$$

The mean of y is

$$\bar{y} = \frac{1}{n} \sum_i y_i$$

The total sum of squares is the total squared sum of deviations from the mean of y , i.e.

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$$

The regression sum of squares, also called the explained sum of squares:

$$SS_{\text{reg}} = \sum_i (\hat{y}_i - \bar{y})^2$$

where $\hat{y}_i = \beta x_i + \beta_0$ is the estimated value of salary \hat{y}_i given a value of experience x_i .

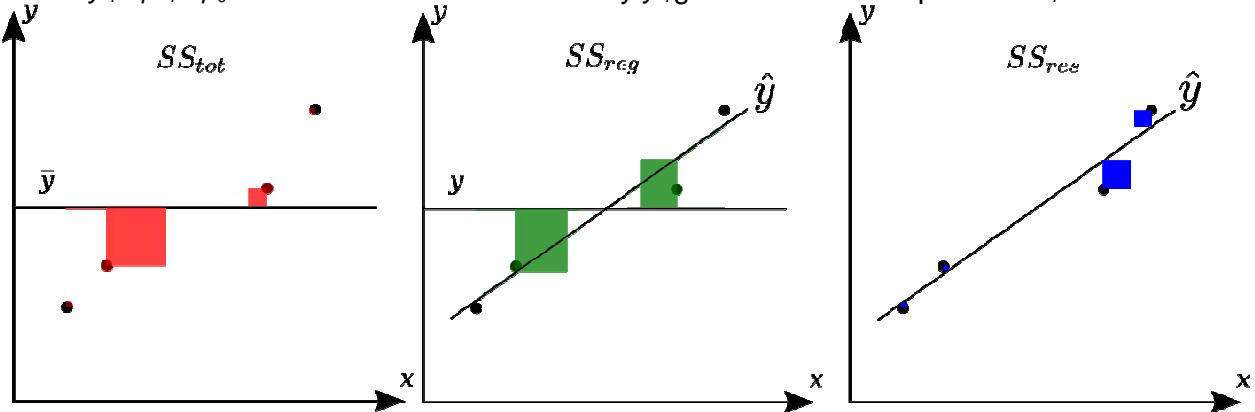


Fig. 6.2

Test

Let $\sigma^2 = SS_{res}/(n - 2)$ be an estimator of the variance of ϵ . The 2 stem from the number of estimated parameters, intercept and coefficient.

- Unexplained variance: $\frac{SS_{res}}{\sigma^2} \sim \chi^2_{n-2}$
- Explained variance: $\frac{SS_{reg}}{\sigma^2} \sim \chi^2_1$. The single degree of freedom comes from the difference between $\frac{SS_{tot}}{\sigma^2} (\sim \chi^2_{n-1})$ and $\frac{SS_{res}}{\sigma^2} (\sim \chi^2_{n-2})$, i.e. $(n - 1) - (n - 2)$ degree of freedom.

The Fisher statistics of the ratio of two variances:

$$F = \frac{\text{Explained variance}}{\text{Unexplained variance}} = \frac{\frac{SS_{reg}}{\sigma^2}/1}{\frac{SS_{res}}{\sigma^2}/(n-2)} \stackrel{D}{=} F(1, n-2)$$

Using the F -distribution, compute the probability of observing a value greater than F under H_0 , i.e.: $P(x > F|H_0)$, i.e. the survival function (1 – Cumulative Distribution Function) at x of the given F -distribution.

Exercise

Compute:

- y_{μ} : `y_mu`
- SS_{tot} : `ss_tot`
- SS_{reg} : `ss_reg`
- SS_{res} : `ss_res`
- Check partition of variance formula based on sum of squares by using assert `np.allclose(val1, val2, atol=1e-05)`
- Compute R^2 and compare it with the `r_value` above
- Compute the F score
- Compute the p -value:
- Plot the $F(1, n)$ distribution for 100 f values within [10, 25]. Draw $P(F(1, n) > F)$, i.e. color the surface defined by the x values larger than F below the $F(1, n)$.
- $P(F(1, n) > F)$ is the p -value, compute it.

Multiple regression

Theory

Multiple Linear Regression is the most basic supervised learning algorithm.

Given: a set of training data $\{x_1, \dots, x_N\}$ with corresponding targets $\{y_1, \dots, y_N\}$.

In linear regression, we assume that the model that generates the data involves only a linear combination of the input variables, i.e.

$$y(x_i, \beta) = \beta^0 + \beta^1 x_i^1 + \dots + \beta^P x_i^P,$$

or, simplified

$$y(x_i, \beta) = \beta_0 + \sum_{j=1}^{P-1} \beta_j x_i^j.$$

Extending each sample with an intercept, $x_i := [1, x_i] \in R^{P+1}$ allows us to use a more general notation based on linear algebra and write it as a simple dot product:

$$y(x_i, \beta) = x_i^T \beta,$$

where $\beta \in R^{P+1}$ is a vector of weights that define the $P+1$ parameters of the model. From now we have P regressors + the intercept.

Minimize the Mean Squared Error MSE loss:

$$MSE(\beta) = \frac{1}{N} \sum_{i=1}^N (y_i - y(x_i, \beta))^2 = \frac{1}{N} \sum_{i=1}^N (y_i - x_i^T \beta)^2$$

Let $X = [x_0^T, \dots, x_N^T]$ be a $N \times P+1$ matrix of N samples of P input features with one column of one and let be $y = [y_1, \dots, y_N]$ be a vector of the N targets. Then, using linear algebra, the mean squared error (MSE) loss can be rewritten:

$$MSE(\beta) = \frac{1}{N} \|y - X\beta\|_2^2.$$

The β that minimises by:

$$\nabla_\beta \left(\frac{1}{N} \|y - X\beta\|_2^2 \right) = 0 \quad \text{the MSE can be found}$$

$$\frac{1}{N} \nabla_\beta (y - X\beta)^T (y - X\beta) = 0 \quad (6.4)$$

$$\frac{1}{N} \nabla_\beta (y^T y - 2\beta^T X^T y + \beta^T X^T X\beta) = 0 \quad (6.6)$$

$$-2X^T y + 2X^T X\beta = 0 \quad (6.7) \quad (6.8)$$

$$X^T X\beta = X^T y \quad (6.9)$$

$$\beta = (X^T X)^{-1} X^T y,$$

where $(X^T X)^{-1} X^T$ is a pseudo inverse of X .

Fit with numpy

```
import numpy as np
import scipy
np.random.seed(seed=42) # make the example reproducible
# Dataset
N, P = 50, 4
X = np.random.normal(size=N * P).reshape((N, P))
## Our model needs an intercept so we add a column of 1s:
X[:, 0] = 1
print(X[:5, :])
betastar = np.array([10, 1., .5, 0.1])
e = np.random.normal(size=N)
y = np.dot(X, betastar) + e
# Estimate the parameters
Xpinv = scipy.linalg.pinv2(X)
betahat = np.dot(Xpinv, y)
print("Estimated beta:\n", betahat)
```

```
[[ 1.       -0.13826430  0.64768854  1.52302986]
 [ 1.      -0.23413696  1.57921282  0.76743473]]
```

```
[1. 0.54256004 -0.46341769 -0.46572975]
[1. -1.91328024 -1.72491783 -0.56228753]
[1. 0.31424733 -0.90802408 -1.4123037 ]]
Estimated beta:
[10.14742501 0.57938106 0.51654653 0.17862194]
```

Exercise

1. What are the dimensions of $\text{pinv}(X)$?
2. Compute the MSE between the predicted values and the true values.

Linear model with statsmodels

Sources: <http://statsmodels.sourceforge.net/devel/examples/>

Multiple regression

Interface with Numpy

```
import statsmodels.api as sm
## Fit and summary:
model = sm.OLS(y, X).fit()
print(model.summary())
# prediction of new values
y_pred = model.predict(X)
# residuals + prediction == true values
assert
np.all(y_pred + model.resid == y)
```

OLS Regression Results												
Dep. Variable:	y	R-squared:	0.363									
Model:	OLS	Adj. R-squared:	0.322									
Method:	Least Squares	F-statistic:	8.748									
Date:	Fri, 06 Jan 2017	Prob (F-statistic):	0.000106									
Time:	10:42:31	Log-Likelihood:	-71.271									
No. Observations:	50	AIC:	150.5									
Df Residuals:	46	BIC:	158.2									
Df Model:	3											
Covariance Type:	nonrobust											
		coef	std									
err	t	P> t	[95.0% Conf. Int.]									
				--const	10.1474	0.150	67.520	0.000	9.845			
10.450	x1	0.5794	0.160	3.623	0.001	0.258	0.901	x2	0.5165	0.151	3.425	0.001
0.213	0.820	x3	0.1786	0.144	1.240	0.221	-0.111	0.469				
Omnibus:		2.493		Durbin-Watson:			2.369					
Prob(Omnibus):		0.288		Jarque-Bera (JB):			1.544					
Skew:		0.330		Prob(JB):			0.462					
Kurtosis:		3.554		Cond. No.			1.27					
Warnings:												
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.												

Interface with Pandas

Use R language syntax for data.frame. For an additive model: $y \sim x_1 + x_2$.

```
import statsmodels.formula.api as smfrmla
df = pd.DataFrame(np.column_stack([X, y]), columns=['inter', 'x1','x2', 'x3', 'y'])
# Build a model excluding the intercept, it is implicit
model = smfrmla.ols("y ~ x1 + x2 + x3", df).fit()
print(model.summary())
```

OLS Regression Results											
<hr/>											
Dep. Variable:	y	R-squared:	0.363								
Model:	OLS	Adj. R-squared:	0.322								
Method:	Least Squares	F-statistic:	8.748								
Date:	Fri, 06 Jan 2017	Prob (F-statistic):	0.000106								
Time:	10:42:43	Log-Likelihood:	-71.271								
No. Observations:	50	AIC:	150.5								
Df Residuals:	46	BIC:	158.2								
Df Model:	3										
Covariance Type:	nonrobust										
<hr/>											
coef	std err	t	P> t	[95.0% Conf. Int.]							
Intercept	10.1474	0.150	67.520	0.000	9.845	10.450	x1	0.5794	0.160	3.623	0.001
	0.258	0.901	x2	0.5165	0.151	3.425	0.001	0.213	0.820	x3	0.1786
	0.221	-0.111		0.469					0.144		1.240
<hr/>											
Omnibus:		2.493		Durbin-Watson:						2.369	
Prob(Omnibus):		0.288		Jarque-Bera (JB):						1.544	
Skew:		0.330		Prob(JB):						0.462	
Kurtosis:		3.554		Cond. No.						1.27	
<hr/>											
Warnings:											
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.											

Multiple regression with categorical independent variables or factors: Analysis of covariance (ANCOVA)

Analysis of covariance (ANCOVA) is a linear model that blends ANOVA and linear regression. ANCOVA evaluates whether population means of a dependent variable (DV) are equal across levels of a categorical independent variable (IV) often called a treatment, while statistically controlling for the effects of other quantitative or continuous variables that are not of primary interest, known as covariates (CV).

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
try:
    salary = pd.read_csv("./data/salary_table.csv")
except:
    url = 'https://raw.github.com/neurospin/pystatsml/master/data/salary_table.csv'
    salary = pd.read_csv(url)
```

One-way AN(C)OVA

- ANOVA: one categorical independent variable, i.e. one factor.
- ANCOVA: ANOVA with some covariates.

```
import statsmodels.formula.api as smfrmla
oneway = smfrmla.ols('salary ~ management + experience', salary).fit()
print(oneway.summary())
aov = sm.stats.anova_lm(oneway, typ=2) # Type 2 ANOVA DataFrame
print(aov)
```

OLS Regression Results				
Dep. Variable:	salary	R-squared:	0.865	
Model:	OLS	Adj. R-squared:	0.859	
Method:	Least Squares	F-statistic:	138.2	
Date:	Fri, 06 Jan 2017	Prob (F-statistic):	1.90e-19	
Time:	10:45:13	Log-Likelihood:	-407.76	
No. Observations:	46	AIC:	821.5	
Df Residuals:	43	BIC:	827.0	
Df Model:	2			
Covariance Type:	nonrobust			
coef	std err	t	P> t	[95.0% Conf. Int.]
Intercept	1.021e+04	525.999	19.411	0.000 9149.578 1.13e+04 management[T.Y] 7145.0151 527.320 13.550
	0.000	6081.572	8208.458	experience 527.1081 51.106 10.314 0.000 424.042 630.174
Omnibus:	11.437			Durbin-Watson: 2.193
Prob(Omnibus):	0.003			Jarque-Bera (JB): 11.260
Skew:	-1.131			Prob(JB): 0.00359
Kurtosis:	3.872			Cond. No. 22.4
Warnings:				
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.				
sum_sq	df	F	PR(>F)	
management	5.755739e+08	1.0	183.593466	4.054116e-17 experience
3.334992e+08	1.0	106.377768	3.349662e-13	
Residual	1.348070e+08	43.0	NaN	NaN

Two-way AN(C)OVA

Ancova with two categorical independent variables, i.e. two factors.

```
import statsmodels.formula.api as smfrmrla
twoway = smfrmrla.ols('salary ~ education + management + experience', salary).fit()
print(twoway.summary())
aov = sm.stats.anova_lm(twoway, typ=2) # Type 2 ANOVA DataFrame
print(aov)
```

OLS Regression Results				
Dep. Variable:	salary	R-squared:	0.957	
Model:	OLS	Adj. R-squared:	0.953	
Method:	Least Squares	F-statistic:	226.8	
Date:	Fri, 06 Jan 2017	Prob (F-statistic):	2.23e-27	
Time:	10:45:17	Log-Likelihood:	-381.63	
No. Observations:	46	AIC:	773.3	

Df Residuals:	41	BIC:	782.4		
Df Model:	4				
Covariance Type:	nonrobust				
<hr/>					
	coef	std err	t	P> t	[95.0% Conf. Int.]
Intercept	8035.5976	386.689	20.781	0.000	7254.663 8816.532
education[T.Master]	3144.0352	361.968	8.686	0.000	2413.025 3875.045
education[T.Ph.D]	2996.2103	411.753	7.277	0.000	2164.659 3827.762
management[T.Y]	6883.5310	313.919	21.928	0.000	6249.559 7517.503
experience	546.1840	30.519	17.896	0.000	484.549 607.819
<hr/>					
Omnibus:	2.293		Durbin-Watson:	2.237	
Prob(Omnibus):	0.318		Jarque-Bera (JB):	1.362	
Skew:	-0.077		Prob(JB):	0.506	
Kurtosis:	2.171		Cond. No.	33.5	
<hr/>					
Warnings:					
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.					
sum_sq	df	F	PR(>F)		
education	9.152624e+07	2.0	43.351589	7.672450e-11	management
	5.075724e+08	1.0	480.825394	2.901444e-24	experience
	1.0	320.281524	5.546313e-21		
Residual	4.328072e+07	41.0	NaN	NaN	

Comparing two nested models

oneway is nested within twoway. Comparing two nested models tells us if the additional predictors (i.e. education) of the full model significantly decrease the residuals. Such comparison can be done using an *F*-test on residuals:

```
print(twoway.compare_f_test(oneway))          # return F, pval, df
(43.351589459181056, 7.6724495704954452e-11, 2.0)
```

Factor coding

See

<http://statsmodels.sourceforge.net/devel/contrasts.html>

By default Pandas use “dummy coding”. Explore:

```
print(twoway.model.data.param_names)
print(twoway.model.data.exog[:10, :]) ['Intercept', 'education[T.Master]', 'education[T.Ph.D]', 'management[T.Y]', 'experience']
[[ 1.  0.  0.  1.  1.]
 [ 1.  0.  1.  0.  1.]
 [ 1.  0.  1.  1.  1.]
 [ 1.  1.  0.  0.  1.]
 [ 1.  0.  1.  0.  1.]
 [ 1.  1.  0.  1.  2.]
 [ 1.  1.  0.  0.  2.]
 [ 1.  0.  0.  0.  2.]
 [ 1.  0.  1.  0.  2.]
 [ 1.  1.  0.  0.  3.]]
```

Contrasts and post-hoc tests

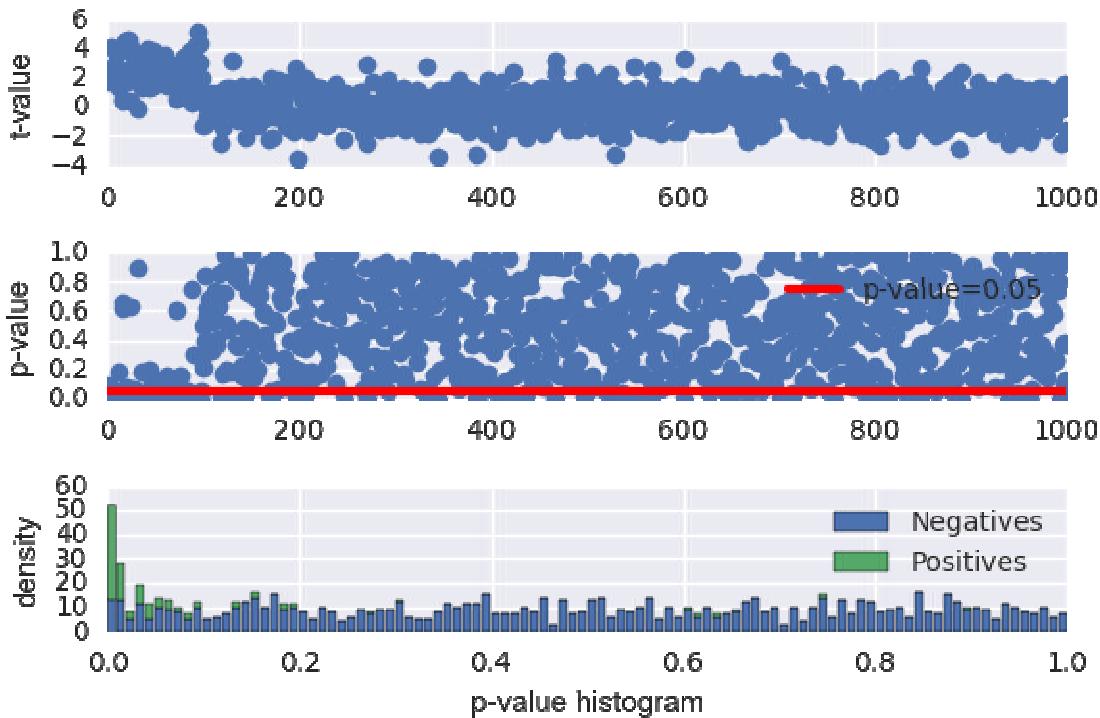
```
# t-test of the specific contribution of experience:  
ttest_exp = twoway.t_test([0, 0, 0, 0, 1])  
ttest_exp.pvalue, ttest_exp.tvalue  
print(ttest_exp)  
# Alternatively, you can specify the hypothesis tests using a string  
twoway.t_test('experience')  
# Post-hoc is salary of Master different salary of Ph.D?  
# ie. t-test salary of Master = salary of Ph.D.  
print(twoway.t_test('education[T.Master] = education[T.Ph.D]'))
```

Test for Constraints				coef	std	err			
t	P> t	[95.0% Conf. Int.]	c0	546.1840	30.519	17.896	0.000	484.549	
				607.819					
Test for Constraints				coef	std	err			
t	P> t	[95.0% Conf. Int.]	c0	147.8249	387.659	0.381	0.705	-	
				635.069	930.719				

Multiple comparisons

```
import numpy as np  
np.random.seed(seed=42) # make example reproducible  
# Dataset  
n_samples, n_features = 100, 1000  
n_info = int(n_features/10) # number of features with information  
n1, n2 = int(n_samples/2), n_samples - int(n_samples/2) snr = .5  
Y = np.random.randn(n_samples, n_features)  
grp = np.array(["g1"] * n1 + ["g2"] * n2)
```

```
# Add some group effect for Pinfo features  
Y[grp=="g1", :n_info] += snr  
import scipy.stats as stats  
import matplotlib.pyplot as plt  
tvals, pvals = np.full(n_features, np.NAN), np.full(n_features, np.NAN)  
for j in range(n_features):  
    tvals[j], pvals[j] = stats.ttest_ind(Y[grp=="g1", j], Y[grp=="g2", j], equal_var=True)  
fig, axis = plt.subplots(3, 1, sharex='col')  
axis[0].plot(range(n_features), tvals, 'o')  
axis[0].set_ylabel("t-value")  
axis[1].plot(range(n_features), pvals, 'o')  
axis[1].axhline(y=0.05, color='red', linewidth=3, label="p-value=0.05")  
#axis[1].axhline(y=0.05, label="toto", color='red')  
axis[1].set_ylabel("p-value")  
axis[1].legend()  
axis[2].hist([pvals[n_info:], pvals[:n_info]], stacked=True, bins=100,  
           label=["Negatives", "Positives"])  
axis[2].set_xlabel("p-value")  
axis[2].set_ylabel("density")  
axis[2].legend()  
plt.tight_layout()
```



Note that under the null hypothesis the distribution of the p -values is uniform.

Statistical measures:

- True Positive (TP) equivalent to a hit. The test correctly concludes the presence of an effect.
- True Negative (TN). The test correctly concludes the absence of an effect.
- False Positive (FP) equivalent to a false alarm, Type I error. The test improperly concludes the presence of an effect. Thresholding at p -value < 0.05 leads to 47 FP.
- False Negative (FN) equivalent to a miss, Type II error. The test improperly concludes the absence of an effect.

```
P, N = n_info, n_features - n_info # Positives, Negatives
TP = np.sum(pvals[:n_info] < 0.05) # True Positives
FP = np.sum(pvals[n_info:] < 0.05) # False Positives

print("No correction, FP: %i (expected: %.2f), TP: %i" % (FP, N * 0.05, TP))
No correction, FP: 47 (expected: 45.00), TP: 71
```

Bonferroni correction for multiple comparisons

The Bonferroni correction is based on the idea that if an experimenter is testing P hypotheses, then one way of maintaining the family wise error rate (FWER) is to test each individual hypothesis at a statistical significance level of $1/P$ times the desired maximum overall level.

So, if the desired significance level for the whole family of tests is α (usually 0.05), then the Bonferroni correction would test each individual hypothesis at a significance level of α/P . For example, if a trial is testing $P = 8$ hypotheses with a desired $\alpha = 0.05$, then the Bonferroni correction would test each individual hypothesis at

$$\alpha = 0.05/8 = 0.00625.$$

```
import statsmodels.sandbox.stats.multicomp as multicomp
pvals_fwer = multicomp.multipletests(pvals, alpha=0.05, method='bonferroni')
TP = np.sum(pvals_fwer[:n_info] < 0.05) # True Positives
FP = np.sum(pvals_fwer[n_info:] < 0.05) # False Positives
print("FWER correction, FP: %i, TP: %i" % (FP, TP))
FWER correction, FP: 0, TP: 6
```

The False discovery rate (FDR) correction for multiple comparisons

FDR-controlling procedures are designed to control the expected proportion of rejected null hypotheses that were incorrect rejections (“false discoveries”). FDR-controlling procedures provide less stringent control of Type I errors compared to the familywise error rate (FWER) controlling procedures (such as the Bonferroni correction), which control the probability of at least one Type I error. Thus, FDR-controlling procedures have greater power, at the cost of increased rates of Type I errors.

```
import statsmodels.sandbox.stats.multicomp as multicomp
pvvals_fdr = multicomp.multipletests(pvals, alpha=0.05, method='fdr_bh')
TP = np.sum(pvvals_fdr[:n_info] < 0.05) # True Positives
FP = np.sum(pvvals_fdr[n_info:] < 0.05) # False Positives
print("FDR correction, FP: %i, TP: %i" % (FP, TP))
```

Multiple comparisons

FDR correction, FP: 3, TP: 20

Exercise

This exercise has 2 goals: apply your knowledge of statistics using vectorized numpy operations. Given the dataset provided for multiple comparison, compute the two-sample t -test (assuming equal variance) for each (column) feature of Y given the two groups defined by grp . You should return two vectors of size $n_features$: one for the t -values and one for the p -values.

CHAPTER

SEVEN

MULTIVARIATE STATISTICS

Multivariate statistics includes all statistical techniques for analyzing samples made of two or more variables. The data set (a $N \times P$ matrix \mathbf{X}) is a collection of N independent samples column vectors $[\mathbf{x}_1, \dots, \mathbf{x}_i, \dots, \mathbf{x}_N]$ of length P .

Linear Algebra

Euclidean norm and distance

The Euclidian norm of a vector $\mathbf{a} \in \mathbb{R}^P$ is denoted

$$\|\mathbf{a}\|_2 = \sqrt{\sum_i^P a_i^2}$$

The Euclidian distance between two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^P$ is

$$\|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{\sum_i^P (a_i - b_i)^2}$$

Dot product and projection

Source: [Wikipedia](#)

In Euclidean space, a Euclidean vector is a geometrical object that possesses both a magnitude and a direction. A vector can be pictured as an arrow. Its magnitude is its length, and its direction is the direction that the arrow points. The magnitude of a vector \mathbf{a} is denoted by $\|\mathbf{a}\|_2$. The dot product of two Euclidean vectors \mathbf{a} and \mathbf{b} is defined by

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|_2 \|\mathbf{b}\|_2 \cos\theta,$$

where θ is the angle between \mathbf{a} and \mathbf{b} .

In particular, if \mathbf{a} and \mathbf{b} are orthogonal, then the angle between them is 90° and

$$\mathbf{a} \cdot \mathbf{b} = 0.$$

At the other extreme, if they are codirectional, then the angle between them is 0° and

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|_2 \|\mathbf{b}\|_2$$

This implies that the dot product of a vector \mathbf{a} by itself is

$$\mathbf{a} \cdot \mathbf{a} = \|\mathbf{a}\|_2^2.$$

The scalar projection (or scalar component) of a Euclidean vector \mathbf{a} in the direction of a Euclidean vector \mathbf{b} is given by

$$a_b = \|\mathbf{a}\|_2 \cos\theta,$$

where θ is the angle between \mathbf{a} and \mathbf{b} .

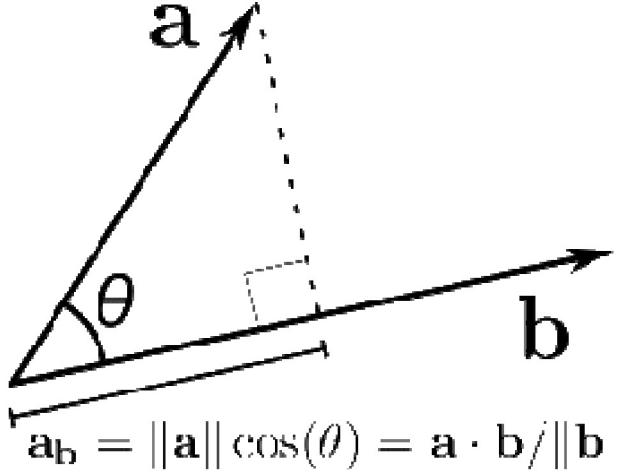


Fig. 7.1: Projection.

```
import numpy as np
np.random.seed(42)
a = np.random.randn(10)
b = np.random.randn(10)
np.dot(a, b)
```

-4.0857885326599241

Mean vector

The mean ($P \times 1$) column-vector μ whose estimator is

$$\bar{\mathbf{x}} = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_i = \frac{1}{N} \sum_{i=1}^N \begin{bmatrix} x_{i1} \\ \vdots \\ x_{ij} \\ \vdots \\ x_{iP} \end{bmatrix} = \begin{bmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_j \\ \vdots \\ \bar{x}_P \end{bmatrix}.$$

Covariance matrix

- The covariance matrix Σ_{xx} is a symmetric positive semi-definite matrix whose element in the j,k position is the covariance between the j^{th} and k^{th} elements of a random vector i.e. the j^{th} and k^{th} columns of \mathbf{X} .
- The covariance matrix generalizes the notion of covariance to multiple dimensions.
- The covariance matrix describe the shape of the sample distribution around the mean assuming an elliptical distribution:

$$\Sigma_{xx} = E(\mathbf{X} - E(\mathbf{X}))^T E(\mathbf{X} - E(\mathbf{X})),$$

whose estimator \mathbf{S}_{xx} is a $P \times P$ matrix given by

$$= \frac{1}{N-1} (\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T)^T (\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T)$$

If we assume that \mathbf{X} is centered, i.e. \mathbf{X} is replaced by $\mathbf{X} - \mathbf{1}\mathbf{x}^{-T}$ then the estimator is

$$= \frac{1}{N-1} \mathbf{X}^T \mathbf{X} = \frac{1}{N-1} \begin{bmatrix} x_{11} & \cdots & x_{N1} \\ x_{1j} & \cdots & x_{Nj} \\ \vdots & & \vdots \\ x_{1P} & \cdots & x_{NP} \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{1k} & x_{1P} \\ \vdots & & \vdots & \vdots \\ x_{N1} & \cdots & x_{Nk} & x_{NP} \end{bmatrix} = \begin{bmatrix} s_{11} & \cdots & s_{1k} & s_{1P} \\ \ddots & & s_{jk} & \vdots \\ & & s_{kk} & s_{kP} \\ & & & s_{PP} \end{bmatrix},$$

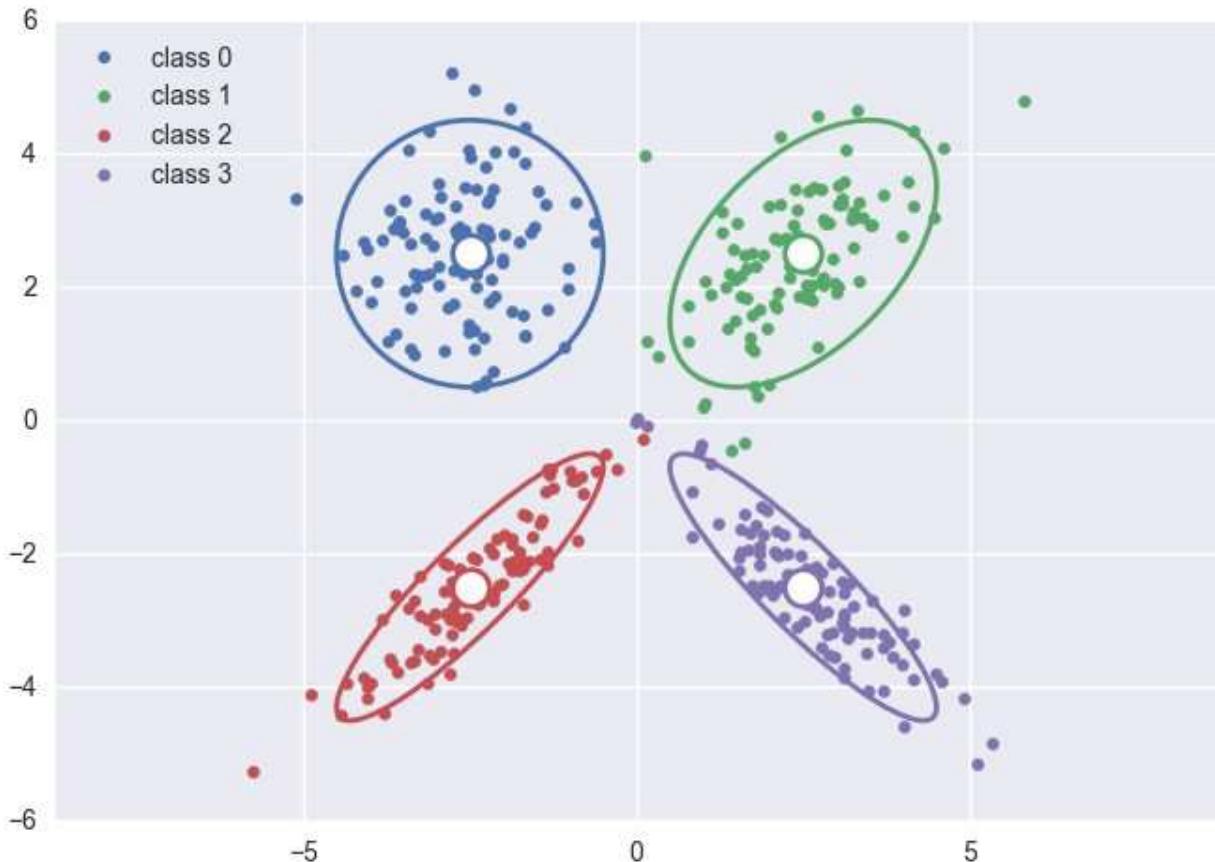
\mathbf{S}_{xx}

where

$$s_{jk} = s_{kj} = \frac{1}{N-1} \mathbf{x}_j^T \mathbf{x}_k = \frac{1}{N-1} \sum_{i=1}^N x_{ij} x_{ik}$$

is an estimator of the covariance between the j^{th} and k^{th} variables.

```
## Avoid warnings and force inline plot
%matplotlib inline import warnings
warnings.filterwarnings("ignore")
import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils
import seaborn as sns # nice color
np.random.seed(42)
colors = sns.color_palette()
n_samples, n_features = 100, 2
mean, Cov, X = [None] * 4, [None] * 4, [None] * 4
mean[0] = np.array([-2.5, 2.5])
Cov[0] = np.array([[1, 0], [0, 1]])
mean[1] = np.array([2.5, 2.5])
Cov[1] = np.array([[1, .5],[.5, 1]])
mean[2] = np.array([-2.5, -2.5])
Cov[2] = np.array([[1, .9],[.9, 1]])
mean[3] = np.array([2.5, -2.5])
Cov[3] = np.array([[1, -.9],[-.9, 1]])
# Generate dataset
for i in range(len(mean)):
    X[i] = np.random.multivariate_normal(mean[i], Cov[i], n_samples)
# Plot
for i in range(len(mean)):
    # Points
    plt.scatter(X[i][:, 0], X[i][:, 1], color=colors[i], label="class %i" % i)
    # Means
    plt.scatter(mean[i][0], mean[i][1], marker="o", s=200, facecolors='w', edgecolors=colors[i], linewidth=2)
    # Ellipses representing the covariance matrices
    pystatsml.plot_utils.plot_cov_ellipse(Cov[i], pos=mean[i], facecolor='none', linewidth=2,
                                           edgecolor=colors[i])
plt.axis('equal')= plt.legend(loc='upper left')
```



Precision matrix

In statistics, precision is the reciprocal of the variance, and the precision matrix is the matrix inverse of the covariance matrix.

It is related to partial correlations that measures the degree of association between two variables, while controlling the effect of other variables.

```
import numpy as np
Cov = np.array([[1.0, 0.9, 0.9, 0.0, 0.0, 0.0],
               [0.9, 1.0, 0.9, 0.0, 0.0, 0.0],
               [0.9, 0.9, 1.0, 0.0, 0.0, 0.0],
               [0.0, 0.0, 0.0, 1.0, 0.9, 0.0],
               [0.0, 0.0, 0.0, 0.9, 1.0, 0.0],
               [0.0, 0.0, 0.0, 0.0, 0.0, 1.0]])
print("# Precision matrix:")
Prec = np.linalg.inv(Cov)
print(Prec.round(2))
print("# Partial correlations:")
Pcor = np.zeros(Prec.shape)
Pcor[:] = np.NaN

for i, j in zip(*np.triu_indices_from(Prec, 1)):
    Pcor[i, j] = - Prec[i, j] / np.sqrt(Prec[i, i] * Prec[j, j])
print(Pcor.round(2))

# Precision matrix:
[[ 6.79   3.21  -3.21    0.     0.      0.    ],
 [-3.21   6.79  -3.21    0.     0.      0.    ],
 [-3.21  -3.21   6.79    0.     0.      0.    ],
 [ 0.     0.     0.     5.26  -4.74    0.    ],
 [ 0.     0.     0.   -4.74   5.26    0.    ]]
```

```
[ 0.      0.      0.      0.      0.      1.    ]]
# Partial corre latio ns:-0.
[[ nan  0.47  0.47     -0.     -0.     -0.    ]
 [ nan  nan   0.47  -0.     -0.     -0.    ]
 [ nan  nan   nan   -0.     -0.     -0.    ]
 [ nan  nan   nan   nan   0.9     -0.    ]
 [ nan  nan   nan   nan   nan   -0.    ]
 [ nan  nan   nan   nan   nan     nan]]
```

Mahalanobis distance

- The Mahalanobis distance is a measure of the distance between two points \mathbf{x} and μ where the dispersion (i.e. the covariance structure) of the samples is taken into account.
- The dispersion is considered through covariance matrix.

This is formally expressed as

$$D_M(\mathbf{x}, \mu) = \sqrt{(\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu)}.$$

Intuitions

- Distances along the principal directions of dispersion are contracted since they correspond to likely dispersion of points.
- Distances orthogonal to the principal directions of dispersion are dilated since they correspond to unlikely dispersion of points.

For example

$$D_M(\mathbf{1}) = \sqrt{\mathbf{1}^T \Sigma^{-1} \mathbf{1}}.$$

```
ones = np.ones(Cov.shape[0])
d_euc = np.sqrt(np.dot(ones, ones))
d_mah = np.sqrt(np.dot(np.dot(ones, Prec), ones))
print("Euclidian norm of ones=%2f. Mahalanobis norm of ones=%2f" % (d_euc, d_mah))
```

Euclidian norm of ones=2.45. Mahalanobis norm of ones=1.77

The first dot product that distances along the principal directions of dispersion are contracted:

```
print(np.dot(ones, Prec))
```

```
[ 0.35714286 0.35714286 0.35714286 0.52631579 0.52631579 1.]
```

```

import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils
%matplotlib inline np.random.seed(40)
colors = sns.color_palette()
mean = np.array([0, 0])
Cov = np.array([[1, .8],[.8, 1]])
samples = np.random.multivariate_normal(mean, Cov, 100)
x1 = np.array([0, 2])
x2 = np.array([2, 2])
plt.scatter(samples[:, 0], samples[:, 1], color=colors[0])
plt.scatter(mean[0], mean[1], color=colors[0], s=200, label="mean")
plt.scatter(x1[0], x1[1], color=colors[1], s=200, label="x1")
plt.scatter(x2[0], x2[1], color=colors[2], s=200, label="x2")
# plot covariance ellipse
pystatsml.plot_utils.plot_cov_ellipse(Cov, pos=mean, facecolor='none', linewidth=2,
edgecolor=colors[0])
# Compute distances
d2_m_x1 = scipy.spatial.distance.euclidean(mean, x1)
d2_m_x2 = scipy.spatial.distance.euclidean(mean, x2)
Covi = scipy.linalg.inv(Cov)
dm_m_x1 = scipy.spatial.distance.mahalanobis(mean, x1, Covi)
dm_m_x2 = scipy.spatial.distance.mahalanobis(mean, x2, Covi)
# Plot distances
vm_x1 = (x1 - mean) / d2_m_x1
vm_x2 = (x2 - mean) / d2_m_x2
jitter = .1
plt.plot([mean[0] - jitter, d2_m_x1 * vm_x1[0] - jitter],
[mean[1], d2_m_x1 * vm_x1[1]], color='k')
plt.plot([mean[0] + jitter, d2_m_x2 * vm_x2[0] - jitter],
[mean[1], d2_m_x2 * vm_x2[1]], color='k')
plt.plot([mean[0] + jitter, dm_m_x1 * vm_x1[0] + jitter],
[mean[1], dm_m_x1 * vm_x1[1]], color='r')
plt.plot([mean[0] + jitter, dm_m_x2 * vm_x2[0] + jitter],
[mean[1], dm_m_x2 * vm_x2[1]], color='r')
plt.legend(loc='lower right')
plt.text(-6.1, 3, 'Euclidian: d(m, x1) = %.1f<d(m, x2) = %.1f % (d2_m_x1, d2_m_x2), color='k')
plt.text(-6.1, 3.5, 'Mahalanobis: d(m, x1) = %.1f>d(m, x2) = %.1f % (dm_m_x1, dm_m_x2), color='r')

```

Mahalanobis distance

```

plt.axis('equal')
print('Euclidian d(m, x1) = %.2f< d(m, x2) = %.2f % (d2_m_x1, d2_m_x2)')
print('Mahalanobis d(m, x1) = %.2f > d(m, x2) = %.2f % (dm_m_x1, dm_m_x2)')

```

If the covariance matrix is the identity matrix, the Mahalanobis distance reduces to the Euclidean distance. If the covariance matrix is diagonal, then the resulting distance measure is called a normalized Euclidean distance.

More generally, the Mahalanobis distance is a measure of the distance between a point \mathbf{x} and a distribution $\mathbb{P}(\mathbf{x}|\mu, \Sigma)$. It is a multi-dimensional generalization of the idea of measuring how many standard deviations away \mathbf{x} is from the mean. This distance is zero if \mathbf{x} is at the mean, and grows as \mathbf{x} moves away from the mean: along each principal component axis, it measures the number of standard deviations from \mathbf{x} to the mean of the distribution.

Multivariate normal distribution

The distribution, or probability density function (PDF) (sometimes just density), of a continuous random variable is a function that describes the relative likelihood for this random variable to take on a given value.

The multivariate normal distribution, or multivariate Gaussian distribution, of a P -dimensional random vector $\mathbf{x} = [x_1, x_2, \dots, x_P]^T$ is

$$\mathcal{N}(\mathbf{x}|\mu, \Sigma) = \frac{1}{(2\pi)^{P/2}|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu)\right\}$$

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats from scipy.stats
import multivariate_normal from mpl_toolkits.mplot3d
import Axes3D
def multivariate_normal_pdf(X, mean, sigma):
    """Multivariate normal probability density function over X (n_samples x n_features)"""
    P = X.shape[1] det = np.linalg.det(sigma)
    norm_const = 1.0 / (((2*np.pi) ** (P/2)) * np.sqrt(det))
    X_mu = X - mu
    inv = np.linalg.inv(sigma)
    d2 = np.sum(np.dot(X_mu, inv) * X_mu, axis=1)
    return norm_const * np.exp(-0.5 * d2)
# mean and covariance
mu = np.array([0, 0])
sigma = np.array([[1, -0.5], [-0.5, 1]])
# x, y grid
x, y = np.mgrid[-3:3:1, -3:3:1]
X = np.stack((x.ravel(), y.ravel())).T
norm = multivariate_normal_pdf(X, mean, sigma).reshape(x.shape)
# Do it with scipy
norm_scp = multivariate_normal(mu, sigma).pdf(np.stack((x, y), axis=2))
assert
np.allclose(norm, norm_scp)
# Plot
fig = plt.figure(figsize=(10, 7))
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, norm, rstride=3, cstride=3,
cmap=plt.cm.coolwarm, linewidth=1, antialiased=False
)
ax.set_zlim(0, 0.2)
ax.xaxis.set_major_locator(plt.LinearLocator(10))
ax.xaxis.set_major_formatter(plt.FormatStrFormatter('%.02f'))
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('p(x)')
plt.title('Bivariate Normal/Gaussian distribution')
fig.colorbar(surf, shrink=0.5, aspect=7, cmap=plt.cm.coolwarm)
plt.show()
```

Exercises

Dot product and Euclidean norm

Given $\mathbf{a} = [2, 1]^T$ and $\mathbf{b} = [1, 1]^T$

1. Write a function `euclidian(x)` that computes the Euclidian norm of vector, \mathbf{x} .
2. Compute the Euclidean norm of \mathbf{a} .
3. Compute the Euclidean distance of $\|\mathbf{a} - \mathbf{b}\|_2$.
4. Compute the projection of \mathbf{b} in the direction of vector \mathbf{a} : b_a .

5. Simulate a dataset \mathbf{X} of $N = 100$ samples of 2-dimensional vectors.
6. Project all samples in the direction of the vector \mathbf{a} .

Covariance matrix and Mahalanobis norm

1. Sample a dataset \mathbf{X} of $N = 100$ samples of 2-dimensional vectors from the bivariate normal distribution

$$\Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8, 1 \end{bmatrix}$$

$\square(\mu, \Sigma)$ where $\mu = [1, 1]^T$ and

2. Compute the mean vector \mathbf{x}^- and center \mathbf{X} . Compare the estimated mean \mathbf{x}^- to the true mean, μ .
3. Compute the empirical covariance matrix \mathbf{S} . Compare the estimated covariance matrix \mathbf{S} to the true covariance matrix, Σ .
4. Compute \mathbf{S}^{-1} (Sinv) the inverse of the covariance matrix by using `scipy.linalg.inv(S)`.
5. Write a function `mahalanobis(x, xbar, Sinv)` that computes the Mahalanobis distance of a vector \mathbf{x} to the mean, \mathbf{x}^- .
6. Compute the Mahalanobis and Euclidian distances of each sample \mathbf{x}_i to the mean \mathbf{x}^- . Store the results in a 100×2 dataframe.

**CHAPTER
EIGHT**

DIMENSION REDUCTION AND FEATURE EXTRACTION

Introduction

In machine learning and statistics, dimensionality reduction or dimension reduction is the process of reducing the number of features under consideration, and can be divided into feature selection (not addressed here) and feature extraction.

Feature extraction starts from an initial set of measured data and builds derived values (features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations. Feature extraction is related to dimensionality reduction.

The goal is to learn a transformation that extracts a few relevant features. This is generally done by exploiting the covariance Σ_{xx} between the input features.

Singular value decomposition (SVD)

Principles

- SVD is a method for transforming correlated variables into a set of uncorrelated ones that better expose the various relationships among the original data items.
- At the same time, SVD is a method for identifying and ordering the dimensions along which data points exhibit the most variation.
- Singular value decomposition is the key part of principal components analysis.

Matrix factorization

The singular values

- \mathbf{D} is a $K \times K$ diagonal matrix made of the singular values of \mathbf{X} with $d_1 \geq d_2 \geq \dots \geq d_K \geq 0$.
- \mathbf{D} scale the projection along the coordinate axes by d_1, d_2, \dots, d_K .
- Singular values are the square roots of the eigenvalues of $\mathbf{X}^T \mathbf{X}$.

The left-singular vectors

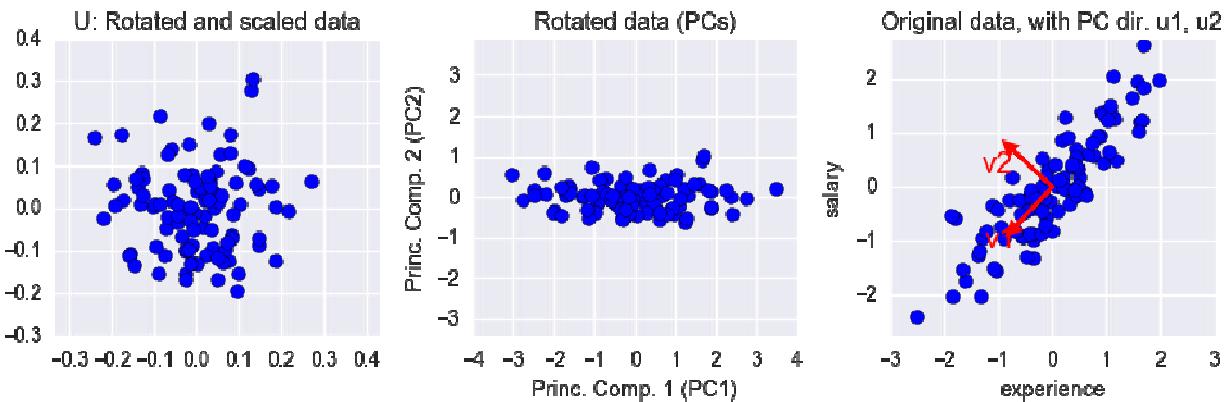
- $\mathbf{U} = [\mathbf{u}_1, \dots, \mathbf{u}_K]$ is an $N \times K$ orthogonal matrix.
- It may be understood as the coordinates on the new orthogonal basis (obtained after the initial rotation) called principal components in the PCA.
- \mathbf{U} form an orthonormal basis for the space spanned by the column vectors of \mathbf{X} .

```

import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns %matplotlib
inline
np.random.seed(42)
# dataset n_samples = 100
experience = np.random.normal(size=n_samples)
salary = 1500 + experience + np.random.normal(size=n_samples, scale=.5)
X = np.column_stack([experience, salary])
# PCA using SVD
X -= X.mean(axis=0) # Centering is required
U, s, Vh = scipy.linalg.svd(X, full_matrices=False)
# U : Unitary matrix having left singular vectors as columns.
# Of shape (n_samples,n_samples) or (n_samples,n_comps), depending on # full_matrices.
# s : The singular values, sorted in non-increasing order. Of shape (n_comps,), # with n_comps =
# min(n_samples, n_features).
# Vh: Unitary matrix having right singular vectors as rows.
# Of shape (n_features, n_features) or (n_comps, n_features) depending

# on full_matrices.
plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(U[:, 0], U[:, 1], s=50)
plt.axis('equal')
plt.title("U: Rotated and scaled data")
plt.subplot(132)
# Project data
PC = np.dot(X, Vh.T)
plt.scatter(PC[:, 0], PC[:, 1], s=50)
plt.axis('equal')
plt.title("Rotated data (PCs)")
plt.xlabel("Princ. Comp. 1 (PC1)")
plt.ylabel("Princ. Comp. 2 (PC2)")
plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], s=50)
for i in range(Vh.shape[0]):
    plt.arrow(x=0, y=0, dx=Vh[i, 0], dy=Vh[i, 1], head_width=0.2, head_length=0.2, linewidth=2, fc='r', ec='r')
    plt.text(Vh[i, 0], Vh[i, 1], 'v%' % (i+1), color="r", fontsize=15, horizontalalignment='right', verticalalignment='top')
plt.axis('equal')
plt.ylim(-4, 4)
plt.title("Original data, with PC dir. u1, u2")
plt.xlabel("experience")
plt.ylabel("salary")
plt.tight_layout()

```



Principal components analysis (PCA)

Sources:

- C. M. Bishop *Pattern Recognition and Machine Learning*, Springer, 2006
- Everything you did and didn't know about PCA
- Principal Component Analysis in 3 Simple Steps

Principles

- Principal components analysis is the main method used for linear dimension reduction.
- The idea of principal component analysis is to find the K principal components directions (called the loadings) $\mathbf{V}_{K \times P}$ that capture the variation in the data as much as possible.
- It converts a set of N P -dimensional observations $\mathbf{X}_{N \times P}$ of possibly correlated variables into a set of N K -dimensional samples $\mathbf{C}_{N \times K}$, where $K < P$. The new variables are linearly uncorrelated. The columns of $\mathbf{C}_{N \times K}$ are called the principal components.
- The dimension reduction is obtained by using only $K < P$ components that exploit correlation (covariance) among the original variables.
- PCA is mathematically defined as an orthogonal linear transformation $\mathbf{V}_{K \times P}$ that transforms the data to a new coordinate system such that the greatest variance by some projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.
$$\mathbf{C}_{N \times K} = \mathbf{X}_{N \times P} \mathbf{V}_{P \times K}$$
- PCA can be thought of as fitting a P -dimensional ellipsoid to the data, where each axis of the ellipsoid represents a principal component. If some axis of the ellipse is small, then the variance along that axis is also small, and by omitting that axis and its corresponding principal component from our representation of the dataset, we lose only a commensurately small amount of information.
- Finding the K largest axes of the ellipse will permit project the data onto a space having dimensionality $K < P$ while maximizing the variance of the projected data.

Dataset pre-processing

Centering

Consider a data matrix, \mathbf{X} , with column-wise zero empirical mean (the sample mean of each column has been shifted to zero), i.e. \mathbf{X} is replaced by $\mathbf{X} - \mathbf{1}\mathbf{x}^{-T}$.

Standardizing

Optionally, standardize the columns, i.e., scale them by their standard-deviation. Without standardization, a variable with a high variance will capture most of the effect of the PCA. The principal direction will be aligned with this variable. Standardization will, however, raise noise variables to the same level as informative variables.

The covariance matrix of centered standardized data is the correlation matrix.

Eigen decomposition of the data covariance matrix

To begin with, consider the projection onto a one-dimensional space ($K = 1$). We can define the direction of this space using a P -dimensional vector \mathbf{v} , which for convenience (and without loss of generality) we shall choose to be a unit vector so that $\|\mathbf{v}\|_2 = 1$ (note that we are only interested in the direction defined by \mathbf{v} , not in the magnitude of \mathbf{v} itself). PCA consists of two main steps:

Projection in the directions that capture the greatest variance

Each P -dimensional data point \mathbf{x}_i is then projected onto \mathbf{v} , where the coordinate (in the coordinate system of \mathbf{v}) is a scalar value, namely $\mathbf{x}_i^T \mathbf{v}$. I.e., we want to find the vector \mathbf{v} that maximizes these coordinates along \mathbf{v} , which we will see corresponds to maximizing the variance of the projected data. This is equivalently expressed as

$$\mathbf{v} = \arg \max_{\|\mathbf{v}\|=1} \frac{1}{N} \sum_i (\mathbf{x}_i^T \mathbf{v})^2$$

We can write this in matrix form as

$$\begin{aligned} \mathbf{v} &= \arg \max_{\|\mathbf{v}\|=1} \frac{1}{N} \|\mathbf{Xv}\|^2 = \frac{1}{N} \mathbf{v}^T \mathbf{X}^T \\ \mathbf{Xv} &= \mathbf{v}^T \mathbf{S}_{xx} \mathbf{v} \end{aligned}$$

where \mathbf{S}_{xx} is a biased estimate of the covariance matrix of the data, i.e.

$$\mathbf{S}_{xx} = \frac{1}{N} \mathbf{X}^T \mathbf{X}$$

We now maximize the projected variance $\mathbf{v}^T \mathbf{S}_{xx} \mathbf{v}$ with respect to \mathbf{v} . Clearly, this has to be a constrained maximization to prevent $\|\mathbf{v}\|_2 \rightarrow \infty$. The appropriate constraint comes from the normalization condition $\|\mathbf{v}\|_2 \equiv \|\mathbf{v}\|_2^2 = \mathbf{v}^T \mathbf{v} = 1$.

To enforce this constraint, we introduce a Lagrange multiplier that we shall denote by λ , and then make an unconstrained maximization of

$$\mathbf{v}^T \mathbf{S}_{xx} \mathbf{v} - \lambda(\mathbf{v}^T \mathbf{v} - 1).$$

By setting the gradient with respect to \mathbf{v} equal to zero, we see that this quantity has a stationary point when

$$\mathbf{S}_{xx} \mathbf{v} = \lambda \mathbf{v}.$$

We note that \mathbf{v} is an eigenvector of \mathbf{S}_{xx} .

If we left-multiply the above equation by \mathbf{v}^T and make use of $\mathbf{v}^T \mathbf{v} = 1$, we see that the variance is given by

$$\mathbf{v}^T \mathbf{S}_{xx} \mathbf{v} = \lambda,$$

and so the variance will be at a maximum when \mathbf{v} is equal to the eigenvector corresponding to the largest eigenvalue, λ . This eigenvector is known as the first principal component.

We can define additional principal components in an incremental fashion by choosing each new direction to be that which maximizes the projected variance amongst all possible directions that are orthogonal to those already considered. If we consider the general case of a K -dimensional projection space, the optimal linear projection for which the variance of the projected data is maximized is now defined by the K eigenvectors, $\mathbf{v}_1, \dots, \mathbf{v}_K$, of the data covariance matrix \mathbf{S}_{xx} that corresponds to the K largest eigenvalues, $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_K$.

Back to SVD

The sample covariance matrix of centered data \mathbf{X} is given by

$$\begin{aligned} \mathbf{X}^T \mathbf{X} &= (\mathbf{U} \mathbf{D} \mathbf{V}^T)^T (\mathbf{U} \mathbf{D} \mathbf{V}^T) \\ &= \mathbf{V} \mathbf{D}^T \mathbf{U}^T \mathbf{U} \mathbf{D} \mathbf{V} \\ &= \mathbf{V} \mathbf{D}^2 \mathbf{V}^T \\ \mathbf{V}^T \mathbf{X}^T \mathbf{X} \mathbf{V} &= \mathbf{D}^2 \\ \frac{1}{N-1} \mathbf{V}^T \mathbf{X}^T \mathbf{X} \mathbf{V} &= \frac{1}{N-1} \mathbf{D}^2 \\ \mathbf{V}^T \mathbf{S}_{xx} \mathbf{V} &= \frac{1}{N-1} \mathbf{D}^2 \end{aligned}$$

Considering only the k^{th} right-singular vectors \mathbf{v}_k associated to the singular value d_k

$$\mathbf{v}_k^T \mathbf{S}_{xx} \mathbf{v}_k = \frac{1}{N-1} d_k^2,$$

It turns out that if you have done the singular value decomposition then you already have the Eigenvalue decomposition for $\mathbf{X}^T \mathbf{X}$. Where - The eigenvectors of \mathbf{S}_{xx} are equivalent to the right singular vectors, \mathbf{V} , of \mathbf{X} .

The eigenvalues, λ_k , of \mathbf{S}_{xx} , i.e. the variances of the components, are equal to $\frac{1}{N-1}$ times the squared singular values, d_k .

Moreover computing PCA with SVD do not require to form the matrix $\mathbf{X}^T \mathbf{X}$, so computing the SVD is now the standard way to calculate a principal components analysis from a data matrix, unless only a handful of components are required.

PCA outputs

The SVD or the Eigen decomposition of the data covariance matrix provides three main quantities:

1. Principal component directions or loadings are the eigenvectors of $\mathbf{X}^T \mathbf{X}$. The $\mathbf{V}_{K \times P}$ or the right-singular vectors of an SVD of \mathbf{X} are called principal component directions of \mathbf{X} . They are generally computed using the SVD of \mathbf{X} .
2. Principal components is the $N \times K$ matrix \mathbf{C} which is obtained by projecting \mathbf{X} onto the principal components directions, i.e.

$$\mathbf{C}_{N \times K} = \mathbf{X}_{N \times P} \mathbf{V}_{P \times K}$$

Since $\mathbf{X} = \mathbf{U} \mathbf{D} \mathbf{V}^T$ and \mathbf{V} is orthogonal ($\mathbf{V}^T \mathbf{V} = \mathbf{I}$):

$$\mathbf{C}_{N \times K} = \mathbf{U} \mathbf{D} \mathbf{V}^T \mathbf{V} \mathbf{P}_{P \times K} \quad (8.1)$$

$$\mathbf{C}_{N \times K} = \mathbf{U} \mathbf{D}_{N \times K}^T \mathbf{I}_{K \times K} \quad (8.2)$$

$$\mathbf{C}_{N \times K} = \mathbf{U} \mathbf{D}_{N \times K}^T \quad (8.3)$$

3. The variance of each component is given by the eigen values $\lambda_k, k = 1, \dots, K$. It can be obtained from the singular

$$\begin{aligned} \text{var}(\mathbf{c}_k) &= \frac{1}{N-1} (\mathbf{X} \mathbf{v}_k)^2 \quad (8.5) \\ &= \frac{1}{N-1} (\mathbf{u}_k d_k)^2 \\ &= \frac{1}{N-1} d_k^2 \end{aligned}$$

Determining the number of PCs

We must choose $K \in [1, \dots, K]$, the number of required components. This can be done by calculating the explained variance ratio of the K first components and by choosing K such that the cumulative explained variance ratio is greater than some given threshold (e.g., $\approx 90\%$). This is expressed as

$$k) = \frac{\sum_j^K (\mathbf{c}_k)}{\sum^K (\mathbf{c})}$$

Interpretation and visualization

PCs

Plot the samples projected on first the principal components as e.g. PC1 against PC2.

PC directions

Exploring the loadings associated with a component provides the contribution of each original variable in the component.

Remark: The loadings (PC directions) are the coefficients of multiple regression of PC on original variables:

$$\mathbf{c} = \mathbf{X}\mathbf{v} \quad (8.8)$$

$$\mathbf{X}^T \mathbf{c} = \mathbf{X}^T \mathbf{X}\mathbf{v} \quad (8.9)$$

$$(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{c} = \mathbf{v} \quad (8.10)$$

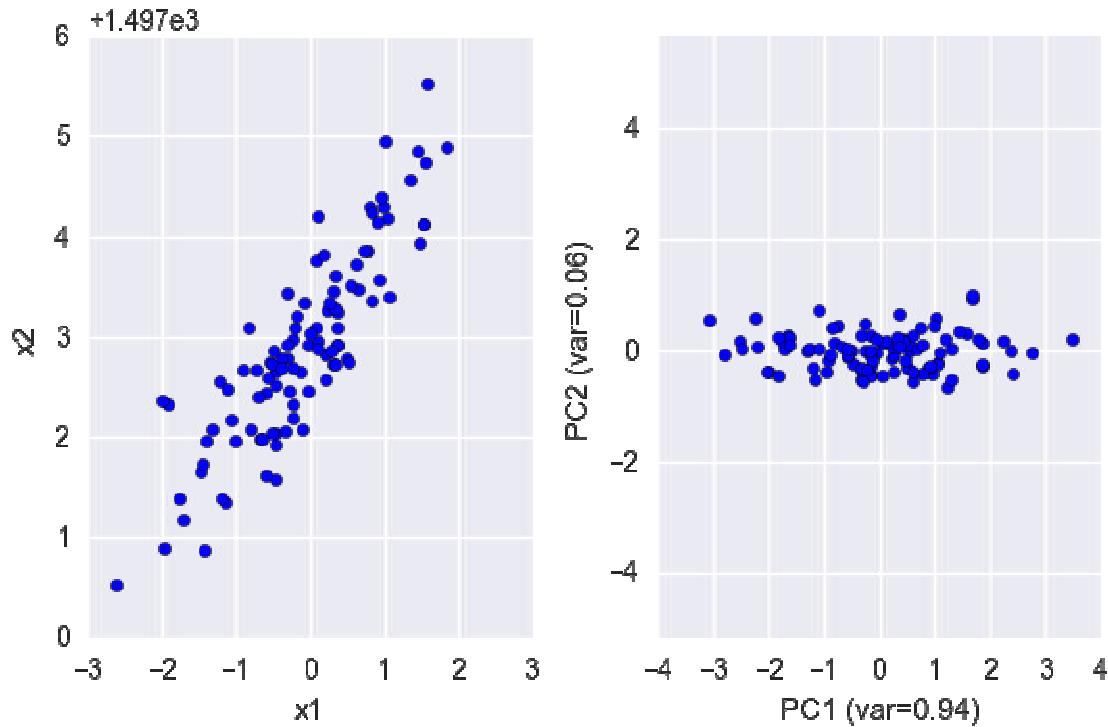
Another way to evaluate the contribution of the original variables in each PC can be obtained by computing the correlation between the PCs and the original variables, i.e. columns of \mathbf{X} , denoted \mathbf{x}_j , for $j = 1, \dots, P$. For the k^{th} PC, compute and plot the correlations with all original variables

$$\text{cor}(\mathbf{c}_k, \mathbf{x}_j), j = 1 \dots K, k = 1 \dots K.$$

These quantities are sometimes called the *correlation loadings*.

```
import numpy as np from
sklearn.decomposition import PCA
import matplotlib.pyplot as plt
np.random.seed(42)
# dataset
n_samples = 100
experience = np.random.normal(size=n_samples)
salary = 1500 + experience + np.random.normal(size=n_samples, scale=.5)
X = np.column_stack([experience, salary])
# PCA with scikit-learn
pca = PCA(n_components=2)
pca.fit(X)
print(pca.explained_variance_ratio_)
PC = pca.transform(X)
plt.subplot(121)
plt.scatter(X[:, 0], X[:, 1])
plt.xlabel("x1");
plt.ylabel("x2")
plt.subplot(122)
plt.scatter(PC[:, 0], PC[:, 1])
plt.xlabel("PC1 (var=%." 2f" % pca.explained_variance_ratio_[0])")
plt.ylabel("PC2 (var=%." 2f" % pca.explained_variance_ratio_[1])")
plt.axis('equal')
plt.tight_layout()
```

```
[ 0.93646607    0.06353393]
```



Exercises

Write a basic PCA class

Write a class BasicPCA with two methods:

- fit(X) that estimates the data mean, principal components directions V and the explained variance of each component.
- transform(X) that projects the data onto the principal components.

Check that your BasicPCA gave similar results, compared to the results from sklearn.

Apply your Basic PCA on the iris dataset

The data set is available at: <https://raw.github.com/neurospin/pystatsml/master/data/iris.csv>

- Describe the data set. Should the dataset been standardized?
- Describe the structure of correlations among variables. Compute a PCA with the maximum number of components.
- Compute the cumulative explained variance ratio. Determine the number of components K by your computed values.
- Print the K principal components directions and correlations of the K principal components with the original variables. Interpret the contribution of the original variables into the PC.
- Plot the samples projected into the Kfirst PCs.
- Color samples by their species.

Multi-dimensional Scaling (MDS)

Resources:

- http://www.stat.pitt.edu/sungkyu/course/2221Fall13/lec8_mds_combined.pdf
- https://en.wikipedia.org/wiki/Multidimensional_scaling
- Hastie, Tibshirani and Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer, Second Edition.

The purpose of MDS is to find a low-dimensional projection of the data in which the pairwise distances between data points is preserved, as closely as possible (in a least-squares sense).

- Let \mathbf{D} be the $(N \times N)$ pairwise distance matrix where d_{ij} is a distance between points i and j .

- The MDS concept can be extended to a wide variety of data types specified in terms of a similarity matrix.

Given the dissimilarity (distance) matrix $\mathbf{D}_{N \times N} = [d_{ij}]$, MDS attempts to find K -dimensional projections of the N points $\mathbf{x}_1, \dots, \mathbf{x}_N \in \mathbb{R}^K$, concatenated in an $\mathbf{X}_{N \times K}$ matrix, so that $d_{ij} \approx \|\mathbf{x}_i - \mathbf{x}_j\|$ are as close as possible. This can be obtained by the minimization of a loss function called the stress function

$$\text{stress}(\mathbf{X}) = \sum_{i,j} (d_{ij} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2.$$

This loss function is known as *least-squares* or *Kruskal-Shepard scaling*. A modification of *least-squares* scaling is the *Sammon mapping*

$$\text{stress}_{\text{Sammon}}(\mathbf{X}) = \sum_{i \neq j} \frac{(d_{ij} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2}{d_{ij}}.$$

The Sammon mapping performs better at preserving small distances compared to the *least-squares* scaling.

Classical multidimensional scaling

Also known as *principal coordinates analysis*, PCoA.

- The distance matrix, \mathbf{D} , is transformed to a *similarity matrix*, \mathbf{B} , often using centered inner products.
- The loss function becomes

$$\text{stress}_{\text{classical}}(\mathbf{X}) = \sum_{i,j} (b_{ij} - \mathbf{x}_i \cdot \mathbf{x}_j)^2$$

- The stress function in classical MDS is sometimes called *strain*.
- The solution for the classical MDS problems can be found from the eigenvectors of the similarity matrix.
- If the distances in \mathbf{D} are Euclidean and double centered inner products are used, the results are equivalent to PCA.

Multi-dimensional Scaling (MDS)

Example

The eurodist dataset provides the road distances (in kilometres) between 21 cities in Europe. Given this matrix of peewees (non-Euclidian) distances $\mathbf{D} = [d_{ij}]$, MDS can be used to recover the coordinates of the cities in some Euclidian referential whose orientation is arbitrary.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# Pairwise distance between european cities try:
url = './data/eurodist.csv'
df = pd.read_csv(url)
except:
url = 'https://raw.githubusercontent.com/neurospin/pystatsml/master/data/eurodist.csv'
df = pd.read_csv(url)
print(df.ix[:5, :5])
city = df["city"]
D = np.array(df.ix[:, 1:]) # Distance matrix
# Arbitrary choice of K=2 components
from sklearn.manifold import MDS
mds = MDS(dissimilarity='precomputed', n_components=2, random_state=40, max_iter=3000,
eps=1e-9)
X = mds.fit_transform(D)
city Athens Barcelona Brussels Calais
```

0	Athens	0	3313	2963	3175
1	Barcelona	3313	0	1318	1326
2	Brussels	2963	1318	0	204
3	Calais	3175	1326	204	0
4	Cherbourg	3339	1294	583	460
5	Cologne	2762	1498	206	409

Recover coordinates of the cities in Euclidian referential whose orientation is arbitrary:

```
from sklearn import metrics
Deuclidian = metrics.pairwise.pairwise_distances(X, metric='euclidean')
print(np.round(Deuclidian[:5, :5]))
```

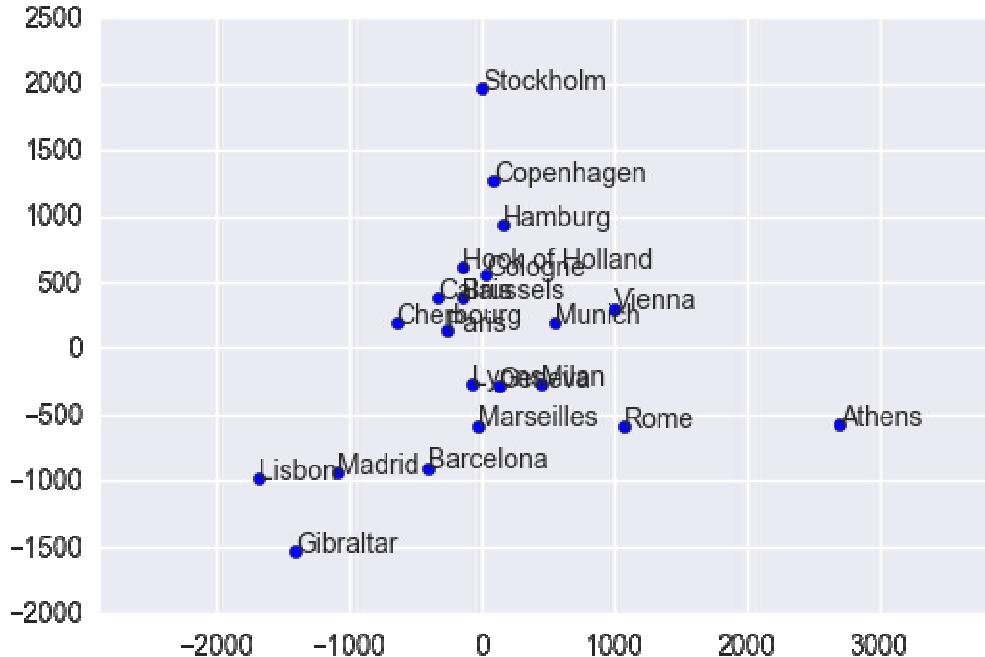
```
[[ 0. 3116. 2994. 3181. 3428.]
 [3116. 0. 1317. 1289. 1128.]
 [2994. 1317. 0. 198. 538.]
 [3181. 1289. 198. 0. 358.]
 [3428. 1128. 538. 358. 0.]]
```

Plot the results:

```
# Plot: apply some rotation and flip
theta = 80 * np.pi / 180.
rot = np.array([[np.cos(theta), -np.sin(theta)],[np.sin(theta), np.cos(theta)]])
Xr = np.dot(X, rot)
# flip x

Xr[:, 0] *= -1
plt.scatter(Xr[:, 0], Xr[:, 1])
for i in range(len(city)):
    plt.text(Xr[i, 0], Xr[i, 1], city[i])
plt.axis('equal')
```

```
(-2000.0, 3000.0, -2000.0, 2500.0)
```



Determining the number of components

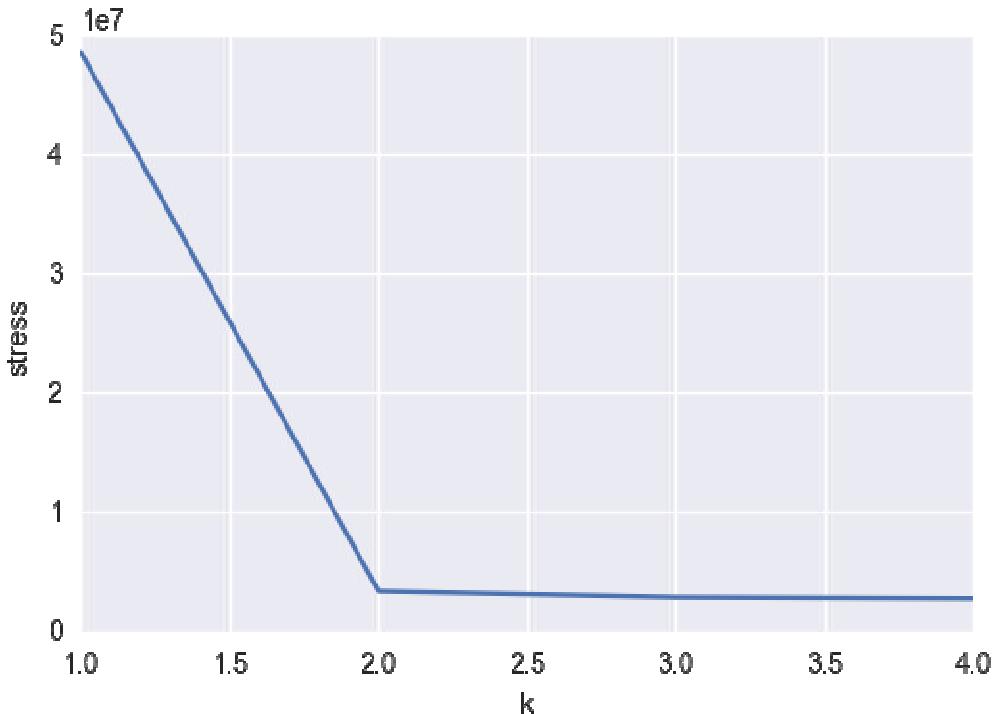
We must choose $K \in \{1, \dots, K\}$ the number of required components. Plotting the values of the stress function, obtained using $k \leq N - 1$ components. In general, start with $1, \dots, K \leq 4$. Choose K where you can clearly distinguish an *elbow* in the stress curve.

Thus, in the plot below, we choose to retain information accounted for by the first two components, since this is where the *elbow* is in the stress curve.

```
k_range = range(1, min(5, D.shape[0]-1))
stress = [MDS(dissimilarity='precomputed', n_components=k, random_state=42, max_iter=300, eps=1e-9).fit(D).stress_ for k in k_range]
print(stress)
plt.plot(k_range, stress)
plt.xlabel("k")
plt.ylabel("stress")
```

```
[48644495.285714284, 3356497.3657523869, 2858455.4958879612, 2756310.6376280105]
```

```
<matplotlib.text.Text at 0x7ff969c19978>
```



Exercises

Apply MDS from sklearn on the iris dataset available at:

<https://raw.github.com/neurospin/pystatsml/master/data/iris.csv>

- Center and scale the dataset.
- Compute Euclidean pairwise distances matrix.
- Select the number of components.
- Show that classical MDS on Euclidean pairwise distances matrix is equivalent to PCA.

Nonlinear dimensionality reduction

Sources:

- Scikit-learn documentation
- Wikipedia

Nonlinear dimensionality reduction or manifold learning cover unsupervised methods that attempt to identify low dimensional manifolds within the original P -dimensional space that represent high data density. Then those methods provide a mapping from the high-dimensional space to the low-dimensional embedding.

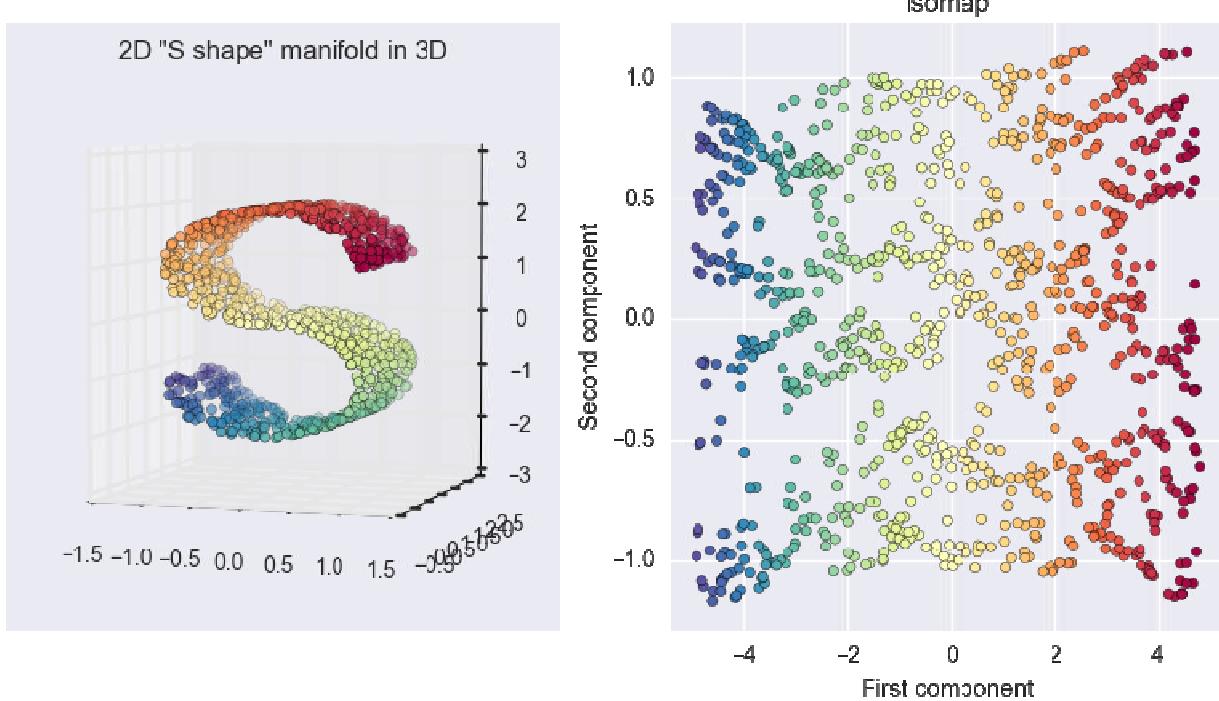
Isomap

Isomap is a nonlinear dimensionality reduction method that combines a procedure to compute the distance matrix with MDS. The distances calculation is based on geodesic distances evaluated on neighbourhood graph:

1. Determine the neighbors of each point. All points in some fixed radius or K nearest neighbors.
2. Construct a neighbourhood graph. Each point is connected to other if it is a K nearest neighbour. Edge length equal to Euclidean distance.
3. Compute shortest path between pairwise of points d_{ij} to build the distance matrix \mathbf{D} .
4. Apply MDS on \mathbf{D} .

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import manifold, datasets
X, color = datasets.samples_generator.make_s_curve(1000, random_state=42)
fig = plt.figure(figsize=(10, 5))
plt.suptitle("Isomap Manifold Learning", fontsize=14)
ax = fig.add_subplot(121, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)
plt.title('2D "S shape" manifold in 3D')
Y = manifold.Isomap(n_neighbors=10, n_components=2).fit_transform(X)
ax = fig.add_subplot(122)
plt.scatter(Y[:, 0], Y[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("Isomap")
plt.xlabel("First component")
plt.ylabel("Second component")
plt.axis('tight')
(-5.4131242078919239,
 5.2729984345096854,
 -1.2877687637642998,
 1.2316524684384262)
```

Isomap Manifold Learning



**CHAPTER
NINE**

CLUSTERING

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). Clustering is one of the main tasks of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, and bioinformatics.

Sources: <http://scikit-learn.org/stable/modules/clustering.html>

K-means clustering

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

Suppose we have a data set $X = \{x_1, \dots, x_N\}$ that consists of N observations of a random D -dimensional Euclidean variable x . Our goal is to partition the data set into some number, K , of clusters, where we shall suppose for the moment that the value of K is given. Intuitively, we might think of a cluster as comprising a group of data points whose inter-point distances are small compared to the distances to points outside of the cluster. We can formalize this notion by first introducing a set of D -dimensional vectors μ_k , where $k = 1, \dots, K$, in which μ_k is a prototype associated with the k^{th} cluster. As we shall see shortly, we can think of the μ_k as representing the centres of the clusters. Our goal is then to find an assignment of data points to clusters, as well as a set of vectors $\{\mu_k\}$, such that the sum of the squares of the distances of each data point to its closest prototype vector μ_k , is at a minimum.

It is convenient at this point to define some notation to describe the assignment of data points to clusters. For each data point x_i , we introduce a corresponding set of binary indicator variables $r_{ik} \in \{0, 1\}$, where $k = 1, \dots, K$, that describes which of the K clusters the data point x_i is assigned to, so that if data point x_i is assigned to cluster k then $r_{ik} = 1$, and $r_{ij} = 0$ for $j \neq k$. This is known as the 1-of- K coding scheme. We can then define an objective function, denoted inertia, as

$$J(r, \mu) = \sum_i \sum_k r_{ik} \|x_i - \mu_k\|_2^2$$

which represents the sum of the squares of the Euclidean distances of each data point to its assigned vector μ_k . Our goal is to find values for the $\{r_{ik}\}$ and the $\{\mu_k\}$ so as to minimize the function J . We can do this through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to the r_{ik} and the μ_k . First we choose some initial values for the μ_k . Then in the first phase we minimize J with respect to the r_{ik} , keeping the μ_k fixed. In the second phase we minimize J with respect to the μ_k , keeping r_{ik} fixed. This two-stage optimization process is then repeated until convergence. We shall see that these two stages of updating r_{ik} and μ_k correspond respectively to the expectation (E) and maximization (M) steps of the expectation maximisation (EM) algorithm, and to emphasize this we shall use the terms E step and M step in the context of the K-means algorithm.

Consider first the determination of the r_{ik} . Because J is a linear function of r_{ik} , this optimization can be performed easily to give a closed form solution. The terms involving different i are independent and so we can optimize for each i separately by choosing r_{ik} to be 1 for whichever value of k gives the minimum value of $\|x_i - \mu_k\|^2$. In other words, we simply assign the i^{th} data point to the closest cluster centre.

Now consider the optimization of the μ_k with the r_{ik} held fixed. The objective function J is a quadratic function of μ_k , and it can be minimized by setting its derivative with respect to μ_k to zero giving

$$2 \sum_i r_{ik} (x_i - \mu_k) = 0$$

which we can easily solve for μ_k to give

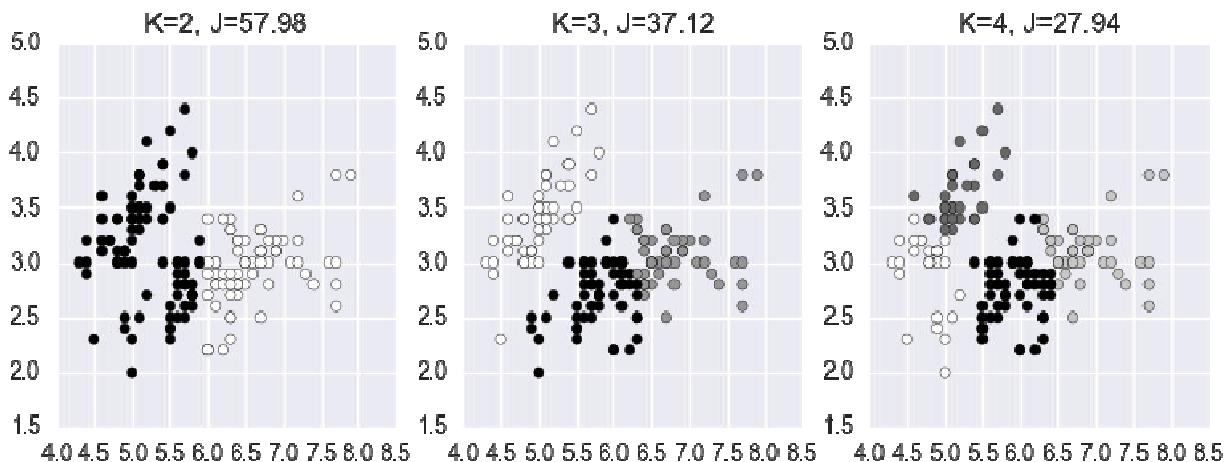
$$\mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}}.$$

The denominator in this expression is equal to the number of points assigned to cluster k , and so this result has a simple interpretation, namely set μ_k equal to the mean of all of the data points x_i assigned to cluster k . For this reason, the procedure is known as the K -means algorithm.

The two phases of re-assigning data points to clusters and re-computing the cluster means are repeated in turn until there is no further change in the assignments (or until some maximum number of iterations is exceeded). Because each phase reduces the value of the objective function J , convergence of the algorithm is assured. However, it may converge to a local rather than global minimum of J .

```
from sklearn import cluster, datasets
import matplotlib.pyplot as plt
import seaborn as sns # nice color
%matplotlib inline
iris = datasets.load_iris()
X = iris.data[:, :2] # use only 'sepal length' and 'sepal width'
y_iris = iris.target
km2 = cluster.KMeans(n_clusters=2).fit(X)
km3 = cluster.KMeans(n_clusters=3).fit(X)
km4 = cluster.KMeans(n_clusters=4).fit(X)
plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=km2.labels_)
plt.title("K=2, J=%f" % km2.inertia_)
plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=km3.labels_)
plt.title("K=3, J=%f" % km3.inertia_)
plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=km4.labels_)#.astype(np.float))
plt.title("K=4, J=%f" % km4.inertia_)
```

<matplotlib.text.Text at 0x7fe4ad47b710>



Exercises

1. Analyse clusters

- Analyse the plot above visually. What would a good value of K be?
- If you instead consider the inertia, the value of J , what would a good value of K be?
- Explain why there is such difference.

- For $K = 2$ why did K -means clustering not find the two “natural” clusters? See the assumptions of K -means: http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_assumptions.html#example-cluster-plot-kmeans-assumptions-py

2. Re-implement the K -means clustering algorithm (homework)

Write a function `kmeans(X,K)` that return an integer vector of the samples' labels.

Hierarchical clustering

Hierarchical clustering is an approach to clustering that build hierarchies of clusters in two main approaches:

- Agglomerative: A *bottom-up* strategy, where each observation starts in their own cluster, and pairs of clusters are merged upwards in the hierarchy.
- Divisive: A *top-down* strategy, where all observations start out in the same cluster, and then the clusters are split recursively downwards in the hierarchy.

In order to decide which clusters to merge or to split, a measure of dissimilarity between clusters is introduced. More specific, this comprise a *distance* measure and a *linkage* criterion. The distance measure is just what it sounds like, and the linkage criterion is essentially a function of the distances between points, for instance the minimum distance between points in two clusters, the maximum distance between points in two clusters, the average distance between points in two clusters, etc. One particular linkage criterion, the Ward criterion, will be discussed next.

Ward clustering

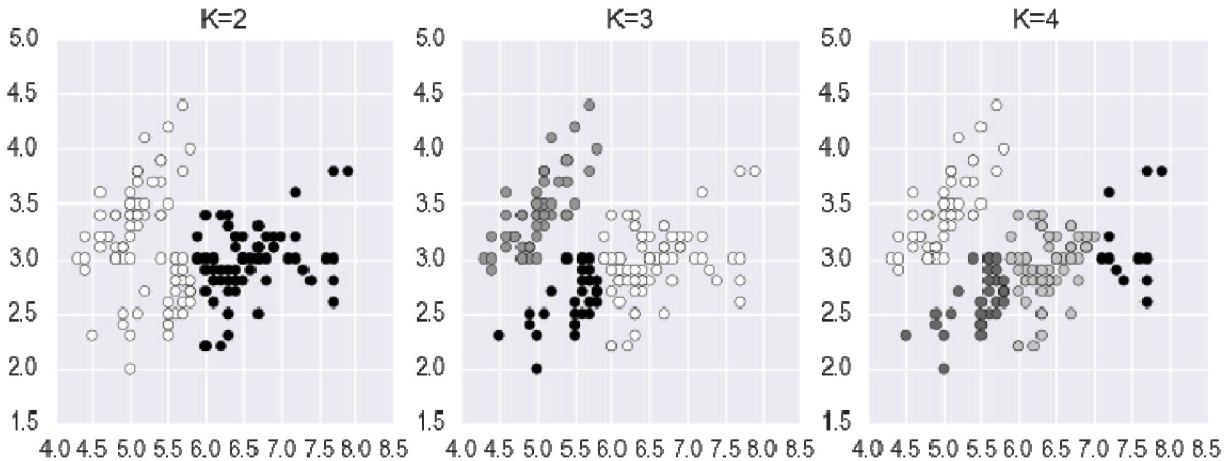
Ward clustering belongs to the family of agglomerative hierarchical clustering algorithms. This means that they are based on a “bottoms up” approach: each sample starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.

In Ward clustering, the criterion for choosing the pair of clusters to merge at each step is the minimum variance criterion. Ward's minimum variance criterion minimizes the total within-cluster variance by each merge. To implement this method, at each step: find the pair of clusters that leads to minimum increase in total within-cluster variance after merging. This increase is a weighted squared distance between cluster centers.

The main advantage of agglomerative hierarchical clustering over K -means clustering is that you can benefit from known neighbourhood information, for example, neighbouring pixels in an image.

```
from sklearn import cluster, datasets
import matplotlib.pyplot as plt
import seaborn as sns # nice color
iris = datasets.load_iris()
X = iris.data[:, :2] # 'sepal length (cm)' 'sepal width (cm)'
y_iris = iris.target
ward2 = cluster.AgglomerativeClustering(n_clusters=2, linkage='ward').fit(X)
ward3 = cluster.AgglomerativeClustering(n_clusters=3, linkage='ward').fit(X)
ward4 = cluster.AgglomerativeClustering(n_clusters=4, linkage='ward').fit(X)
plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=ward2.labels_)
plt.title("K=2")
plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=ward3.labels_)
plt.title("K=3")
plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=ward4.labels_) # .astype(np.float))
plt.title("K=4")
```

<matplotlib.text.Text at 0x7fe4ace5cf0>



Gaussian mixture models

The Gaussian mixture model (GMM) is a simple linear superposition of Gaussian components over the data, aimed at providing a rich class of density models. We turn to a formulation of Gaussian mixtures in terms of discrete latent variables: the K hidden classes to be discovered. Differences compared to K -means:

- Whereas the K -means algorithm performs a hard assignment of data points to clusters, in which each data point is associated uniquely with one cluster, the GMM algorithm makes a soft assignment based on posterior probabilities.
- Whereas the classic K -means is only based on Euclidean distances, classic GMM use a Mahalanobis distances that can deal with non-spherical distributions. It should be noted that Mahalanobis could be plugged within an improved version of K -Means clustering. The Mahalanobis distance is unit less and scale-invariant, and takes into account the correlations of the data set.

The Gaussian mixture distribution can be written as a linear superposition of K Gaussians in the form:

$$p(x) = \sum_{k=1}^K \pi_k(x|\mu_k, \Sigma_k) p(k),$$

where:

- $\pi(x|\mu_k, \Sigma_k)$ is the multivariate Gaussian distribution defined over a P -dimensional vector x of continuous variables.
- The $p(k)$ are the mixing coefficients also known as the class probability of class k , and they sum to one:

$$\sum_{k=1}^K p(k) = 1.$$
- $\pi(x|\mu_k, \Sigma_k) = p(x|k)$ is the conditional distribution of x given a particular class k .

The goal is to maximize the log-likelihood of the GMM:

$$\ln \prod_{i=1}^N \left\{ \sum_{k=1}^K \mathcal{N}(x_i | \mu_k, \Sigma_k) p(k) \right\} = \sum_{i=1}^N \ln \left\{ \sum_{k=1}^K \mathcal{N}(x_i | \mu_k, \Sigma_k) p(k) \right\}.$$

To compute the classes parameters: $p(k), \mu_k, \Sigma_k$ we sum over all samples, by weighting each sample i by its responsibility or contribution to class k : $p(k|x_i)$ such that for each point its contribution to all classes sum to one $\sum_k p(k|x_i) = 1$. This contribution is the conditional probability of class k given x : $p(k|x)$ (sometimes called the posterior). It can be computed using Bayes' rule:

$$p(k|x) = \frac{p(x|k)p(k)}{p(x)}$$

$$= \frac{\mathcal{N}(x | \mu_k, \Sigma_k)p(k)}{\sum_{k=1}^K \mathcal{N}(x | \mu_k, \Sigma_k)p(k)}$$

Since the class parameters, $p(k)$, μ_k and Σ_k , depend on the responsibilities $p(k | x)$ and the responsibilities depend on class parameters, we need a two-step iterative algorithm: the expectation-maximization (EM) algorithm. We discuss this algorithm next.

The expectation-maximization (EM) algorithm for Gaussian mixtures

Given a Gaussian mixture model, the goal is to maximize the likelihood function with respect to the parameters (comprised of the means and covariances of the components and the mixing coefficients).

Initialize the means μ_k , covariances Σ_k and mixing coefficients $p(k)$

Gaussian mixture models

1. E step. For each sample i , evaluate the responsibilities for each class k using the current parameter values

$$p(k | x_i) = \frac{\mathcal{N}(x_i | \mu_k, \Sigma_k)p(k)}{\sum_{k=1}^K \mathcal{N}(x_i | \mu_k, \Sigma_k)p(k)}$$

2. M step. For each class, re-estimate the parameters using the current responsibilities

$$\begin{aligned}\mu_k^{\text{new}} &= \frac{1}{N_k} \sum_{i=1}^N p(k | x_i) x_i \\ \Sigma_k^{\text{new}} &= \frac{1}{N_k} \sum_{i=1}^N p(k | x_i) (x_i - \mu_k^{\text{new}})(x_i - \mu_k^{\text{new}})^T \\ p^{\text{new}}(k) &= \frac{N_k}{N}\end{aligned}$$

Evaluate the log-likelihood

$$\sum_{i=1}^N \ln \left\{ \sum_{k=1}^K \mathcal{N}(x | \mu_k, \Sigma_k)p(k) \right\},$$

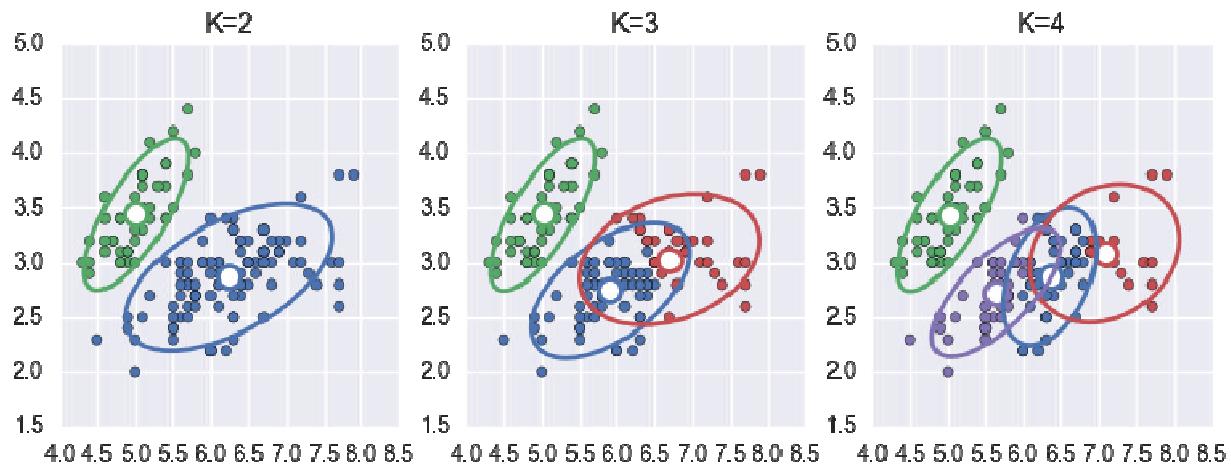
and check for convergence of either the parameters or the log-likelihood. If the convergence criterion is not satisfied return to step 1.

```
import numpy as np from sklearn
import datasets
import matplotlib.pyplot as plt
import seaborn as sns # nice color
import sklearn from sklearn.mixture
import GaussianMixture
import pystatsml.plot_utils
colors = sns.color_palette()
iris = datasets.load_iris()
X = iris.data[:, :2] # 'sepal length (cm)' 'sepal width (cm)'
y_iris = iris.target
gmm2 = GaussianMixture(n_components=2, covariance_type='full').fit(X)
gmm3 = GaussianMixture(n_components=3, covariance_type='full').fit(X)
gmm4 = GaussianMixture(n_components=4, covariance_type='full').fit(X)
plt.figure(figsize=(9, 3)) plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=[colors[i] for i in gmm2.predict(X)])
for lab in gmm2.predict(X):
    for i in range(gmm2.covariances_.shape[0]):
```

```

pystatsml.plot_utils.plot_cov_ellipse(cov=gmm2.covariances_[i, :], pos=gmm2.means[i, :], facecolor='none',
linewidth=2, edgecolor=colors[i])
plt.scatter(gmm2.means_[i, 0], gmm2.means_[i, 1], edgecolor=colors[i], marker="o", s=100, facecolor="w",
linewidth=2)
plt.title("K=2")
plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=[colors[lab]
for lab in gmm3.predict(X)])
for i in range(gmm3.covariances_.shape[0]):
    pystatsml.plot_utils.plot_cov_ellipse(cov=gmm3.covariances_[i, :], pos=gmm3.means_[:, i, :],
facecolor='none', linewidth=2, edgecolor=colors[i])
plt.scatter(gmm3.means_[i, 0], gmm3.means_[i, 1], edgecolor=colors[i], marker="o", s=100, facecolor="w",
linewidth=2)
plt.title("K=3")
plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=[colors[lab]
for lab in gmm4.predict(X)]) # astype(np.float) for i in range(gmm4.covariances_.shape[0]):
    pystatsml.plot_utils.plot_cov_ellipse(cov=gmm4.covariances_[i, :], pos=gmm4.means[i, :], facecolor='none',
linewidth=2, edgecolor=colors[i])
    plt.scatter(gmm4.means_[i, 0], gmm4.means_[i, 1], edgecolor=colors[i], marker="o", s=100, facecolor="w",
linewidth=2)
plt.title("K=4")

```



Model selection

Bayesian information criterion

In statistics, the Bayesian information criterion (BIC) is a criterion for model selection among a finite set of models; the model with the lowest BIC is preferred. It is based, in part, on the likelihood function and it is closely related to the Akaike information criterion (AIC).

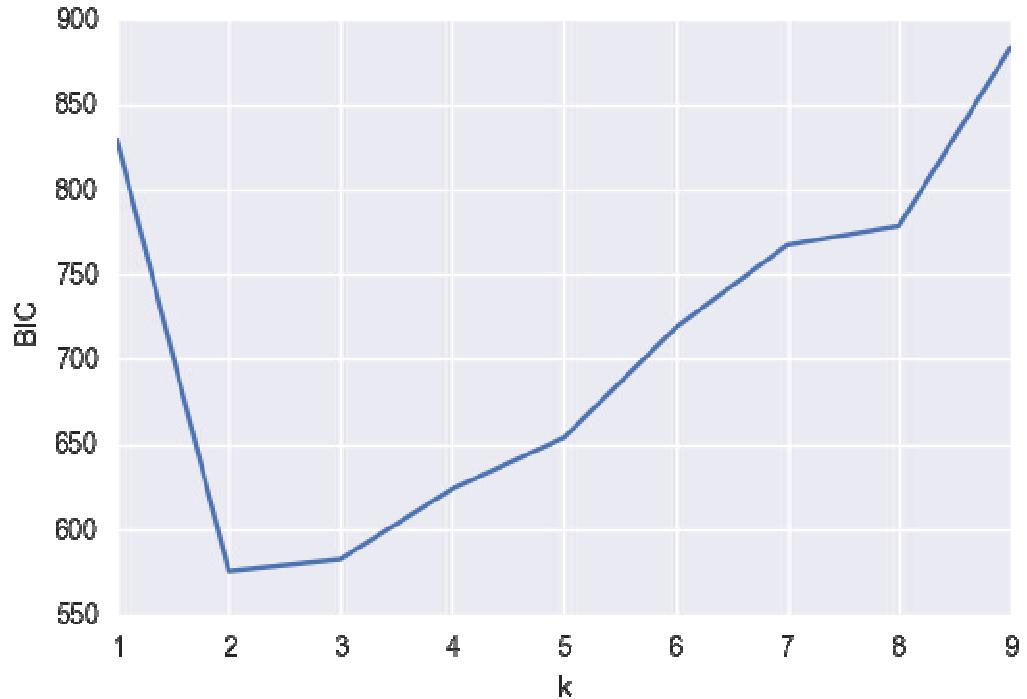
```

X = iris.data
y_iris = iris.target
bic = list() #print(X)
ks = np.arange(1, 10)
for k in ks:
    gmm = GaussianMixture(n_components=k, covariance_type='full')
    gmm.fit(X)
    bic.append(gmm.bic(X))

```

```
k_chosen = ks[np.argmin(bic)]
```

```
plt.plot(ks, bic)
plt.xlabel("K")
plt.ylabel("BIC")
print("Choose k=", k_chosen)
Choose k= 2
```



CHAPTER TEN

LINEAR METHODS FOR REGRESSION

Ordinary least squares

Linear regression models the output, or target variable $y \in \mathbb{R}$ as a linear combination of the $(P - 1)$ -dimensional input $x \in \mathbb{R}^{(P-1)}$. Let \mathbf{X} be the $N \times P$ matrix with each row an input vector (with a 1 in the first position), and similarly let y be the N -dimensional vector of outputs in the training set, the linear model will predict the y given \mathbf{X} using the parameter vector, or weight vector $\beta \in \mathbb{R}^P$ according to

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon,$$

where $\varepsilon \in \mathbb{R}^N$ are the residuals, or the errors of the prediction. The β is found by minimizing an objective function, which is the loss function, $\square(\beta)$, i.e. the error measured on the data. This error is the sum of squared errors (SSE) loss. Minimizing the SSE is the Ordinary Least Square OLS regression as objective function.

$$\text{OLS}(\beta) = \square(\beta) \tag{10.1}$$

$$= \text{SSE}(\beta) \tag{10.2}$$

$$= \sum_i^N (y_i - \mathbf{x}_i^\top \beta)^2 \tag{10.3}$$

$$= (\mathbf{y} - \mathbf{X}\beta)^\top (\mathbf{y} - \mathbf{X}\beta) \tag{10.4}$$

$$= \|\mathbf{y} - \mathbf{X}\beta\|_2^2, \tag{10.5}$$

which is a simple ordinary least squares (OLS) minimization.

Linear regression with scikit-learn

Scikit learn offer many models for supervised learning, and they all follow the same application programming interface (API), namely:

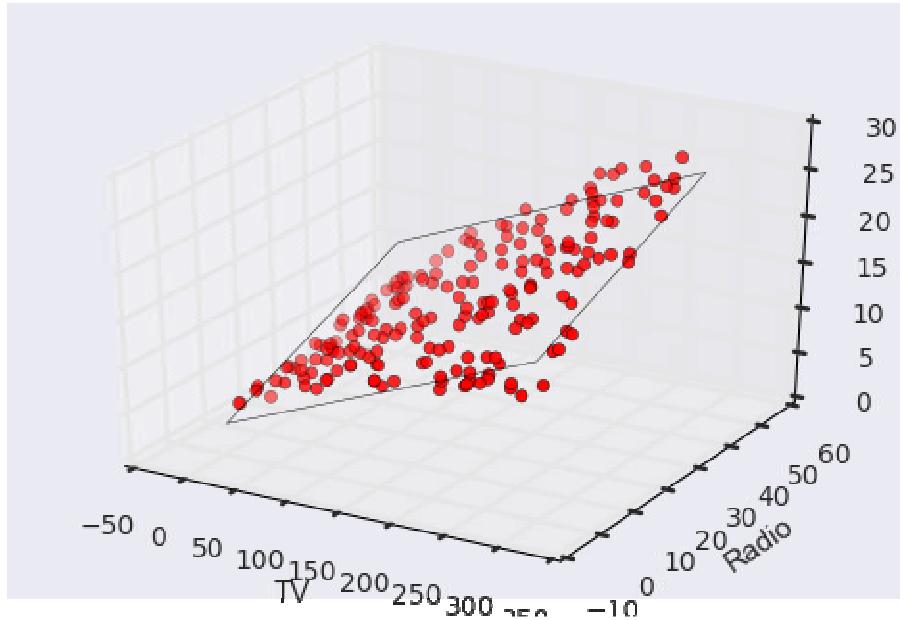
```
model = Estimator()
model.fit(X, y)
predictions = model.predict(X)

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model as lm
import sklearn.metrics as metrics
```

```
%matplotlib inline
```

```
# Fit Ordinary Least Squares: OLS
csv = pd.read_csv('http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv', index_col=0)
X = csv[['TV', 'Radio']]
y = csv['Sales']
lr = lm.LinearRegression().fit(X, y)
y_pred = lr.predict(X)
print("R-squared =", metrics.r2_score(y, y_pred))
print("Coefficients =", lr.coef_)
# Plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(csv['TV'], csv['Radio'], csv['Sales'], c='r', marker='o')
xx1, xx2 = np.meshgrid(np.linspace(csv['TV'].min(), csv['TV'].max(), num=10),
                       np.linspace(csv['Radio'].min(), csv['Radio'].max(), num=10))
XX = np.column_stack([xx1.ravel(), xx2.ravel()])
yy = lr.predict(XX) ax.plot_surface(xx1, xx2, yy.reshape(xx1.shape), color='None')
ax.set_xlabel('TV') ax.set_ylabel('Radio') _ = ax.set_zlabel('Sales')
```

R-squared = 0.897194261083
Coefficients = [0.04575482 0.18799423]



Overfitting

In statistics and machine learning, overfitting occurs when a statistical model describes random errors or noise instead of the underlying relationships. Overfitting generally occurs when a model is excessively complex, such as having too many parameters relative to the number of observations. A model that has been over fit will generally have poor predictive performance, as it can exaggerate minor fluctuations in the data.

A learning algorithm is trained using some set of training samples. If the learning algorithm has the capacity to overbite the training samples the performance on the training sample set will improve while the performance on unseen test sample set will decline.

The overfitting phenomenon has three main explanations: - excessively complex models, - multicollinearity, and - high dimensionality.

Model complexity

Complex learners with too many parameters relative to the number of observations may overfit the training dataset.

Multicollinearity

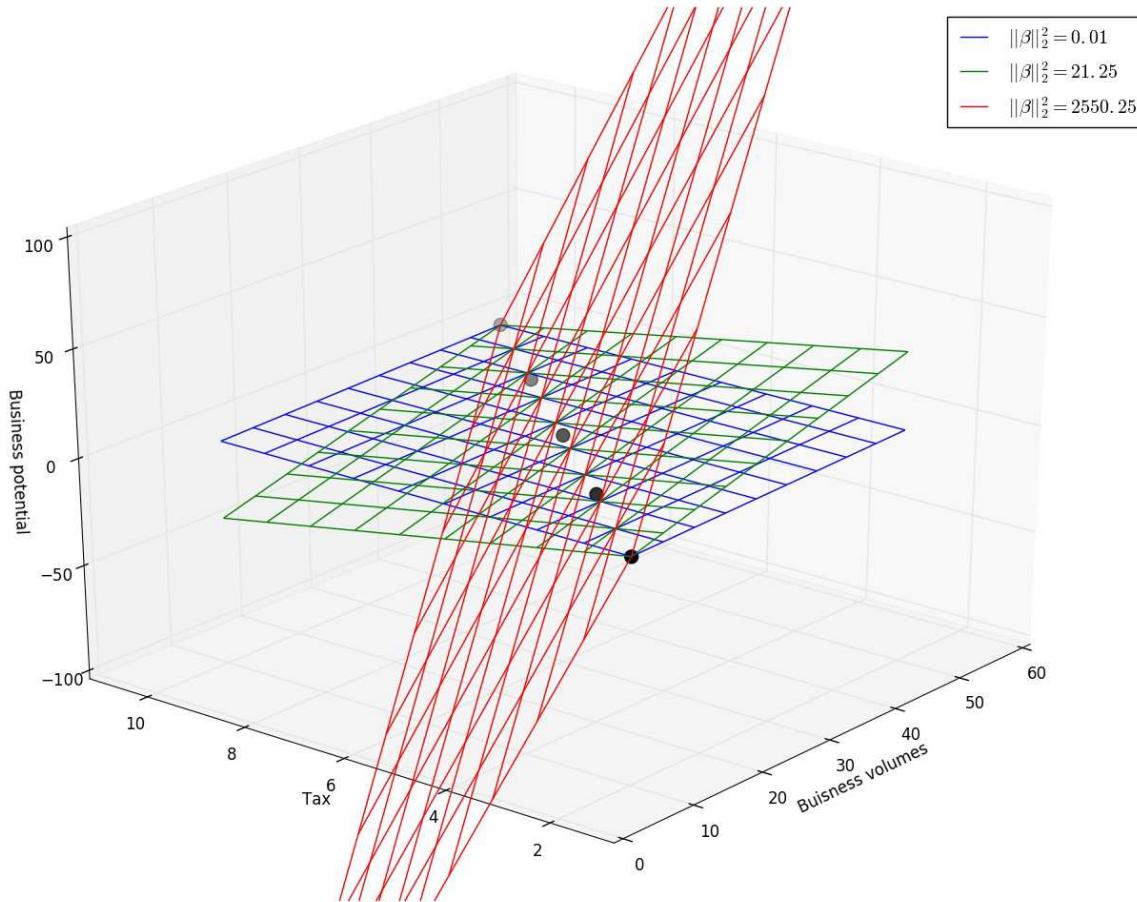
Predictors are highly correlated, meaning that one can be linearly predicted from the others. In this situation the coefficient estimates of the multiple regression may change erratically in response to small changes in the model or the data. Multicollinearity does not reduce the predictive power or reliability of the model as a whole, at least not within the sample data set; it only affects computations regarding individual predictors. That is, a multiple regression model with correlated predictors can indicate how well the entire bundle of predictors predicts the outcome variable, but it may not give valid results about any individual predictor, or about which predictors are redundant with respect to others. In case of perfect multicollinearity the predictor matrix is singular and therefore cannot be inverted. Under these circumstances, for a general linear model $\mathbf{y} = \mathbf{X}\beta + \epsilon$, the ordinary least-squares estimator, $\beta_{OLS} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$, does not exist.

An example where correlated predictor may produce an unstable model follows:

```
import numpy as np from mpl_toolkits.mplot3d
import Axes3D
import matplotlib.pyplot as plt
bv = np.array([10, 20, 30, 40, 50]) # business volume tax = .2 * bv      # Tax
bp = .1 * bv + np.array([-1, 1, 1, -2, 1]) # business potential
X = np.column_stack([bv, tax])
beta_star = np.array([.1, 0]) # true solution
"Since tax and bv are correlated, there is an infinite number of linear combinations leading to the same prediction. "
# 10 times the bv then subtract it 9 times using the tax variable:
beta_medium = np.array([.1 * 10, -.1 * 9 * (1/2)])
# 100 times the bv then subtract it 99 times using the tax variable:
beta_large = np.array([.1 * 100, -.1 * 99 * (1/2)])
# Check that all model lead to the same result

assert np.all(np.dot(X, beta_star) == np.dot(X, beta_medium))
assert np.all(np.dot(X, beta_star) == np.dot(X, beta_large))
```

Multicollinearity between the predictors: business volumes and tax produces unstable models with arbitrary large coefficients.



Dealing with multicollinearity:

- Regularisation by e.g. ℓ_2 shrinkage: Introduce a bias in the solution by making $(X^T X)^{-1}$ non-singular. See ℓ_2 shrinkage.
- Feature selection: select a small number of features. See: Isabelle Guyon and André Elisseeff *An introduction to variable and feature selection* The Journal of Machine Learning Research, 2003.
- Feature selection: select a small number of features using ℓ_1 shrinkage.
- Extract few independent (uncorrelated) features using e.g. principal components analysis (PCA), partial least squares regression (PLS-R) or regression methods that cut the number of predictors to a smaller set of uncorrelated components.

High dimensionality

High dimensions means a large number of input features. Linear predictor associate one parameter to each input feature, so a high-dimensional situation (P , number of features, is large) with a relatively small number of samples N (so-called large P small N situation) generally lead to an overfit of the training data. Thus it is generally a bad idea to add many input features into the learner. This phenomenon is called the curse of dimensionality.

One of the most important criteria to use when choosing a learning algorithm is based on the relative size of P and N .

- Remember that the “covariance” matrix $X^T X$ used in the linear model is a $P \times P$ matrix of rank $\min(N, P)$. Thus if $P > N$ the equation system is over parameterized and admit an infinity of solutions that might be specific to the learning dataset. See also ill-conditioned or singular matrices.

- The sampling density of N samples in an P -dimensional space is proportional to $N^{1/P}$. Thus a high dimensional space becomes very sparse, leading to poor estimations of samples densities.
- Another consequence of the sparse sampling in high dimensions is that all sample points are close to an edge of the sample. Consider N data points uniformly distributed in a P -dimensional unit ball centered at the origin. Suppose we consider a nearest-neighbour estimate at the origin. The median distance from the origin to the closest data point is given by the expression

$$d(P, N) = \left(1 - \frac{1}{2}^N\right)^{1/P}$$

A more complicated expression exists for the mean distance to the closest point. For $N = 500$, $P = 10$, $d(P, N) \approx 0.52$, more than halfway to the boundary. Hence most data points are closer to the boundary of the sample space than to any other data point. The reason that this presents a problem is that prediction is much more difficult near the edges of the training sample. One must extrapolate from neighbouring sample points rather than interpolate between them. (Source: T Hastie, R Tibshirani, J Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second Edition, 2009.)

- Structural risk minimization provides a theoretical background of this phenomenon. (See VC dimension.)
- See also bias variance trade-off.

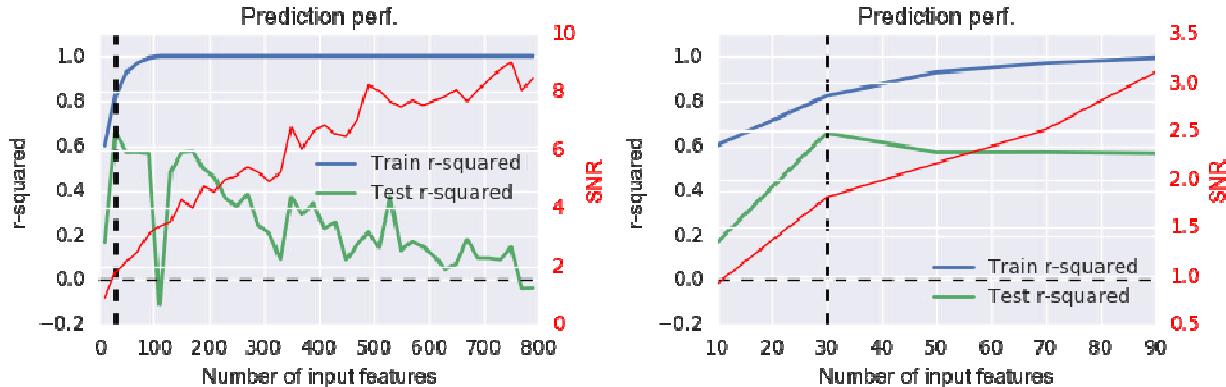
```
import seaborn # nicer plots
def fit_on_increasing_size(model):
    n_samples = 100
    n_features_ = np.arange(10, 800, 20)
    r2_train, r2_test, snr = [], [], []
    for n_features in n_features_:
        # Sample the dataset (* 2 nb of samples)
        n_features_info = int(n_features/10)
        np.random.seed(42) # Make reproducible
        X = np.random.randn(n_samples * 2, n_features)
        beta = np.zeros(n_features)
        beta[:n_features_info] = 1
        Xbeta = np.dot(X, beta)
        eps = np.random.randn(n_samples * 2)
        y = Xbeta + eps
        # Split the dataset into train and test sample
        Xtrain, Xtest = X[:n_samples, :], X[n_samples:, :], ytrain,
        ytest = y[:n_samples], y[n_samples:]
        # fit/predict
        lr = model.fit(Xtrain, ytrain)
        y_pred_train = lr.predict(Xtrain)
        y_pred_test = lr.predict(Xtest)
        snr.append(Xbeta.std() / eps.std())
        r2_train.append(metrics.r2_score(ytrain, y_pred_train))
        r2_test.append(metrics.r2_score(ytest, y_pred_test))
    return n_features_, np.array(r2_train), np.array(r2_test), np.array(snr)
def plot_r2_snr(n_features_, r2_train, r2_test, xvline, snr, ax):
    """
    Two scales plot. Left y-axis: train test r-squared. Right y-axis SNR.
    """
    ax.set_xscale('log')
    ax.set_xlim(10, 800)
    ax.set_xticks([10, 100, 1000])
    ax.set_xticklabels(['10', '100', '1000'])
    ax.set_xlabel('Number of features')
    ax.set_ylabel('R-squared')
    ax.set_yticks([0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
    ax.set_ylabel('Train Test R-squared')
    ax2 = ax.twinx()
    ax2.set_yticks(snr)
    ax2.set_ylabel('SNR')
    ax2.set_ylim(0, 1000)
    ax2.set_xscale('log')
    ax2.set_xlim(10, 800)
    ax2.set_xticks([10, 100, 1000])
    ax2.set_xticklabels(['10', '100', '1000'])
    ax2.set_xlabel('Number of features')
    ax2.set_ylabel('SNR')
    ax2.set_yticks([0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000])
    ax2.set_ylabel('SNR')
    ax2.set_title('Number of features vs R-squared and SNR')
    ax2.axhline(xvline, color='red', linestyle='--')
    ax2.axvline(xvline, color='red', linestyle='--')
    ax2.axvline(100, color='blue', linestyle='--')
    ax2.axvline(1000, color='blue', linestyle='--')
    ax2.axhline(100, color='blue', linestyle='--')
    ax2.axhline(1000, color='blue', linestyle='--')
    ax2.axvline(10000, color='green', linestyle='--')
    ax2.axhline(10000, color='green', linestyle='--')
    ax2.axvline(100000, color='green', linestyle='--')
    ax2.axhline(100000, color='green', linestyle='--')
```

Two scales plot. Left y-axis: train test r-squared. Right y-axis SNR. " "

```

ax.plot(n_features_, r2_train, label="Train r-squared", linewidth=2)
ax.plot(n_features_, r2_test, label="Test r-squared", linewidth=2)
ax.axvline(x=xvline, linewidth=2, color='k', ls='--') ax.axhline(y=0, linewidth=1, color='k', ls='--')
ax.set_xlim(-0.2, 1.1)
ax.set_xlabel("Number of input features")
ax.set_ylabel("r-squared") ax.legend(loc='best')
ax.set_title("Prediction perf.")
ax_right = ax.twinx()
ax_right.plot(n_features_, snr, 'r-', label="SNR", linewidth=1)
ax_right.set_ylabel("SNR", color='r')
for tl in ax_right.get_yticklabels():
    tl.set_color('r')
# Model = linear regression mod =
lm.LinearRegression()
# Fit models on dataset n_features, r2_train, r2_test, snr =
fit_on_increasing_size(model=mod)
argmax = n_features_[np.argmax(r2_test)]
# plot fig, axis = plt.subplots(1, 2, figsize=(9, 3))
# Left pane: all features plot_r2_snr(n_features, r2_train, r2_test, argmax, snr, axis[0])
# Right pane: Zoom on 100 first features plot_r2_snr(n_features[n_features <= 100],
r2_train[n_features <= 100], r2_test[n_features <= 100], argmax, snr[n_features <= 100],
axis[1])
plt.tight_layout()

```



Exercises

Study the code above and:

- Describe the datasets: N : nb_samples, P : nb_features.
- What is n_features_info?
- Give the equation of the generative model.
- What is modified by the loop?
- What is the SNR?

Comment the graph above, in terms of training and test performances:

- How does the train and test performances change as a function of x ?
- Is it the expected results when compared to the SNR?
- What can you conclude?

Ridge regression (regularization)

Overfitting generally leads to excessively complex weight vectors, accounting for noise or spurious correlations within predictors. To avoid this phenomenon the learning should constrain the solution in order to fit a global pattern. This constraint will reduce (bias) the capacity of the learning algorithm. Adding such a penalty will force the coefficients to be small, i.e. to shrink them toward zeros.

Therefore the loss function $\square(\beta)$ (generally the SSE) is combined with a penalty function $\Omega(\beta)$ leading to the general form:

$$\text{Penalized}(\beta) = \square(\beta) + \lambda\Omega(\beta)$$

The respective contribution of the loss and the penalty is controlled by the regularization parameter λ .

Ridge regression impose a ℓ_2 penalty on the coefficients, i.e. it penalizes with the Euclidian norm of the coefficients while minimizing SSE. The objective function becomes:

$$\text{Ridge}(\beta) = \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_2^2.$$

The β that minimises $F_{\text{Ridge}}(\beta)$ can be found by the following derivation:

$$\nabla_\beta \text{Ridge}(\beta) = 0 \quad (10.6)$$

$$\nabla_\beta (\frac{1}{N}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\beta^T\beta) = 0 \quad (10.7)$$

$$\nabla_\beta (\frac{1}{N}(\mathbf{y}^T\mathbf{y} - 2\beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta + \lambda\beta^T\beta)) = 0 \quad (10.8)$$

$$-2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\beta + 2\lambda\beta = 0 \quad (10.9)$$

$$-\mathbf{X}^T\mathbf{y} + (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\beta = 0 \quad (10.10)$$

$$\begin{aligned} (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\beta &= \mathbf{X}^T\mathbf{y} \\ \beta &= (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \end{aligned}$$

- The solution adds a positive constant to the diagonal of $\mathbf{X}^T\mathbf{X}$ before inversion. This makes the problem nonsingular, even if $\mathbf{X}^T\mathbf{X}$ is not of full rank, and was the main motivation behind ridge regression.
- Increasing λ shrinks the β coefficients toward 0.
- This approach penalizes the objective function by the Euclidian (`:math:'ell_2'`) norm of the coefficients such that solutions with large coefficients become unattractive.

The ridge penalty shrinks the coefficients toward zero. The figure illustrates: the OLS solution on the left. The ℓ_1 and ℓ_2 penalties in the middle pane. The penalized OLS in the right pane. The right pane shows how the penalties shrink the coefficients toward zero. The black points are the minimum found in each case, and the white points represents the true solution used to generate the data.

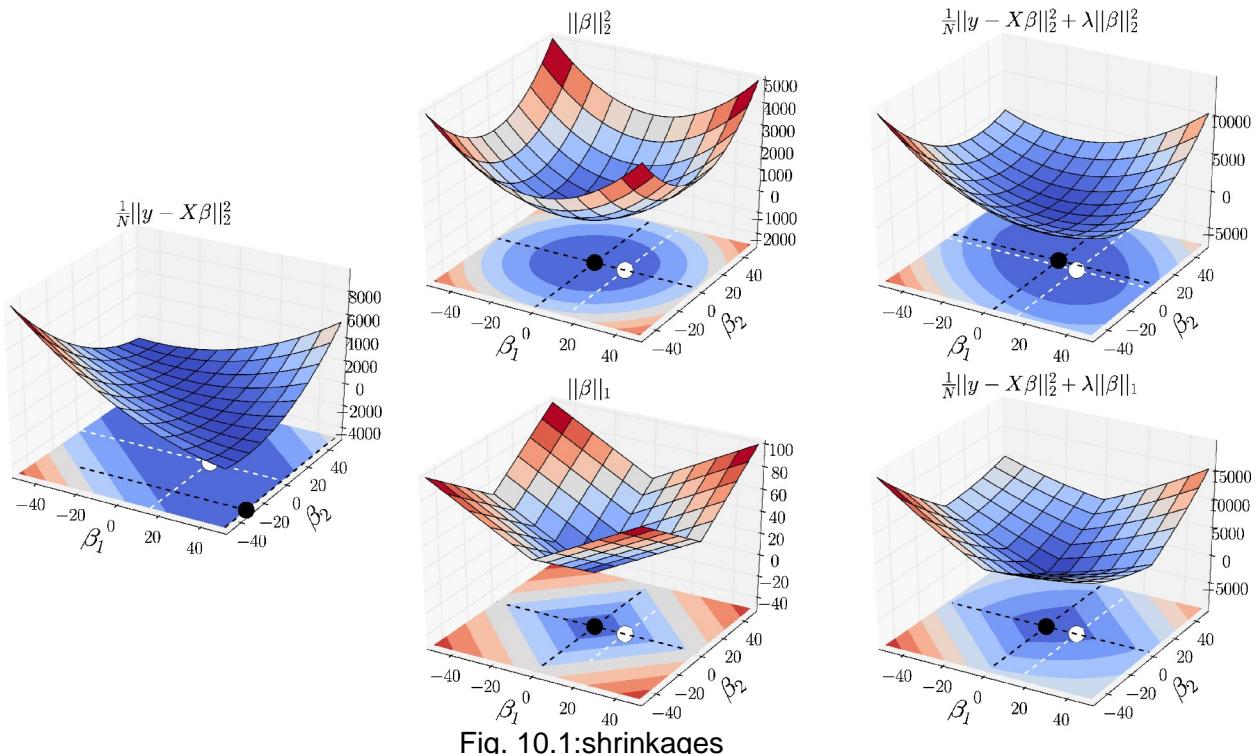
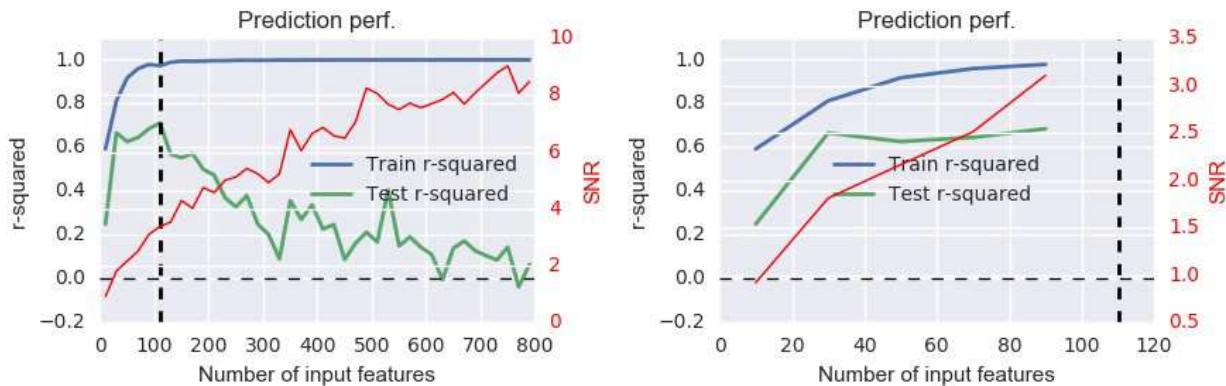


Fig. 10.1:shrinkages

```

import matplotlib.pyplot as plt
import numpy as np
import sklearn.linear_model as lm
# lambda is alpha! mod =
lm.Ridge(alpha=10)
# Fit models on dataset n_features, r2_train, r2_test, snr =
fit_on_increasing_size(model=mod)
argmax = n_features[np.argmax(r2_test)]
# plot fig, axis = plt.subplots(1, 2, figsize=(9, 3))
# Left pane: all features
plot_r2_snr(n_features, r2_train, r2_test, argmax, snr, axis[0])
# Right pane: Zoom on 100 first features plot_r2_snr(n_features[n_features <= 100],
r2_train[n_features <= 100], r2_test[n_features <= 100], argmax,
snr[n_features <= 100], axis[1])
plt.tight_layout()

```



Exercice

What benefit has been obtained by using ℓ_2 regularization?

Lasso regression (ℓ_1 -regularization)

Lasso regression penalizes the coefficients by the ℓ_1 norm. This constraint will reduce (bias) the capacity of the learning algorithm. To add such a penalty forces the coefficients to be small, i.e. it shrinks them toward zero. The objective function to minimize becomes:

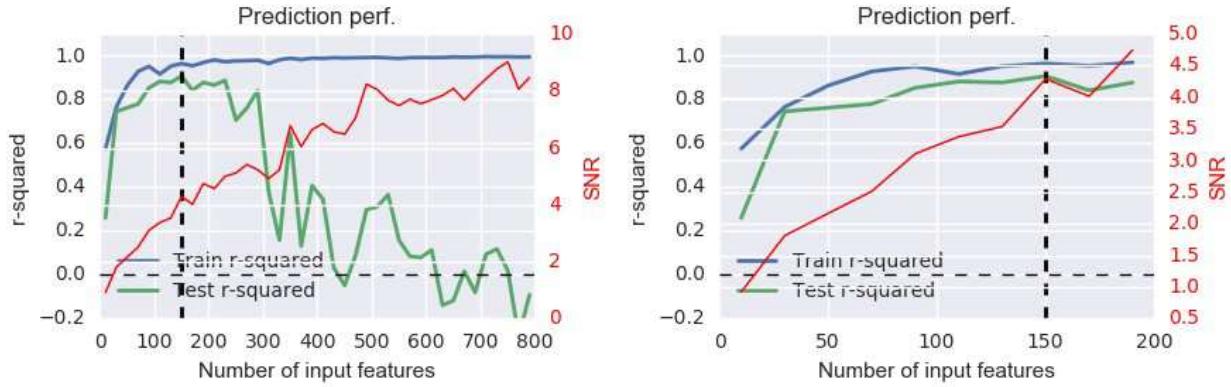
$$\text{Lasso}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\beta}\|_1. \quad (10.13)$$

This penalty forces some coefficients to be exactly zero, providing a feature selection property.

```

import matplotlib.pyplot as plt
import numpy as np
import sklearn.linear_model as lm
# lambda is alpha ! mod =
lm.Lasso(alpha=.1)
# Fit models on dataset
n_features, r2_train, r2_test, snr = fit_on_increasing_size(model=mod)
argmax = n_features[np.argmax(r2_test)]
# plot
fig, axis = plt.subplots(1, 2, figsize=(9, 3))
# Left pane: all features plot_r2_snr(n_features, r2_train, r2_test, argmax, snr, axis[0])
# Right pane: Zoom on 200 first features plot_r2_snr(n_features[n_features <= 200],
r2_train[n_features <= 200], r2_test[n_features <= 200], argmax, snr[n_features <= 200],
axis[1])
plt.tight_layout()

```



Sparsity of the ℓ_1 norm

Occam's razor

Occam's razor (also written as Ockham's razor, and *lex parsimoniae* in Latin, which means law of parsimony) is a problem solving principle attributed to William of Ockham (1287-1347), who was an English Franciscan friar and scholastic philosopher and theologian. The principle can be interpreted as stating that among competing hypotheses, the one with the fewest assumptions should be selected.

Principle of parsimony

The simplest of two competing theories is to be preferred. Definition of parsimony: Economy of explanation in conformity with Occam's razor.

Among possible models with similar loss, choose the simplest one:

- Choose the model with the smallest coefficient vector, ie. smallest ℓ_2 ($\|\beta\|_2$) or ℓ_1 ($\|\beta\|_1$) norm of β , i.e. ℓ_2 or ℓ_1 penalty. See also bias-variance tradeoff.
- Choose the model that uses the smallest number of predictors. In other words, choose the model that has many predictors with zero weights. Two approaches are available to obtain this: (i) Perform a feature selection as a preprocessing prior to applying the learning algorithm, or (ii) embed the feature selection procedure within the learning process.

Sparsity-induced penalty or embedded feature selection with the ℓ_1 penalty

The penalty based on the ℓ_1 norm promotes sparsity (scattered, or not dense): it forces many coefficients to be exactly zero. This also makes the coefficient vector scattered.

The figure below illustrates the OLS loss under a constraint acting on the ℓ_1 norm of the coefficient vector. I.e., it illustrates the following optimization problem:

$$\begin{aligned} & \text{minimize} \\ & \|y - X\beta\|_{2\beta}^2 \text{ subject} \\ & \text{to } \|\beta\|_1 \leq 1. \end{aligned}$$

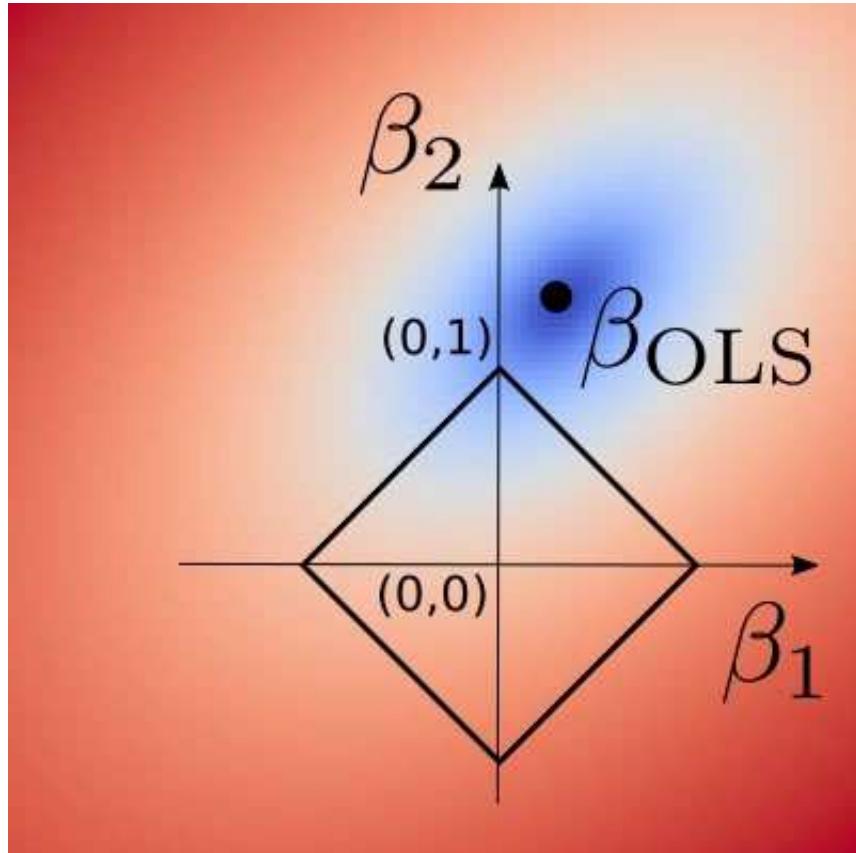


Fig. 10.2: Sparsity of L1 norm

Optimization issues

Section to be completed

- No more closed-form solution.
- Convex but not differentiable.
- Requires specific optimization algorithms, such as the fast iterative shrinkage-thresholding algorithm (FISTA): Amir Beck and Marc Teboulle, *A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems* SIAM J. Imaging Sci., 2009.

Elastic-net regression (ℓ_2 - ℓ_1 -regularization)

The Elastic-net estimator combines the ℓ_1 and ℓ_2 penalties, and results in the problem to

$$\text{Enet}(\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}^T \boldsymbol{\beta}\|_2^2 + \alpha (\rho \|\boldsymbol{\beta}\|_1 + (1 - \rho) \|\boldsymbol{\beta}\|_2^2), \quad (10.14)$$

where α acts as a global penalty and ρ as an ℓ_1/ℓ_2 ratio.

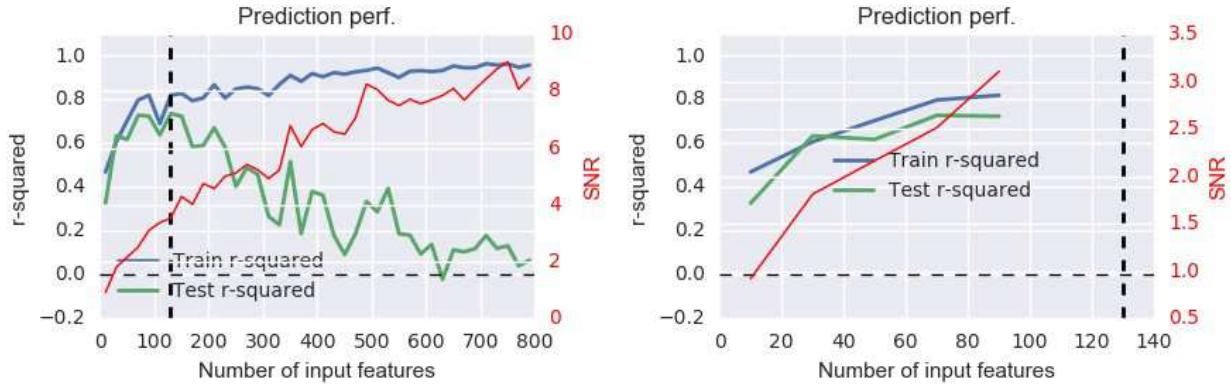
Rationale

- If there are groups of highly correlated variables, Lasso tends to arbitrarily select only one from each group. These models are difficult to interpret because covariates that are strongly associated with the outcome are not included in the predictive model. Conversely, the elastic net encourages a grouping effect, where strongly correlated predictors tend to be in or out of the model together.
- Studies on real world data and simulation studies show that the elastic net often outperforms the lasso, while enjoying a similar sparsity of representation.

```

import matplotlib.pyplot as plt
import numpy as np
import sklearn.linear_model as lm
mod = lm.ElasticNet(alpha=.5, l1_ratio=.5)
# Fit models on dataset
n_features, r2_train, r2_test, snr = fit_on_increasing_size(model=mod)
argmax = n_features[np.argmax(r2_test)]
# plot fig, axis = plt.subplots(1, 2, figsize=(9, 3))
# Left pane: all features plot_r2_snr(n_features, r2_train, r2_test, argmax, snr, axis[0])
# Right pane: Zoom on 100 first features plot_r2_snr(n_features[n_features <= 100],
r2_train[n_features <= 100], r2_test[n_features <= 100], argmax, snr[n_features <= 100],
axis[1])
plt.tight_layout()

```



CHAPTER ELEVEN

LINEAR CLASSIFICATION

A linear classifier achieves its classification decision of \hat{y}_i , based on the value of a linear combination of the input features of a given sample x_i , such that

$$\hat{y}_i = f(w \cdot x_i),$$

where $w \cdot x_i := w^T x_i$ is the dot product between w and x_i .

Fisher's linear discriminant with equal class covariance

This geometric method does not make any probabilistic assumptions, instead it relies on distances. It looks for the linear projection of the data points onto a vector, w , that maximizes the between/within variance ratio, denoted $F(w)$. Under a few assumptions, it will provide the same results as linear discriminant analysis (LDA), explained below.

Suppose two classes of observations, C_0 and C_1 , have means μ_0 and μ_1 and the same total within-class scatter ("covariance") matrix,

$$S_w = \sum_{i \in C_0} (x_i - \mu_0)(x_i - \mu_0)^T + \sum_{j \in C_1} (x_j - \mu_1)(x_j - \mu_1)^T \quad (11.1)$$

$$= \mathbf{X}_c^T \mathbf{X}_c \quad (11.2)$$

where X_c is the $(N \times P)$ matrix of data centered on their respective means:

$$X_c = \begin{bmatrix} \mathbf{X}_0 - \mu_0 \\ \mathbf{X}_1 - \mu_1 \end{bmatrix}$$

where X_0 and X_1 are the $(N_0 \times P)$ and $(N_1 \times P)$ matrices of samples of classes C_0 and C_1 .

Let S_B being the scatter "between-class" matrix, given by

$$S_B = (\mu_1 - \mu_0)(\mu_1 - \mu_0)^T.$$

The linear combination of features $w^T x$ have means $w^T \mu_i$ for $i = 0, 1$, and variance $w^T X_c^T X_c w$. Fisher defined the separation between these two distributions to be the ratio of the variance between the classes to the variance within the classes:

$$F_{\text{Fisher}}(w) = \frac{\sigma_{\text{between}}^2}{2}$$

The Fisher most discriminant projection

In the two-class case, the maximum separation occurs by a projection on the $(\mu_1 - \mu_0)$ using the Mahalanobis metric

S_B^{-1} , so that

$w \propto S_w^{-1}(\mu_1 - \mu_0)$.

Demonstration

Differentiating $F_{\text{Fisher}}(w)$ with respect to w gives

$$\nabla_w F_{\text{Fisher}}(w) = 0$$

$$\nabla_w \left(\frac{w^T S_B w}{w^T S_w w} \right) = 0$$

Since we do not care about the magnitude of w , only its direction, we replaced the scalar factor $(w^T S_B w)/(w^T S_w w)$ by λ .

In the multiple-class case, the solutions w are determined by the eigenvectors of $S_w^{-1}S_B$ that correspond to the $K - 1$ largest eigenvalues.

However, in the two-class case (in which $S_B = (\mu_1 - \mu_0)(\mu_1 - \mu_0)^T$)

it is easy to show that $w = S_w^{-1}(\mu_1 - \mu_0)$ is the unique eigenvector of $S_w^{-1}S_B$:

$$S_w^{-1}(\mu_1 - \mu_0)(\mu_1 - \mu_0)^T w = \lambda w$$

$$S_w^{-1}(\mu_1 - \mu_0)(\mu_1 - \mu_0)^T S_w^{-1}(\mu_1 - \mu_0) = \lambda S_w^{-1}(\mu_1 - \mu_0),$$

where here

$$\lambda = (\mu_1 - \mu_0)^T S_w^{-1}(\mu_1 - \mu_0). \text{ Which leads to the result}$$

$$w \propto S_w^{-1}(\mu_1 - \mu_0).$$

The separating hyperplane

The separating hyperplane is a $P - 1$ -dimensional hyper surface, orthogonal to the projection vector, w . There is no single best way to find the origin of the plane along w , or equivalently the classification threshold that determines whether a point should be classified as belonging to C_0 or to C_1 . However, if the projected points have roughly the same distribution, then the threshold can be chosen as the hyperplane exactly between the projections of the two means, i.e. as

$$T = \mathbf{w} \cdot \frac{1}{2}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$$

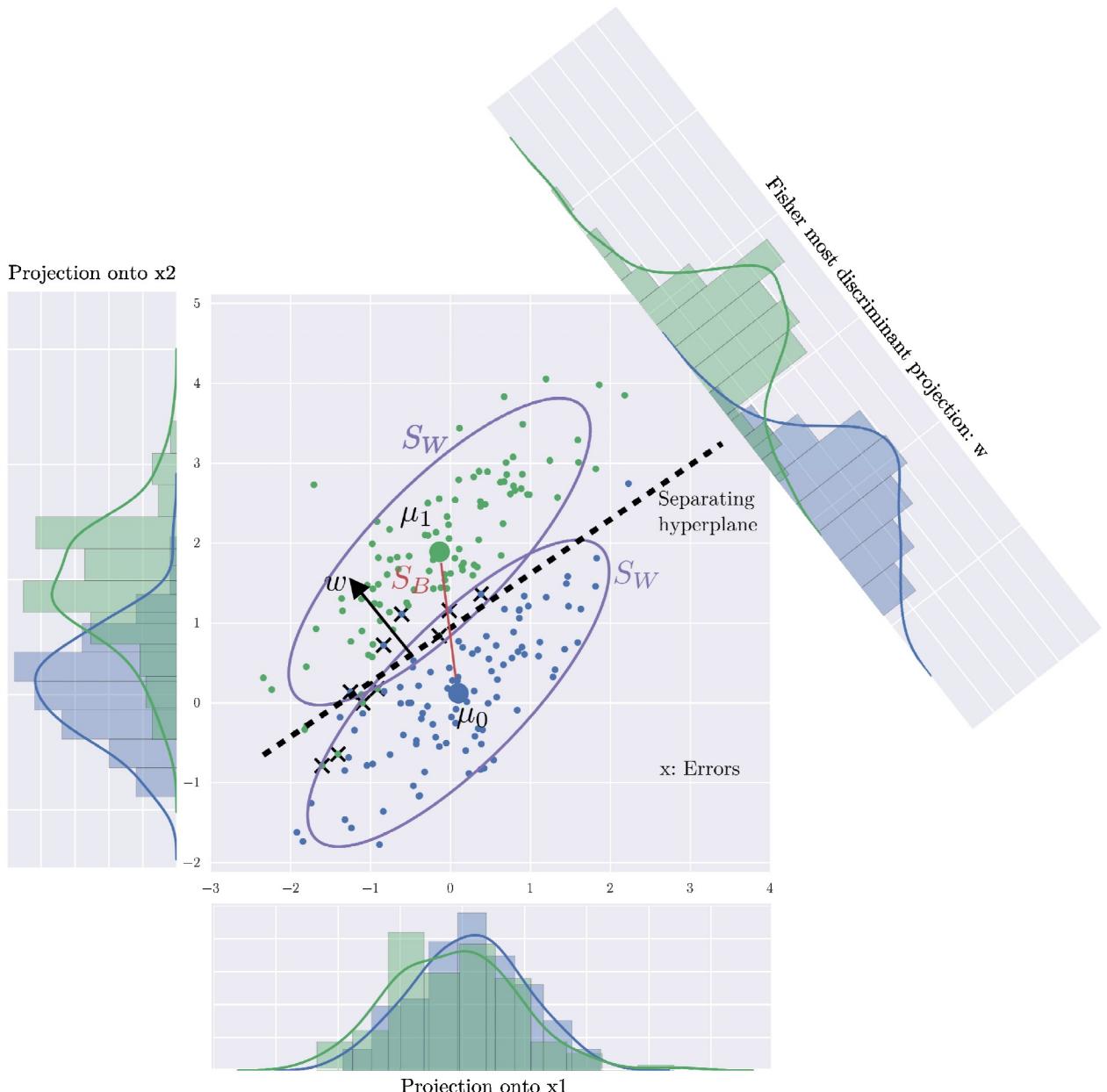


Fig. 11.1: The Fisher most discriminant projection

Exercise

Write a class `FisherLinearDiscriminant` that implements the Fisher's linear discriminant analysis. This class must be compliant with the scikit-learn API by providing two methods: `fit(X,y)` which fits the model and returns the object itself; `predict(X)` which returns a vector of the predicted values. Apply the object on the dataset presented for the LDA.

Linear discriminant analysis (LDA)

Linear discriminant analysis (LDA) is a probabilistic generalization of Fisher's linear discriminant. It uses Bayes' rule to fix the threshold based on prior probabilities of classes.

1. First compute the class-conditional distributions of x given class C_k : $p(x|C_k) = p(x|\mu_k, S_W)$. Where $p(x|\mu_k, S_W)$ is the multivariate Gaussian distribution defined over a P-dimensional vector x of continuous variables.
2. Estimate the prior probabilities of class k , $p(C_k) = N_k/N$.

3. Compute posterior probabilities (ie. the probability of a each class given a sample) combining conditional with priors using Bayes' rule:

$$p(C_k|x) = \frac{p(C_k)p(\mathbf{x}|C_k)}{p(\mathbf{x})}$$

Where $p(x)$ is the marginal distribution obtained by summing of classes: As usual, the denominator in Bayes' theorem can be found in terms of the quantities appearing in the numerator, because

$$p(x) = \sum_k p(\mathbf{x}|C_k)p(C_k)$$

4. Classify x using the Maximum-a-Posterior probability: $C_k = \text{argmax}_{C_k} p(C_k|x)$

LDA is a generative model since the class-conditional distributions can be used to generate samples of each classes.

LDA is useful to deal with imbalanced group sizes (eg.: $N_1 \gg N_0$) since priors probabilities can be used to explicitly re-balance the classification by setting $p(C_0) = p(C_1) = 1/2$ or whatever seems relevant.

LDA can be generalised to the multicasts case with $K > 2$.

With $N_1 = N_0$, LDA lead to the same solution than Fisher's linear discriminant.

Exercise

How many parameters are required to estimate to perform a LDA ?

```
%matplotlib inline
import numpy as np
from sklearn.discriminant_analysis
import LinearDiscriminantAnalysis as LDA
# Dataset
n_samples, n_features = 100, 2
mean0, mean1 = np.array([0, 0]), np.array([0, 2])
Cov = np.array([[1, .8], [.8, 1]])
np.random.seed(42)

X0 = np.random.multivariate_normal(mean0, Cov, n_samples)
X1 = np.random.multivariate_normal(mean1, Cov, n_samples)
X = np.vstack([X0, X1])
y = np.array([0] * X0.shape[0] + [1] * X1.shape[0])
# LDA with scikit-learn
lda = LDA()
proj = lda.fit(X, y).transform(X)
y_pred_lda = lda.predict(X)
errors = y_pred_lda != y
print("Nb errors=%i, error rate=%f" % (errors.sum(), errors.sum() / len(y_pred_lda)))
```

Nb errors=10, error rate=0.05

Logistic regression

Logistic regression is called a generalized linear models. ie.: it is a linear model with a link function that maps the output of linear multiple regression to a the posterior probability of each class $p(C_k|x) \in [0,1]$ using the logistic sigmoid function:

$$p(C_k|w, x_i) = \frac{1}{1 + \exp(-w \cdot x_i)}$$

Logistic regression seeks to minimizes the likelihood L as Loss function □:

$$\min L(w) = \prod_i^N p(C_k|w, x_i)$$

Partially, the Loss function □ is the log-likelihood:

$$\min \mathcal{L}(w) = \log L(w) = \sum_i^N \log p(C_k|w, x_i)$$

In the two-class case the algorithms simplify considerably by coding the two-classes (C_0 and C_1) via a 0/1 response y_i . Indeed, since $p(C_0|w, x_i) = 1 - p(C_1|w, x_i)$, the log-likelihood can be re-written:

$$\log L(\mathbf{w}) = \sum_i^N \{y_i \log p(C_1|\mathbf{w}, \mathbf{x}_i) + (1 - y_i) \log(1 - p(C_1|\mathbf{w}, \mathbf{x}_i))\}$$

$$\log L(\mathbf{w}) = \sum_i^N \{y_i \mathbf{w} \cdot \mathbf{x}_i - \log(1 + \exp^{\mathbf{w} \cdot \mathbf{x}_i})\}$$

Logistic regression is a discriminative model since it focuses only on the posterior probability of each class $p(C_k|x)$. It only requires to estimate the P weight of the w vector. Thus it should be favoured over LDA with many input features. In small dimension and balanced situations it would provide similar predictions than LDA.

However imbalanced group sizes cannot be explicitly controlled. It can be managed using a reweighting of the input samples.

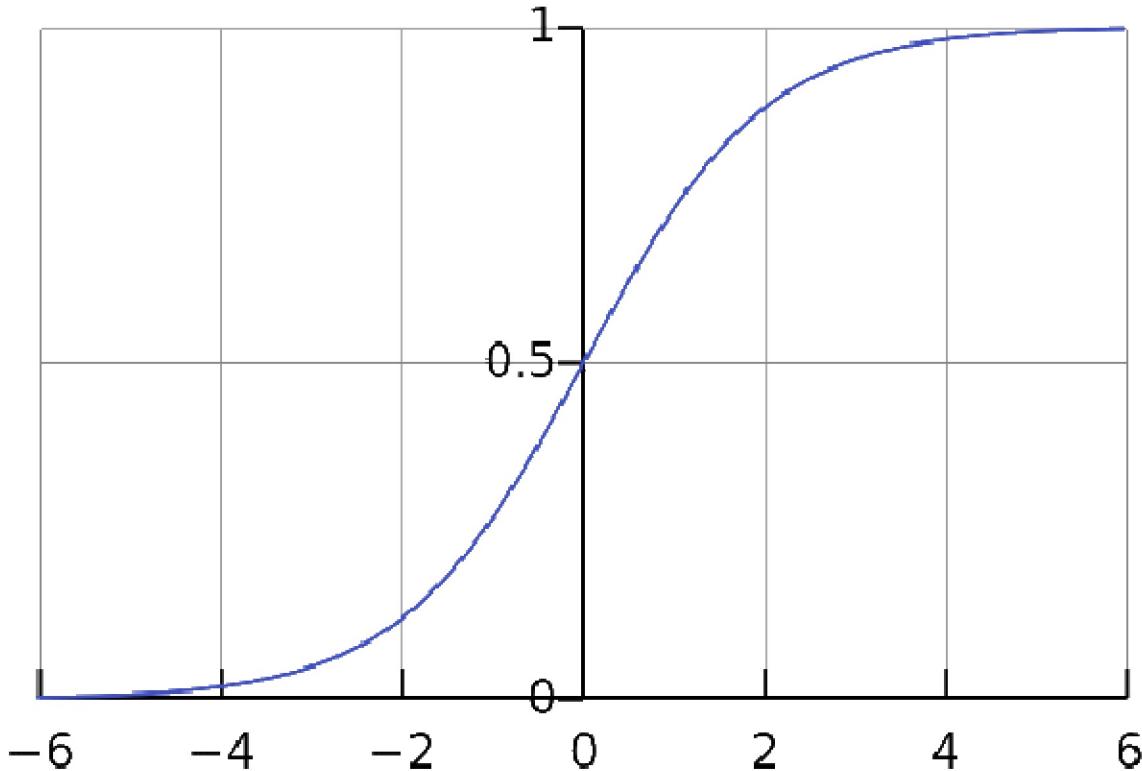


Fig. 11.2: logistic sigmoid function

```

from sklearn import linear_model
logreg = linear_model.LogisticRegression(C=1e8)
# This class implements regularized logistic regression. C is the Inverse of regularization strength.
# Large value => no regularization.
logreg.fit(X, y)
y_pred_logreg = logreg.predict(X)
errors = y_pred_logreg != y
print("Nb errors=%i, error rate=%f" % (errors.sum(), errors.sum() / len(y_pred_logreg)))
print(logreg.coef_)

```

Nb errors=10, error rate=0.05
[[-5.15162649 5.57299286]]

Exercise

Explore the Logistic Regression parameters and proposes a solution in cases of highly imbalanced training dataset $N_1 \gg N_0$ when we know that in reality both classes have the same probability $p(C_1) = p(C_0)$.

Overfitting

VC dimension (for Vapnik–Chervonenkis dimension) is a measure of the capacity (complexity, expressive power, richness, or flexibility) of a statistical classification algorithm, defined as the cardinality of the largest set of points that the algorithm can shatter.

Theorem: Linear classifier in R^P have VC dimension of $P+1$. Hence in dimension two ($P=2$) any random partition of 3 points can be learned.

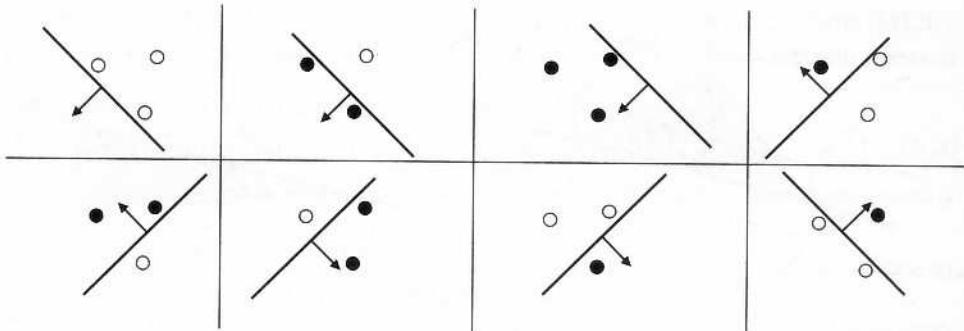


Fig. 11.3: In 2D we can shatter any three non-collinear points

Ridge Fisher's linear classification (L2-regularization)

When the matrix S_w is not full rank or $P \gg N$, the Fisher most discriminant projection estimate of the is not unique. This can be solved using a biased version of S_w :

$$S_{w\text{ Ridge}} = S_w + \lambda I$$

where I is the $P \times P$ identity matrix. This leads to the regularized (ridge) estimator of the Fisher's linear discriminant analysis:

$$W_{\text{Ridge}} \propto (S_w + \lambda I)^{-1}(\mu_1 - \mu_0)$$

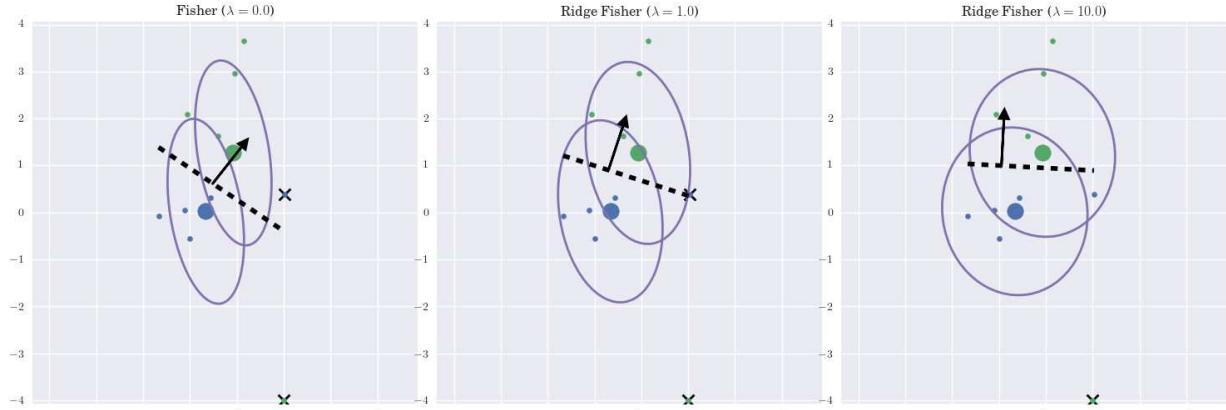


Fig. 11.4: The Ridge Fisher most discriminant projection

Increasing λ will:

- Shrinks the coefficients toward zero.
- The covariance will converge toward the diagonal matrix, reducing the contribution of the pairwise covariances.

Ridge logistic regression (L2-regularization)

The objective function to be minimized is now the combination of the logistic loss $\log L(w)$ with a penalty of the L2 norm of the weights vector. In the two-class case, using the 0/1 coding we obtain:

$$\min \text{Logistic ridge}(w) = \log L(w) + \lambda \|w\|^2 \quad (11.11)$$

$$\sum_i^N \{y_i w^T x_i - \log(1 + \exp^{w^T x_i})\} + \lambda \|w\|^2 = \quad (11.12)$$

```
# Dataset # Build a classification task using 3 informative features
```

```
from sklearn import datasets
X, y = datasets.make_classification(n_samples=100,
```

<code>n_features=20, n_informative=3, n_redundant=0, n_repeated=0, n_classes=2, random_state=0, shuffle=False)</code>

```
from sklearn import linear_model
lr = linear_model.LogisticRegression(C=1)
# This class implements regularized logistic regression. C is the Inverse of regularization strength.
# Large value => no regularization.
lr.fit(X, y)
y_pred_lr = lr.predict(X)
errors = y_pred_lr != y
print("Nb errors=%i, error rate=% .2f % (errors.sum(), errors.sum() / len(y))) print(lr.coef_)
```

```
Nb errors=26, error rate=0.26
[[ -0.12061092 0.7357655 -0.01842318 -0.10835785 0.25328562 0.4221318
  0.15152184 0.16522461 0.84404799 0.01962765 -0.15995078 -0.01925974
  -0.02807379 0.42939869 -0.06368702 -0.07922044 0.15529371 0.29963205
  0.54633137 0.03866807]]
```

Lasso logistic regression (L1-regularization)

The objective function to be minimized is now the combination of the logistic loss $\log L(w)$ with a penalty of the L1 norm of the weights vector. In the two-class case, using the 0/1 coding we obtain:

$$\min \text{Logistic Lasso}(w) = \log L(w) + \lambda \|w\|_1 \quad (11.14)$$

$$= \sum_i^N \{y_i w \cdot x_i - \log(1 + \exp^{w \cdot x_i})\} + \lambda \|w\|_1 \quad (11.15)$$

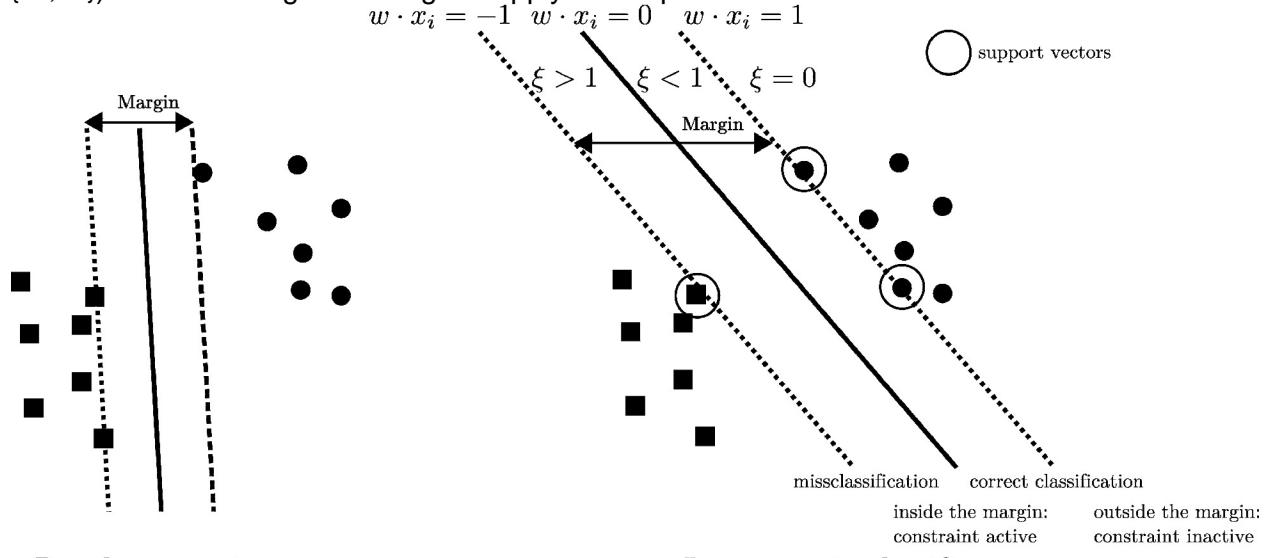
```
from sklearn import linear_model
lrl1 = linear_model.LogisticRegression(penalty='l1')
# This class implements regularized logistic regression. C is the Inverse of regularization strength.
# Large value => no regularization.
lrl1.fit(X, y)
y_pred_lrl1 = lrl1.predict(X)
errors = y_pred_lrl1 != y
print("Nb errors=%i, error rate=%f" % (errors.sum(), errors.sum() / len(y_pred_lrl1)))
print(lrl1.coef_)

Nb errors=27, error rate=0.27 [-
0.11335795 0.68150965 0.          0.19754476 0.36480308
```

Ridge linear Support Vector Machine (L2-regularization)

Support Vector Machine seek for separating hyperplane with maximum margin to enforce robustness against noise. Like logistic regression it is a discriminative method that only focuses of predictions.

Here we present the non separable case of Maximum Margin Classifiers with ± 1 coding (ie.: $y_i \in \{-1, +1\}$). In the next figure the legend apply to samples of "dot" class.



Linear SVM for classification (also called SVM-C or SVC) minimizes:

$$\begin{aligned} \min \quad & \text{Linear SVM}(w) = \text{penalty}(w) + C \text{ Hinge loss}(w) \\ = \|w\|_2^2 + C \sum_i & \xi_i \\ \text{with} \quad & \forall i \quad y_i(w \cdot x_i) \geq 1 - \xi_i \end{aligned}$$

Here we introduced the slack variables: ξ_i , with $\xi_i = 0$ for points that are on or inside the correct margin boundary and $\xi_i = |y_i - (w \cdot x_i)|$ for other points. Thus:

1. If $y_i(w \cdot x_i) \geq 1$ then the point lies outside the margin but on the correct side of the decision boundary. In this case $\xi_i = 0$. The constraint is thus not active for this point. It does not contribute to the prediction.
2. If $1 > y_i(w \cdot x_i) \geq 0$ then the point lies inside the margin and on the correct side of the decision boundary. In this case $0 < \xi_i \leq 1$. The constraint is active for this point. It does contribute to the prediction as a support vector.
3. If $0 < y_i(w \cdot x_i)$ then the point is on the wrong side of the decision boundary (misclassification). In this case $0 < \xi_i > 1$. The constraint is active for this point. It does contribute to the prediction as a support vector.

This loss is called the hinge loss, defined as:

$$\max(0, 1 - y_i(w \cdot x_i))$$

So linear SVM is closed to Ridge logistic regression, using the hinge loss instead of the logistic loss. Both will provide very similar predictions.

```
from sklearn import svm
svmlin = svm.LinearSVC()
# Remark: by default LinearSVC uses squared_hinge as loss
svmlin.fit(X, y)
y_pred_svmlin = svmlin.predict(X)
errors = y_pred_svmlin != y
print("Nb errors=%i, error rate=%." % (errors.sum(), len(y_pred_svmlin)))
print(svmlin.coef_)

Nb errors=26, error rate=0.26
[[-0.05611026 0.31189817 0.00272904 -0.05149528 0.0994013 0.17726815
 0.06520128 0.08921178 0.35339409 0.00601206 -0.06200898 -0.00741242
 -0.02156839 0.1827221 -0.02163261 -0.04060274 0.07204827 0.13083533
 0.23722076 0.00823331]]
```

Lasso linear Support Vector Machine (L1-regularization)

Linear SVM for classification (also called SVM-C or SVC) with L1-regularization

$$\begin{aligned} \min \quad & F_{\text{Lasso linear SVM}}(w) = \lambda \|w\|_1 + C \sum_i \xi_i \\ \text{with} \quad & \forall i \quad y_i(w \cdot x_i) \geq 1 - \xi_i \end{aligned}$$

```
from sklearn import svm
svmlinl1 = svm.LinearSVC(penalty='l1', dual=False)
# Remark: by default LinearSVC uses squared_hinge as loss
svmlinl1.fit(X, y)
y_pred_svmlinl1 = svmlinl1.predict(X)
errors = y_pred_svmlinl1 != y
print("Nb errors=%i, error rate=%." % (errors.sum(), errors.sum() / len(y_pred_svmlinl1)))
print(svmlinl1.coef_)

Nb errors=26, error rate=0.26
[[-0.05333774 0.29934019 0. -0.03541597
 0.05807924 0.07587815 0.34065253 0.
 -0.01312278 0.16866252 -0.01450365 -0.02500341
 0.22485376 0.00473435]]
```

Exercise

Compare predictions of Logistic regression (LR) and their SVM counterparts, ie.: L2 LR vs L2 SVM and L1 LR vs L1 SVM

Lasso linear Support Vector Machine (L1-regularization)

- Compute the correlation between pairs of weights vectors.
- Compare the predictions of two classifiers using their decision function:
 - Give the equation of the decision function for a linear classifier, assuming that there is no intercept.
 - Compute the correlation decision function.
 - Plot the pairwise decision function of the classifiers.
- Conclude on the differences between Linear SVM and logistic regression.

Elastic-net classification (L2-L1-regularization)

The objective function to be minimized is now the combination of the logistic loss $\log L(\mathbf{w})$ or the hinge loss with combination of L1 and L2 penalties. In the two-class case, using the 0/1 coding we obtain:

$$\min \text{Logistic enet}(\mathbf{w}) = \log L(\mathbf{w}) + \alpha (\rho \|\mathbf{w}\|_1 + (1 - \rho) \|\mathbf{w}\|_2^2)$$

$$\min \text{Hinge enet}(\mathbf{w}) = \text{Hinge loss}(\mathbf{w}) + \alpha (\rho \|\mathbf{w}\|_1 + (1 - \rho) \|\mathbf{w}\|_2^2)$$

```
from sklearn import datasets
from sklearn import linear_model as lm
import matplotlib.pyplot as plt
X, y = datasets.make_classification(n_samples=100, n_features=20,
                                    n_informative=3, n_redundant=0,
                                    n_repeated=0, n_classes=2,
                                    random_state=0, shuffle=False)
enetloglike = lm.SGDClassifier(loss="log", penalty="elasticnet", alpha=0.0001, l1_ratio=0.15,
                                class_weight='balanced')
enetloglike.fit(X, y)
enethinge = lm.SGDClassifier(loss="hinge", penalty="elasticnet", alpha=0.0001, l1_ratio=0.15,
                             class_weight='balanced')
enethinge.fit(X, y)
SGDClassifier(alpha=0.0001, average=False, class_weight='balanced', epsilon=0.1, eta0=0.0,
              fit_intercept=True, l1_ratio=0.15, learning_rate='optimal', loss='hinge', n_iter=5, n_jobs=1,
              penalty='elasticnet', power_t=0.5, random_state=None, shuffle=True, verbose=0,
              warm_start=False)
```

Exercise

Compare predictions of Elastic-net Logistic regression (LR) and Hinge-loss Elastic-net

- Compute the correlation between pairs of weights vectors.
- Compare the predictions of two classifiers using their decision function:
 - Compute the correlation decision function.
 - Plot the pairwise decision function of the classifiers.
- Conclude on the differences between the two losses.

Metrics of classification performance evaluation

Metrics for binary classification

source: https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Imagine a study evaluating a new test that screens people for a disease. Each person taking the test either has or does not have the disease. The test outcome can be positive (classifying the

person as having the disease) or negative (classifying the person as not having the disease). The test results for each subject may or may not match the subject's actual status. In that setting:

- True positive (TP): Sick people correctly identified as sick
- False positive (FP): Healthy people incorrectly identified as sick
- True negative (TN): Healthy people correctly identified as healthy
- False negative (FN): Sick people incorrectly identified as healthy
- Accuracy (ACC):

$$ACC = (TP + TN) / (TP + FP + FN + TN)$$
- Sensitivity (SEN) or recall of the positive class or true positive rate (TPR) or hit rate:

$$SEN = TP / P = TP / (TP+FN)$$
- Specificity (SPC) or recall of the negative class or true negative rate:

$$SPC = TN / N = TN / (TN+FP)$$
- Precision or positive predictive value (PPV):

$$PPV = TP / (TP + FP)$$
- Balanced accuracy (bACC): is a useful performance measure is the balanced accuracy which avoids inflated performance estimates on imbalanced datasets (Brodersen, et al. (2010). "The balanced accuracy and its posterior distribution"). It is defined as the arithmetic mean of sensitivity and specificity, or the average accuracy obtained on either class:

$$bACC = 1/2 * (SEN + SPC)$$
- F1 Score (or F-score) which is a weighted average of precision and recall are useful to deal with imbalanced datasets

The four outcomes can be formulated in a 2×2 contingency table or confusion matrix
https://en.wikipedia.org/wiki/Sensitivity_and_specificity

For more precision see: http://scikit-learn.org/stable/modules/model_evaluation.html

```
from sklearn import metrics
y_pred = [0, 1, 0, 0]
y_true = [0, 1, 0, 1]
metrics.accuracy_score(y_true, y_pred)
# The overall precision an recall
metrics.precision_score(y_true, y_pred)
metrics.recall_score(y_true, y_pred)
# Recalls on individual classes: SEN & SPC
recalls = metrics.recall_score(y_true, y_pred, average=None)
recalls[0]
# is the recall of class 0: specificity
recalls[1] # is the recall of class 1: sensitivity
# Balanced accuracy
b_acc = recalls.mean()
# The overall precision an recall on each individual class
p, r, f, s = metrics.precision_recall_fscore_support(y_true, y_pred)
```

Area Under Curve (AUC) of Receiver operating characteristic (ROC)

Some classifier may have found a good discriminative projection w . However if the threshold to decide the final predicted class is poorly adjusted, the performances will highlight an high specificity and a low sensitivity or the contrary.

In this case it is recommended to use the AUC of a ROC analysis which basically provide a measure of overlap of the two classes when points are projected on the discriminative axis. For more detail on ROC and AUC

see: https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

```

from sklearn import metrics
score_pred = np.array([.1, .2, .3, .4, .5, .6, .7, .8])
y_true = np.array([0, 0, 0, 0, 1, 1, 1, 1])
thres = .9
y_pred = (score_pred > thres).astype(int)
print("Predictions:", y_pred)
metrics.accuracy_score(y_true, y_pred)
# The overall precision and recall on each individual class
p, r, f, s = metrics.precision_recall_fscore_support(y_true, y_pred)
print("Recalls:", r)
# 100% of specificity, 0% of sensitivity
# However AUC=1 indicating a perfect separation of the two classes
auc = metrics.roc_auc_score(y_true, score_pred)
print("AUC:", auc)

```

Predictions: [0 0 0 0 0 0 0]

Recalls: [1. 0.]

AUC: 1.0

Imbalanced classes

Learning with discriminative (logistic regression, SVM) methods is generally based on minimizing the misclassification of training samples, which may be unsuitable for imbalanced datasets where the recognition might be biased in favour of the most numerous class. This problem can be addressed with a generative approach, which typically requires more parameters to be determined leading to reduced performances in high dimension.

Dealing with imbalanced class may be addressed by three main ways (see Japkowicz and Stephen (2002) for a review), resampling, reweighting and one class learning.

In sampling strategies, either the minority class is oversampled or majority class is undersampled or some combination of the two is deployed. Under sampling (Zhang and Mani, 2003) the majority class would lead to a poor usage of the left-out samples. Sometime one cannot afford such strategy since we are also facing a small sample size problem even for the majority class. Informed oversampling, which goes beyond a trivial duplication of minority class samples, requires the estimation of class conditional distributions in order to generate synthetic samples. Here generative models are required. An alternative, proposed in (Chawla et al., 2002) generate samples along the line segments joining any/all of the k minority class nearest neighbors. Such procedure blindly generalizes the minority area without regard to the majority class, which may be particularly problematic with high-dimensional and potentially skewed class distribution.

Reweighting, also called cost-sensitive learning, works at an algorithmic level by adjusting the costs of the various classes to counter the class imbalance. Such reweighting can be implemented within SVM (Chang and Lin, 2001) or logistic regression (Friedman et al., 2010) classifiers. Most classes of Scikit learn offer such reweighting possibilities.

The `class_weight` parameter can be positioned into the "balanced" mode which uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as $N/(2N_k)$.

```

import numpy as np from sklearn
import linear_model from sklearn
import datasets from sklearn
import metrics
import matplotlib.pyplot as plt
# dataset
X, y = datasets.make_classification(n_samples=500, n_features=5,
                                    n_informative=2, n_redundant=0,
                                    n_repeated=0, n_classes=2,
                                    random_state=1, shuffle=False)
print("#samples of class %i = %i;" % (lev, np.sum(y == lev)) for lev in np.unique(y))
print('# No Reweighting balanced dataset')
lr_inter = linear_model.LogisticRegression(C=1)
lr_inter.fit(X, y)
p, r, f, s = metrics.precision_recall_fscore_support(y, lr_inter.predict(X))
print("SPC: %.3f; SEN: %.3f" % tuple(r))
print('# => The predictions are balanced in sensitivity and specificity\n')
# Create imbalanced dataset, by subsampling sample of class 0: keep only 10% of
# classe 0's samples and all classe 1's samples.
n0 = int(np.rint(np.sum(y == 0) / 20))
subsample_idx = np.concatenate((np.where(y == 0)[0][:n0], np.where(y == 1)[0]))
Ximb = X[subsample_idx, :]

```

```

yimb = y[subsample_idx]
print("#samples of class %i = %i;" % (lev, np.sum(yimb == lev)))
for lev in np.unique(yimb):
    print('# No Reweighting on imbalanced dataset')
    lr_inter = linear_model.LogisticRegression(C=1)
    lr_inter.fit(Ximb, yimb)
    p, r, f, s = metrics.precision_recall_fscore_support(yimb, lr_inter.predict(Ximb))
    print("SPC: %.3f; SEN: %.3f" % tuple(r))
    print('# => Sensitivity >> specificity\n')
    print('# Reweighting on imbalanced dataset')
    lr_inter_reweight = linear_model.LogisticRegression(C=1, clas\boldsymbol{S_W}
eight="balanced")
    lr_inter_reweight.fit(Ximb, yimb)
    p, r, f, s = metrics.precision_recall_fscore_support(yimb, lr_inter_reweight.predict(Ximb))
    print("SPC: %.3f; SEN: %.3f" % tuple(r))
    print('# => The predictions are balanced in sensitivity and specificity\n')
File "<ipython-input-34-2de881c6d3f4>", line 43 lr_inter_reweight = linear_model.LogisticRegression(C=1,
clas\boldsymbol{S_W} _eight="balanced")

```

SyntaxError: unexpected character after line continuation character

TWELVE

NON LINEAR LEARNING ALGORITHMS

Support Vector Machines (SVM)

SVM are based kernel methods require only a user-specified kernel function $K(x_i, x_j)$, i.e., a similarity function over pairs of data points (x_i, x_j) into kernel (dual) space on which learning algorithms operate linearly, i.e. every operation on points is a linear combination of $K(x_i, x_j)$.

Outline of the SVM algorithm:

1. Map points x into kernel space using a kernel function: $x \rightarrow K(x, .)$.
2. Learning algorithms operate linearly by dot product into high-kernel space $K(., x_i) \cdot K(., x_j)$.
 - Using the kernel trick (Mercer's Theorem) replace dot product in high dimensional space by a simpler operation such that $K(., x_i) \cdot K(., x_j) = K(x_i, x_j)$. Thus we only need to compute a similarity measure for each pairs of point and store in a $N \times N$ Gram matrix.
 - Finally, The learning process consist of estimating the α_i of the decision function that maximises the hinge loss (of $f(x)$) plus some penalty when applied on all training points.
3. Predict a new point x using the decision function.

Gaussian kernel (RBF, Radial Basis Function):

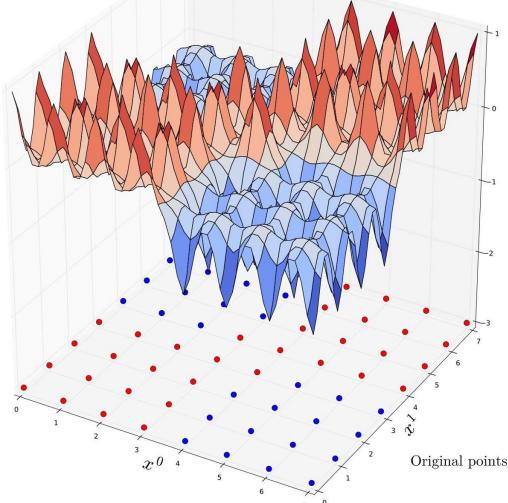
One of the most commonly used kernel is the Radial Basis Function (RBF) Kernel. For a pair of points x_i, x_j the RBF kernel is defined as:

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right) \quad (12.1)$$

Where σ (or γ) defines the kernel width parameter. Basically, we consider a Gaussian function centered on each training sample x_i , it has a ready interpretation as a similarity measure as it decreases with squared Euclidean distance between the two feature vectors.

Non linear SVM also exists for regression problems.

```
import numpy as np
from sklearn.svm import SVC
from sklearn import datasets
(1) Kernel mapping:  
x → K(x_i, x) = exp(-||x_i - x_j||^2 / 2σ^2)
```



(2) Learn the decision function:
 $f(x) = \text{sign}\left(\sum_{i \in SV} \alpha_i y_i \exp\left(-\frac{\|x_i - x\|^2}{2\sigma^2}\right)\right)$

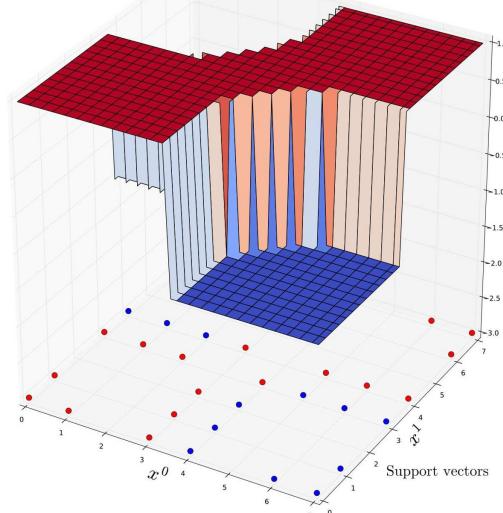


Fig. 12.1: Support Vector Machines.

```

import matplotlib.pyplot as plt
# dataset
X, y = datasets.make_classification(n_samples=10, n_features=2, n_redundant=0, n_classes=2,
                                     random_state=1, shuffle=False)
clf = SVC(kernel='rbf')
#, gamma=1)
clf.fit(X, y)
print("#Errors: %i" % np.sum(y != clf.predict(X)))
clf.decision_function(X)
# Useful internals:
# Array of support vectors
clf.support_vectors_
# indices of support vectors within original X
np.all(X[clf.support_[:,]] == clf.support_vectors_)
#Errors: 0
True

```

Random forest

A random forest is a meta estimator that fits a number of decision tree learners on various subsamples of the dataset and use averaging to improve the predictive accuracy and control overfitting.

Decision tree learner

A tree can be “learned” by splitting the training dataset into subsets based on an features value test.

Each internal node represents a “test” on an feature resulting on the split of the current sample. At each step the algorithm selects the feature and a cut-off value that maximises a given metric. Different metrics exist for regression tree (target is continuous) or classification tree (the target is qualitative).

This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. This general principle is implemented by many recursive partitioning tree algorithms.

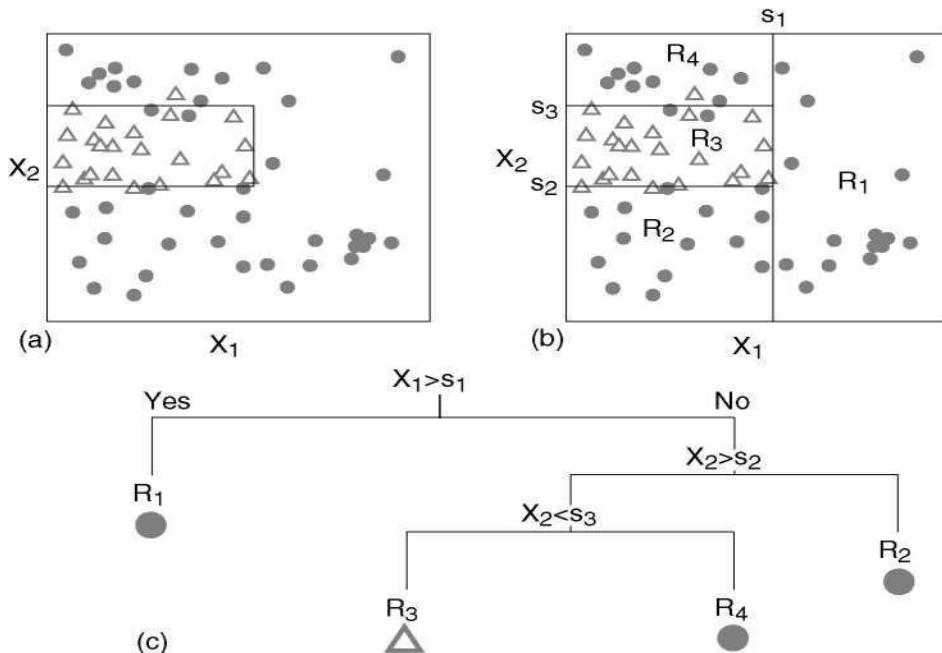


Fig. 12.2: Classification tree.

Decision trees are simple to understand and interpret however they tend to overfit the data. However decision trees tend to overbite the training set. Leo Breiman propose random forest to deal with this issue.

```
from sklearn.ensemble import RandomForestClassifier
forest = RandomForestClassifier(n_estimators = 100)
forest.fit(X, y)
print("#Errors: %d" % np.sum(y != forest.predict(X)))
```

CHAPTER THIRTEEN

RESAMPLING METHODS

Left out samples validation

The training error can be easily calculated by applying the statistical learning method to the observations used in its training. But because of overfitting, the training error rate can dramatically underestimate the error that would be obtained on new samples.

The test error is the average error that results from a learning method to predict the response on a new samples that is, on samples that were not used in training the method. Given a data set, the use of a particular learning method is warranted if it results in a low test error. The test error can be easily calculated if a designated test set is available. Unfortunately, this is usually not the case.

Thus the original dataset is generally split in a training and a test (or validation) data sets. Large training set (80%) small test set (20%) might provide a poor estimation of the predictive performances. On the contrary, large test set and small training set might produce a poorly estimated learner. This is why, on situation where we cannot afford such split, it recommended to use cross-Validation scheme to estimate the predictive power of a learning algorithm.

Cross-Validation (CV)

Cross-Validation scheme randomly divides the set of observations into K groups, or folds, of approximately equal size. The first fold is treated as a validation set, and the method $f()$ is fitted on the remaining union of $K - 1$ folds: $(f(X_{-K}, y_K))$.

The mean error measure (generally a loss function) is evaluated on the observations in the held-out fold. For each sample i we consider the model estimated on the data set that did not contain it, noted $-K(i)$. This procedure is repeated K times; each time, a different group of observations is treated as a test set. Then we compare the predicted value $(f(X_{-K(i)}) = y_i^*)$ with true value y_i using a Error function $L()$. Then the cross validation estimate of prediction error is

$$CV(f) = \frac{1}{N} \sum_i^N L(y_i, f(X_{-K(i)}))$$

This validation scheme is known as the K-Fold CV. Typical choices of K are 5 or 10, [Kohavi 1995]. The extreme case where $K = N$ is known as leave-one-out cross-validation, LOO-CV.

CV for regression

Usually the error function $L()$ is the r-squared score. However other function could be used.

```

import numpy as np from sklearn
import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics from
sklearn.cross_validation
import KFold
X, y = datasets.make_regression(n_samples=100, n_features=100, n_informative=10, random_state=42)
model = lm.Ridge(alpha=10)
cv = KFold(len(y), n_folds=5, random_state=42)
y_test_pred = np.zeros(len(y)) y_train_pred = np.zeros(len(y))
for train, test in cv:
    X_train, X_test, y_train, y_test = X[train, :], X[test, :], y[train], y[test] model.fit(X_train, y_train)
    y_test_pred[test] = model.predict(X_test) y_train_pred[train] = model.predict(X_train)
print("Train r2: %.2f" % metrics.r2_score(y, y_train_pred))
print("Test r2: %.2f" % metrics.r2_score(y, y_test_pred))

```

Train r2:0.99
Test r2:0.72

Scikit-learn provides user-friendly function to perform CV:

```

from sklearn.cross_validation import cross_val_score
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test r2: %.2f" % scores.mean())
# provide a cv scores = cross_val_score(estimator=model, X=X, y=y, cv=cv)
print("Test r2: %.2f" % scores.mean())

```

Test r2:0.73
Test r2:0.73

CV for classification

With classification problems it is essential to sample folds where each set contains approximately the same percentage of samples of each target class as the complete set. This is called stratification. In this case, we will use StratifiedKFold which is a variation of k-fold which returns stratified folds.

Usually the error function $L()$ are, at least, the sensitivity and the specificity. However other function could be used.

```

import numpy as np from sklearn
import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
from sklearn.cross_validation import StratifiedKFold
X, y = datasets.make_classification(n_samples=100, n_features=100, n_informative=10, random_state=42)

model = lm.LogisticRegression(C=1)
cv = StratifiedKFold(y, n_folds=5)
y_test_pred = np.zeros(len(y))
y_train_pred = np.zeros(len(y))
for train, test in cv:
    X_train, X_test, y_train, y_test = X[train, :], X[test, :], y[train], y[test]
    model.fit(X_train, y_train) y_test_pred[test] = model.predict(X_test)
    y_train_pred[train] = model.predict(X_train)
recall_test = metrics.recall_score(y, y_test_pred, average=None)
recall_train = metrics.recall_score(y, y_train_pred, average=None)
acc_test = metrics.accuracy_score(y, y_test_pred)
print("Train SPC: %.2f, SEN: %.2f" % tuple(recall_train))
print("Test SPC: %.2f, SEN: %.2f" % tuple(recall_test))
print("Test ACC: %.2f" % acc_test)

```

Train SPC:1.00; SEN:1.00
Test SPC:0.80; SEN:0.82
Test ACC:0.81

Scikit-learn provides user-friendly function to perform CV:

```
from sklearn.cross_validation import cross_val_score
scores = cross_val_score(estimator=model, X=X, y=y, cv=5) scores.mean()
# provide CV and score
def balanced_acc(estimator, X, y):
    """
    Balanced accuracy scorer
    """
    return metrics.recall_score(y, estimator.predict(X), average=None).mean()
scores = cross_val_score(estimator=model, X=X, y=y, cv=cv, scoring=balanced_acc)
print("Test ACC: %.2f" % scores.mean())
Test ACC:0.81
```

Note that with Scikit-learn user-friendly function we average the scores' average obtained on individual folds which may provide slightly different results than the overall average presented earlier.

CV for model selection: setting the hyper parameters

It is important to note CV may be used for two separate goals:

1. Model assessment: having chosen a final model, estimating its prediction error (generalization error) on new data.
2. Model selection: estimating the performance of different models in order to choose the best one. One special case of model selection is the selection model's hyper parameters. Indeed remember that most of learning algorithm have a hyper parameters (typically the regularization parameter) that has to be set.

Generally we must address the two problems simultaneously. The usual approach for both problems is to randomly divide the dataset into three parts: a training set, a validation set, and a test set.

- The training set (train) is used to fit the models;
- the validation set (val) is used to estimate prediction error for model selection or to determine the hyper parameters over a grid of possible values.
- the test set (test) is used for assessment of the generalization error of the final chosen model.

Grid search procedure

Model selection of the best hyper parameters over a grid of possible values

For each possible values of hyper parameters α_k :

1. Fit the learner on training set: $f(X_{train}, y_{train}, \alpha_k)$
2. Evaluate the model on the validation set and keep the parameter(s) that minimises the error measure $\alpha^* = \operatorname{argmin} L(f(X_{train}), y_{val}, \alpha_k)$
3. Refit the learner on all training + validation data using $f(X_{trainUval}, y_{trainUval}, \alpha^*)$
4. ** Model assessment ** of f on the test set: $L(f(X_{test}), y_{test})$

Nested CV for model selection and assessment

Most of time, we cannot afford such three-way split. Thus, again we will use CV, but in this case we need two nested CVs.

One outer CV loop, for model assessment. This CV performs K splits of the dataset into training plus validation (X_{-K}, y_{-K}) set and a test set X_K, y_K

One inner CV loop, for model selection. For each run of the outer loop, the inner loop performs L splits of dataset (X_K, y_K) into training set: (X_{K-L}, y_{K-L}) and a validation set: $(X_{K,L}, y_{K,L})$.

Implementation with scikit-learn

Note that the inner CV loop combined with the learner form a new learner with an automatic model (parameter) selection procedure. This new learner can be easily constructed using Scikit-learn. The learned is wrapped inside a GridSearchCV class.

Then the new learned can be plused into the classical outer CV loop.

```
import numpy as np from sklearn
import datasets
import sklearn.linear_model as lm
from sklearn.grid_search import GridSearchCV
import sklearn.metrics as metrics
from sklearn.cross_validation
import KFold
# Dataset
noise_sd = 10

X, y, coef = datasets.make_regression(n_samples=50, n_features=100, noise=noise_sd, n_informative=2,
random_state=42, coef=True)
# Use this to tune the noise parameter such that snr < 5
print("SNR:", np.std(np.dot(X, coef)) / noise_sd)
# param grid over alpha & l1_ratio
param_grid = {'alpha': 10. ** np.arange(-3, 3), 'l1_ratio':[.1, .5, .9]}
# Wrap
model = GridSearchCV(lm.ElasticNet(max_iter=10000), param_grid, cv=5)
# 1) Biased usage: fit on all data, ommit outer CV loop
model.fit(X, y)
print("Train r2:%.2f" % metrics.r2_score(y, model.predict(X)))
print(model.best_params_)
# 2) User made outer CV, usefull to extract specific imformation
cv = KFold(len(y), n_folds=5, random_state=42) y_test_pred = np.zeros(len(y))
y_train_pred = np.zeros(len(y)) alphas = list()
for train, test in cv:
    X_train, X_test, y_train, y_test = X[train, :], X[test, :], y[train], y[test] model.fit(X_train, y_train)
    y_test_pred[test] = model.predict(X_test) y_train_pred[train] = model.predict(X_train)
    alphas.append(model.best_params_)
print("Train r2:%.2f" % metrics.r2_score(y, y_train_pred))
print("Test r2:%.2f" % metrics.r2_score(y, y_test_pred))
print("Selected alphas:", alphas)
# 3.) user-friendly sklearn for outer CV
from sklearn.cross_validation import cross_val_score
scores = cross_val_score(estimator=model, X=X, y=y, cv=cv)
print("Test r2:%.2f" % scores.mean())
SNR: 2.63584694464
Train r2:0.96
{'l1_ratio': 0.9, 'alpha': 1.0}
Train r2:1.00
Test r2:0.62
Selected alphas: [{'l1_ratio': 0.9, 'alpha': 0.001}, {'l1_ratio': 0.9, 'alpha': 0.001}, {'l1_ratio': 0.9, 'alpha': 0.001}, {'l1_ratio': 0.9, 'alpha': 0.01}, {'l1_ratio': 0.9, 'alpha': 0.001}]
Test r2:0.55
```

Regression models with built-in cross-validation

Sklearn wil automatically select a grib of parameters, most of time use the defaults values. n_jobs is the number of CPUs to use during the cross validation. If -1, use all the CPUs.

CV for model selection: setting the hyper parameters

```

from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
from sklearn.cross_validation import cross_val_score
# Dataset
X, y, coef = datasets.make_regression(n_samples=50, n_features=100, noise=10, n_informative=2,
                                       random_state=42, coef=True)
print("== Ridge (L2 penalty) ==")
model = lm.RidgeCV()
# Let sklearn select a list of alphas with default LOO-CV
scores = cross_val_score(estimator=model, X=X, y=y, cv=5) print("Test r2:%.2f" % scores.mean())
print("== Lasso (L1 penalty) ==")
model = lm.LassoCV(n_jobs=-1)
# Let sklearn select a list of alphas with default 3CV
scores = cross_val_score(estimator=model, X=X, y=y, cv=5) print("Test r2:%.2f" % scores.mean())
print("== ElasticNet (L1 penalty) ==")
model = lm.ElasticNetCV(l1_ratio=[.1, .5, .9], n_jobs=-1) # Let sklearn select
# a list of alphas with default 3CV
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())

```

```

== Ridge (L2 penalty) ==
Test r2:0.23
== Lasso (L1 penalty) ==
Test r2:0.74
== ElasticNet (L1 penalty) ==
Test r2:0.58

```

Classification models with built-in cross-validation

```

from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
from sklearn.cross_validation import cross_val_score
X, y = datasets.make_classification(n_samples=100, n_features=100, n_informative=10, random_state=42)
# provide CV and score
def balanced_acc(estimator, X, y):
    """
    Balanced accuracy scorer
    """
    return metrics.recall_score(y, estimator.predict(X), average=None).mean()
print("== Logistic Ridge (L2 penalty) ==")
model = lm.LogisticRegressionCV(class_weight='balanced', scoring=balanced_acc, n_jobs=-1)
# Let sklearn select a list of alphas with default LOO-CV

```

```

scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test ACC:%.2f" % scores.mean())

```

```

== Logistic Ridge (L2 penalty) ==
Test ACC:0.77

```

Random Permutations

A permutation test is a type of non-parametric randomization test in which the null distribution of a test statistic is estimated by randomly permuting the observations.

Permutation tests are highly attractive because they make no assumptions other than that the observations are independent and identically distributed under the null hypothesis.

1. Compute a observed statistic t_{obs} on the data.
2. Use randomization to compute the distribution of t under the null hypothesis: Perform N random permutation of the data. For each sample of permuted data, i the data compute the statistic t_i . This procedure provides the distribution of t under the null hypothesis H_0 : $P(t|H_0)$

3. Compute the p-value = $P(t > t_{obs}|H_0)|\{t_i > t_{obs}\}|$, where t_i 's include t_{obs} .

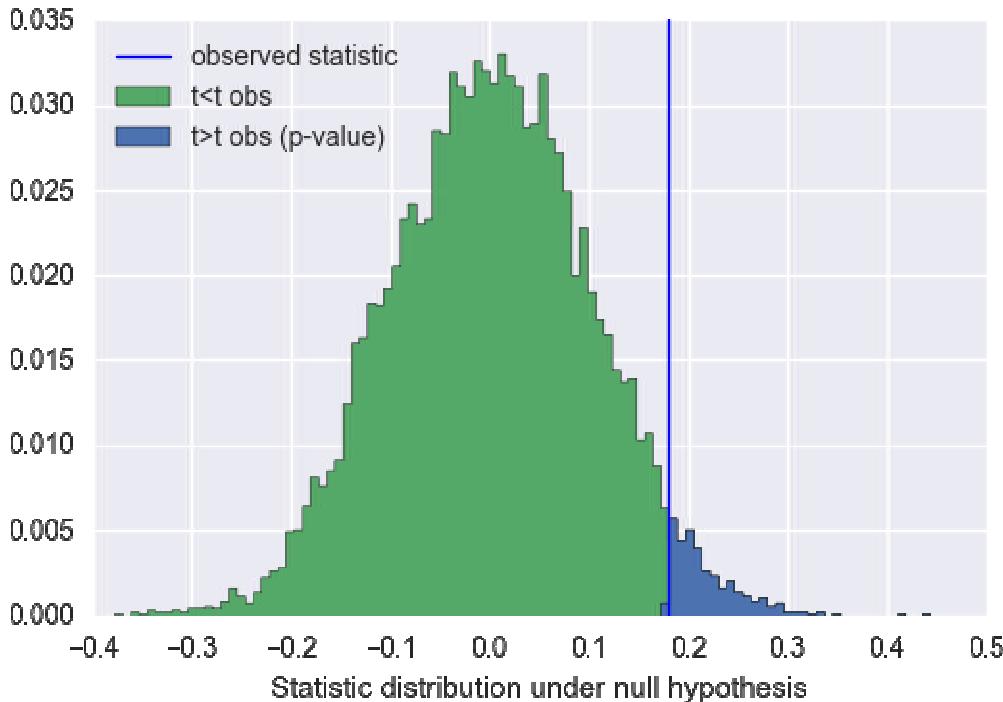
Example with a correlation

The statistic is the correlation.

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline %%matplotlib qt
np.random.seed(42) x = np.random.normal(loc=10, scale=1,
size=100)
y = x + np.random.normal(loc=-3, scale=3, size=100) # snr = 1/2
# Permutation: simulate the null hypothesis nperm = 10000
perms = np.zeros(nperm + 1)
perms[0] = np.corrcoef(x, y)[0, 1]
for i in range(1, nperm):
    perms[i] = np.corrcoef(np.random.permutation(x), y)[0, 1]
# Plot
# Re-weight to obtain distribution
weights = np.ones(perms.shape[0]) / perms.shape[0]
plt.hist([perms[perms >= perms[0]], perms], histtype='stepfilled',
        bins=100, label=["t>t obs (p-value)", "t< t obs"], weights=[weights[perms >= perms[0]], weights])
plt.xlabel("Statistic distribution under null hypothesis") plt.axvline(x=perms[0], color='blue', linewidth=1,
label="observed statistic")

plt.legend(loc="upper left")
# One-tailed empirical p-value pval_perm = np.sum(perms >= perms[0]) /
perms.shape[0]
# Compare with Pearson's correlation test _, pval_test = stats.pearsonr(x, y) print("Permutation two tailed p-
value=% .5f. Pearson test p-value=% .5f" % (2*pval_>perm, pval_test))
```

Permutation two tailed p-value=0.06959. Pearson test p-value=0.07355



Exercise

Given the logistic regression presented above and its validation given a 5 folds CV.

1. Compute the p-value associated with the prediction accuracy using a permutation test.
2. Compute the p-value associated with the prediction accuracy using a parametric test.

Bootstrapping

Bootstrapping is a random sampling with replacement strategy which provides an non-parametric method to assess the variability of performances scores such standard errors or confidence intervals.

A great advantage of bootstrap is its simplicity. It is a straightforward way to derive estimates of standard errors and confidence intervals for complex estimators of complex parameters of the distribution, such as percentile points, proportions, odds ratio, and correlation coefficients.

1. Perform B sampling, with replacement, of the dataset.
2. For each sample i fit the model and compute the scores.
3. Assess standard errors and confidence intervals of scores using the scores obtained on the B resampled dataset.

```
import numpy as np from sklearn
import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
import pandas as pd
# Regression dataset
n_features = 5
n_features_info = 2
n_samples = 100
X = np.random.randn(n_samples, n_features)
beta = np.zeros(n_features)
beta[:n_features_info] = 1
Xbeta = np.dot(X, beta)
eps = np.random.randn(n_samples)
y = Xbeta + eps
# Fit model on all data (!! risk of overfit)
model = lm.RidgeCV()
model.fit(X, y)
print("Coefficients on all data:")
print(model.coef_)
# Bootstrap loop
nboot = 100 # !! Should be at least 1000
scores_names = ["r2"]
scores_boot = np.zeros((nboot, len(scores_names)))
coefs_boot = np.zeros((nboot, X.shape[1]))
orig_all = np.arange(X.shape[0])
for boot_i in range(nboot):
    boot_tr = np.random.choice(orig_all, size=len(orig_all), replace=True)
    boot_te = np.setdiff1d(orig_all, boot_tr, assume_unique=False)
    Xtr, ytr = X[boot_tr, :], y[boot_tr]
    Xte, yte = X[boot_te, :], y[boot_te]
    model.fit(Xtr, ytr)
    y_pred = model.predict(Xte).ravel()
    scores_boot[boot_i, :] = metrics.r2_score(yte, y_pred)
    coefs_boot[boot_i, :] = model.coef_
# Compute Mean, SE, CI
scores_boot = pd.DataFrame(scores_boot, columns=scores_names)
scores_stat = scores_boot.describe(percentiles=[.99, .95, .5, .1, .05, .01])
print("r-squared: Mean=%2f, SE=%2f, CI=(%2f %2f) %\ tuple(scores_stat.ix[['mean", "std", "5%", "95%"], "r2"]))
coefs_boot = pd.DataFrame(coefs_boot)
coefs_stat = coefs_boot.describe(percentiles=[.99, .95, .5, .1, .05, .01])
print("Coefficients distribution")
print(coefs_stat)
```

Coefficients on all data:

[0.98143428 0.84248041 0.12029217 0.09319979 0.08717254] r-squared:

Mean=0.57, SE=0.09, CI=(0.39 0.70) Coefficients distribution

	0	1	2	3	4
count	100.000000	100.000000	100.000000	100.000000	100.000000
mean	0.975189				
0.831922	0.116888	0.099109	0.085516	std 0.106367	0.096548 0.108676 0.090312
0.091446	min 0.745082	0.593736	-0.112740	-0.126522	-0.141713 1% 0.770362
0.640142	-0.088238	-0.094403	-0.113375		
5%	0.787463	0.657473	-0.045593	-0.046201	-0.090458
10%	0.829129	0.706492	-0.037838	-0.020650	-0.044990
50%	0.980603	0.835724	0.133070	0.093240	0.088968
95%	1.127518	0.999604	0.278735	0.251137	
0.221887 99%	1.144834	1.036715		0.292784	
0.291197	0.287006 max	1.146670		1.077265	
0.324374	0.298135	0.289569			

CHAPTER FOURTEEN

SCIKIT-LEARN PROCESSING PIPELINES

Data preprocessing

Sources: <http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html>

Encoding categorical features

To be done

```
import pandas as pd
pd.get_dummies(['A', 'B', 'C', 'A', 'B', 'D'])
```

Standardization of input features

Sources:

- <http://scikit-learn.org/stable/modules/preprocessing.html>
- <http://stats.stackexchange.com/questions/111017/question-about-standardizing-in-ridge-regression>

“Standardizing” or mean removal and variance scaling, is not systematic. For example multiple linear regression does not require it. However it is a good practice in many cases:

- The variable combination method is sensitive to scales. If the input variables are combined via a distance function (such as Euclidean distance) in an RBF network, standardizing inputs can be crucial. The contribution of an input will depend heavily on its variability relative to other inputs. If one input has a range of 0 to 1, while another input has a range of 0 to 1,000,000, then the contribution of the first input to the distance will be swamped by the second input.
- Regularized learning algorithm. Lasso or Ridge regression regularize the linear regression by imposing a penalty on the size of coefficients. Thus the coefficients are shrunk toward zero and toward each other. But when this happens and if the independent variables do not have the same scale, the shrinking is not fair. Two independent variables with different scales will have different contributions to the penalized terms, because the penalized term is norm (a sum of squares, or absolute values) of all the coefficients. To avoid such kind of problems, very often, the independent variables are centered and scaled in order to have variance 1.

```
import numpy as np
# dataset
np.random.seed(42)
n_samples, n_features, n_features_info = 100, 5, 3
```

```

X = np.random.randn(n_samples, n_features)
beta = np.zeros(n_features)
beta[:n_features_info] = 1
Xbeta = np.dot(X, beta)
eps = np.random.randn(n_samples)
y = Xbeta + eps
X[:, 0] *= 1e6 # inflate the first feature
X[:, 1] += 1e6 # bias the second feature
y = 100 * y + 1000 # bias and scale the output
import sklearn.linear_model as lm from sklearn
import preprocessing
from sklearn.cross_validation
import cross_val_score
print("== Linear regression: scaling is not required ==")
model = lm.LinearRegression()
model.fit(X, y)
print("Coefficients:", model.coef_, model.intercept_)
print("Test R2: %.2f" % cross_val_score(estimator=model, X=X, y=y, cv=5).mean())
print("== Lasso without scaling ==")
model = lm.LassoCV() model.fit(X, y)
print("Coefficients:", model.coef_, model.intercept_)
print("Test R2: %.2f" % cross_val_score(estimator=model, X=X, y=y, cv=5).mean())
print("== Lasso with scaling ==")
model = lm.LassoCV()
scaler = preprocessing.StandardScaler()
Xc = scaler.fit(X).transform(X)
model.fit(Xc, y)
print("Coefficients:", model.coef_, model.intercept_)
print("Test R2: %.2f" % cross_val_score(estimator=model, X=Xc, y=y, cv=5).mean())
== Linear regression: scaling is not required == Coefficients: [
1.05421281e-04 1.13551103e+02
-7.23145329e-01] -113550117.827
Test R2: 0.77
== Lasso without scaling ==
Coefficients: [ 8.61125764e-05
0.00000000e+00] 986.15608907
Test R2: 0.09
== Lasso with scaling ==
Coefficients: [ 87.46834069 105.13635448
-982.302793647 Test
R2: 0.77

```

Scikit-learn pipelines

Sources: <http://scikit-learn.org/stable/modules/pipeline.html>

Note that statistics such as the mean and standard deviation are computed from the training data, not from the validation or test data. The validation and test data must be standardized using the statistics computed from the training data. Thus Standardization should be merged together with the learner using a Pipeline.

Pipeline chain multiple estimators into one. All estimators in a pipeline, except the last one, must have the fit() and transform() methods. The last must implement the fit() and predict() methods.

Standardization of input features

```
from sklearn
import preprocessing
import sklearn.linear_model as lm
from sklearn.pipeline
import make_pipeline
model = make_pipeline(preprocessing.StandardScaler(), lm.LassoCV())
# or
from sklearn.pipeline import Pipeline
model = Pipeline([('standardscaler', preprocessing.StandardScaler()), ('lassocv', lm.LassoCV())])
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test r2: %.2f" % scores.mean())
Test r2: 0.77
```

Features selection

An alternative to features selection based on ℓ_1 penalty is to use a preprocessing step of univariate feature selection.

Such methods, called filters, are a simple, widely used method for supervised dimension reduction [26]. Filters are univariate methods that rank features according to their ability to predict the target, independently of other features. This ranking may be based on parametric (e.g., t-tests) or nonparametric (e.g., Wilcoxon tests) statistical methods. Filters are computationally efficient and more robust to overfitting than multivariate methods.

```
import numpy as np
import sklearn.linear_model as lm from sklearn
import preprocessing from sklearn.cross_validation
import cross_val_score from sklearn.feature_selection
import SelectKBest from sklearn.feature_selection
import f_regression from sklearn.pipeline
import Pipeline
np.random.seed(42)
n_samples, n_features, n_features_info = 100, 100, 3
X = np.random.randn(n_samples, n_features)
beta = np.zeros(n_features)
beta[:n_features_info] = 1
Xbeta = np.dot(X, beta)
eps = np.random.randn(n_samples)
y = Xbeta + eps
X[:, 0] *= 1e6 # inflate the first feature

X[:, 1] += 1e6 # bias the second feature
y = 100 * y + 1000 # bias and scale the output
model = Pipeline([('anova', SelectKBest(f_regression, k=3)), ('lm', lm.LinearRegression())])

scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Anova filter + linear regression, test r2: %.2f" % scores.mean())
from sklearn.pipeline import Pipeline
model = Pipeline([('standardscaler', preprocessing.StandardScaler()), ('lassocv', lm.LassoCV())])
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Standardize + Lasso, test r2: %.2f" % scores.mean())
```

```
Anova filter + linear regression, test r2: 0.72
Standardize + Lasso, test r2: 0.66
```

Regression pipelines with CV for parameters selection

Now we combine standardization of input features, feature selection and learner with hyperparameter within a pipeline which is warped in a grid search procedure to select the best hyperparameters based on a (inner)CV. The overall is plugged in an outer CV.

```
import numpy as np from sklearn
import datasets
import sklearn.linear_model as lm from sklearn
import preprocessing from sklearn.cross_validation
import cross_val_score from sklearn.feature_selection
import SelectKBest from sklearn.feature_selection
import f_regression from sklearn.pipeline
import Pipeline from sklearn.grid_search
import GridSearchCV
import sklearn.metrics as metrics
# Datasets
n_samples, n_features, noise_sd = 100, 100, 20
X, y, coef = datasets.make_regression(n_samples=n_samples, n_features=n_features, noise=noise_sd,
n_informative=5, random_state=42, coef=True)
# Use this to tune the noise parameter such that snr < 5
print("SNR:", np.std(np.dot(X, coef)) / noise_sd)
print("====")
print("== Basic linear regression ==")
print("====")
scores = cross_val_score(estimator=lm.LinearRegression(), X=X, y=y, cv=5)
print("Test r2: %.2f" % scores.mean())
print("====")
print("== Scaler + anova filter + ridge regression ==")
print("====")
anova_ridge = Pipeline([
    ('standardscaler', preprocessing.StandardScaler()),
    ('selectkbest', SelectKBest(f_regression)),
    ('ridge', lm.Ridge())
]) param_grid = {'selectkbest_k':np.arange(10, 110, 10),
```

```

        'ridge_alpha':[.001, .01, .1, 1, 10, 100] }

# Expect execution in ipython, for python remove the %time
print("-----")
print("-- Parallelize inner loop --")
print("-----")
anova_ridge_cv = GridSearchCV(anova_ridge, cv=5, param_grid=param_grid, n_jobs=-1)
%time scores = cross_val_score(estimator=anova_ridge_cv, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())
print("-----")
print("-- Parallelize outer loop --") print("-----")
anova_ridge_cv = GridSearchCV(anova_ridge, cv=5, param_grid=param_grid)
%time scores = cross_val_score(estimator=anova_ridge_cv, X=X, y=y, cv=5, n_jobs=-1)
print("Test r2:%.2f" % scores.mean())
print("=====")
print("==     Scaler      +      Elastic-net      regression      ==")
print("=====")
alphas = [.0001, .001, .01, .1, 1, 10, 100, 1000] l1_ratio = [.1, .5, .9]
print("-----")
print("-- Parallelize outer loop --")
print("-----")
enet = Pipeline([
    ('standardscaler', preprocessing.StandardScaler()),
    ('enet', lm.ElasticNet(max_iter=10000)),
])
param_grid = {'enet_alpha': alphas,
'enet_l1_ratio':l1_ratio}
enet_cv = GridSearchCV(enet, cv=5, param_grid=param_grid)
%time scores = cross_val_score(estimator=enet_cv, X=X, y=y, cv=5, n_jobs=-1)
print("Test r2:%.2f" % scores.mean())
print("-----")
print("-- Parallelize outer loop + built-in CV  --")
print("-- Remark: scaler is only done on outer loop --")
print("-----")
enet_cv = Pipeline([
    ('standardscaler', preprocessing.StandardScaler()),
    ('enet', lm.ElasticNetCV(max_iter=10000, l1_ratio=l1_ratio, alphas=alphas)),
])
%time scores = cross_val_score(estimator=enet_cv, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())
SNR: 3.28668201676
=====
== Basic linear regression ==
=====
```

```

Test r2:0.29
=====
== Scaler + anova filter + ridge regression ==
=====

-- Parallelize inner loop --

CPU times: user 6.06 s, sys: 836 ms, total: 6.9 s
Wall time: 7.97 s
Test r2:0.86
-----
-- Parallelize outer loop --

CPU times: user 270 ms, sys: 129 ms, total: 399 ms
Wall time: 3.51 s
Test r2:0.86
=====
== Scaler + Elastic-net regression ==
=====

-- Parallelize outer loop --

CPU times: user 44.4 ms, sys: 80.5 ms, total: 125 ms
Wall time: 1.43 s
Test r2:0.82
-----
-- Parallelize outer loop + built-in CV
-- Remark: scaler is only done on outer loop --
-----CPU times: user 227 ms, sys: 0
ns, total: 227 ms
Wall time: 225 ms
Test r2:0.82

```

Classification pipelines with CV for parameters selection

Now we combine standardization of input features, feature selection and learner with hyperparameter within a pipeline which is warped in a grid search procedure to select the best hyperparameters based on a (inner)CV. The overall is plugged in an outer CV.

```

import numpy as np from sklearn
import datasets
import sklearn.linear_model as lm from sklearn
import preprocessing from sklearn.cross_validation
import cross_val_score from sklearn.feature_selection
import SelectKBest from sklearn.feature_selection
import f_classif from sklearn.pipeline
import Pipeline from sklearn.grid_search
import GridSearchCV
import sklearn.metrics as metrics
# Datasets
n_samples, n_features, noise_sd = 100, 100, 20
X, y = datasets.make_classification(n_samples=n_samples, n_features=n_features, n_informative=5,
random_state=42)

```

```

def balanced_acc(estimator, X, y):
    """
    Balanced accuracy scorer
    """
    return metrics.recall_score(y, estimator.predict(X), average=None).mean()
print("====")
print("== Basic logistic regression ==")
print("====")
scores = cross_val_score(estimator=lm.LogisticRegression(C=1e8, class_weight='balanced'),
                           X=X, y=y, cv=5, scoring=balanced_acc)
print("Test bACC: %.2f" % scores.mean())
print("====")
print("== Scaler + anova filter + ridge logistic regression ==")
print("====")
anova_ridge = Pipeline([
    ('standardscaler', preprocessing.StandardScaler()),
    ('selectkbest', SelectKBest(f_classif)),
    ('ridge', lm.LogisticRegression(penalty='l2', class_weight='balanced'))
])
param_grid = {'selectkbest__k': np.arange(10, 110, 10),
              'ridge__C': [.0001, .001, .01, .1, 1, 10, 100, 1000, 10000]}
# Expect execution in ipython, for python remove the %time
print("----")
print("-- Parallelize inner loop --")
print("----")
anova_ridge_cv = GridSearchCV(anova_ridge, cv=5, param_grid=param_grid, scoring=balanced_acc, n_jobs=-1)
%time scores = cross_val_score(estimator=anova_ridge_cv, X=X, y=y, cv=5, scoring=balanced_acc)
print("Test bACC: %.2f" % scores.mean())
print("----")
print("-- Parallelize outer loop --")
print("----")
anova_ridge_cv = GridSearchCV(anova_ridge, cv=5, param_grid=param_grid, scoring=balanced_acc)
%time scores = cross_val_score(estimator=anova_ridge_cv, X=X, y=y, cv=5, scoring=balanced_acc, n_jobs=-1)
print("Test bACC: %.2f" % scores.mean())
print("====")
print("== Scaler + lasso logistic regression ==")
print("====")
Cs = np.array([.0001, .001, .01, .1, 1, 10, 100, 1000, 10000])
alphas = 1 / Cs l1_ratio = [.1, .5, .9]

```

```

print("-----") print("-- Parallelize outer loop --")
print("-----")
lasso = Pipeline([
    ('standardscaler', preprocessing.StandardScaler()),
    ('lasso', lm.LogisticRegression(penalty='l1', class_weight='balanced')),
])
param_grid = {'lasso__C':Cs}
enet_cv = GridSearchCV(lasso, cv=5, param_grid=param_grid, scoring=balanced_acc) %time
scores = cross_val_score(estimator=enet_cv, X=X, y=y, cv=5,\ scoring=balanced_acc, n_jobs=-1)
print("Test bACC:%.2f" % scores.mean())
print("-----")
print("-- Parallelize outer loop + built-in CV --")
print("-- Remark: scaler is only done on outer loop --")
print("-----")
lasso_cv = Pipeline([
    ('standardscaler', preprocessing.StandardScaler()),
    ('lasso', lm.LogisticRegressionCV(Cs=Cs, scoring=balanced_acc)),
])
%time scores = cross_val_score(estimator=lasso_cv, X=X, y=y, cv=5)
print("Test bACC:%.2f" % scores.mean())
print("=====") print("== Scaler +")
Elasticnet      logistic      regression      ==
print("=====")
print("-----")
print("-- Parallelize outer loop --") print("-----")
enet = Pipeline([
    ('standardscaler', preprocessing.StandardScaler()),
    ('enet', lm.SGDClassifier(loss="log",      penalty="elasticnet",      alpha=0.0001,      l1_ratio=0.15,
                               class_weight='balanced')),
])
param_grid = {'enet_alpha':alphas,'enet_l1_ratio':l1_ratio}
enet_cv = GridSearchCV(enet, cv=5, param_grid=param_grid, scoring=balanced_acc)
%time scores = cross_val_score(estimator=enet_cv, X=X, y=y, cv=5,\ scoring=balanced_acc, n_jobs=-1)
print("Test bACC:%.2f" % scores.mean())

```

```

=====
== Basic logistic regression ==
=====
Test bACC:0.52
=====
== Scaler + anova filter + ridge logistic regression ==
=====
```

```

-- Parallelize inner loop --

CPU times: user 3.02 s, sys: 562 ms, total: 3.58 s
Wall time: 4.43 s
Test bACC:0.67
-----
-- Parallelize outer loop --

CPU times: user 59.3 ms, sys: 114 ms, total: 174 ms
Wall time: 1.88 s
Test bACC:0.67
=====
== Scaler + lasso logistic regression ==
=====
-- Parallelize outer loop --

CPU times: user 81 ms, sys: 96.7 ms, total: 178 ms

```

```
Wall time: 484 ms
Test bACC:0.57
-----
-- Parallelize outer loop + built-in CV
-- Remark: scaler is only done on outer loop --
-----
CPU times: user 575 ms, sys: 3.01 ms, total: 578 ms
Wall time: 327 ms
Test bACC:0.60
=====
== Scaler + Elasticnet logistic regression ==
=====

-----
-- Parallelize outer loop --
-----
CPU times: user 429 ms, sys: 100 ms, total: 530 ms
Wall time: 979 ms
Test bACC:0.61
```

CHAPTER
FIFTEEN
CASE STUDIES OF ML

Default of credit card clients Data Set

Sources:

[HYPERLINK "http://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients"](http://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients)

<http://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients>

Yeh, I. C., & Lien, C. H. (2009).

The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients.

Expert Systems with Applications, 36(2), 2473-2480.

Data Set Information:

This research aimed at the case of customers default payments in Taiwan.

Attribute Information:

This research employed a binary variable, default payment (Yes = 1, No = 0), as the response variable. This study reviewed the literature and used the following 23 variables as explanatory variables:

- X1: Amount of the given credit (NT dollar): it includes both the individual consumer credit and his/her family (supplementary) credit.
- X2: Gender (1 = male; 2 = female).
- X3: Education (1 = graduate school; 2 = university; 3 = high school; 4 = others).
- X4: Marital status (1 = married; 2 = single; 3 = others).
- X5: Age (year).
- X6 - X11: History of past payment. We tracked the past monthly payment records (from April to September, 2005) as follows: X6 = the repayment status in September, 2005; X7 = the repayment status in August, 2005;...;X11 = the repayment status in April, 2005. The measurement scale for the repayment status is: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months;...; 8 = payment delay for eight months; 9 = payment delay for nine months and above.
- X12-X17: Amount of bill statement (NT dollar). X12 = amount of bill statement in September, 2005; X13 = amount of bill statement in August, 2005;...; X17 = amount of bill statement in April, 2005.
- X18-X23: Amount of previous payment (NT dollar). X18 = amount paid in September, 2005; X19 = amount paid in August, 2005;...;X23 = amount paid in April, 2005.

Read dataset

```
from __future__ import print_function
import pandas as pd
import numpy as np
url = 'https://raw.github.com/neurospin/pystatsml/master/data/default%20of%20credit%20card%20clients.xls'
data = pd.read_excel(url, skiprows=1, sheetname='Data')
df = data.copy()
target = 'default payment next month'
print(df.columns)
#Index(['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_0',
#       'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2',
#       'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',
#       'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6',
#       'default payment next month'],
#       dtype='object')
```

Data recoding of categorial factors

- Categorical factors with 2 levels are kept
- Categorical that are ordinal are kept
- Undocumented values are replaced with NaN

Missing data

```
print(df.isnull().sum()) #ID
                           0
#LIMIT_BAL                 0
#SEX                         0
#EDUCATION                  468
#MARRIAGE                   377
#AGE                          0
#PAY_0                        0
#PAY_2                        0
#PAY_3                        0
#PAY_4                        0
#PAY_5                        0
#PAY_6                        0
#BILL_AMT1                   0
#BILL_AMT2                   0
#BILL_AMT3                   0
#BILL_AMT4                   0
#BILL_AMT5                   0
#BILL_AMT6                   0
#PAY_AMT1                   0
#PAY_AMT2                   0
#PAY_AMT3                   0
#PAY_AMT4                   0
#PAY_AMT5                   0
#PAY_AMT6                   0
```

```
#default payment next month          0
#dtype: int64
df.ix[df["EDUCATION"].isnull(), "EDUCATION"] = df["EDUCATION"].mean() df.ix[df["MARRIAGE"].isnull(),
"MARRIAGE"] = df["MARRIAGE"].mean() print(df.isnull().sum().sum()) # 0
describe_factor(df[target]) {0: 23364, 1: 6636}
```

Prepare Data set

```
predictors = df.columns.drop(['ID', target])
X = np.asarray(df[predictors])
y = np.asarray(df[target])
```

Univariate analysis

Machine Learning with SVM

On this large dataset, we can afford to set aside some test samples. This will also save computation time. However we will have to do some manual work.

```
import numpy as np from sklearn
import datasets
import sklearn.svm as svm from sklearn
import preprocessing from sklearn.cross_validation
import cross_val_score, train_test_split from sklearn.cross_validation
import StratifiedKFold from sklearn.feature_selection
import SelectKBest from sklearn.feature_selection
import f_classif from sklearn.pipeline
import Pipeline from sklearn.grid_search
import GridSearchCV import sklearn.metrics as metrics
def balanced_acc(estimator, X, y):
    return metrics.recall_score(y, estimator.predict(X), average=None).mean()
print("====")
print("== Put aside half of the samples as test set ==")
print("====")
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.5, random_state=0, stratify=y)
print("====")
print("== Scale trainin and test data ==")
print("====")
scaler = preprocessing.StandardScaler()
Xtrs = scaler.fit(Xtr).transform(Xtr)
Xtes = scaler.transform(Xte)
```

```

print("=====") print("== SVM
==") print("=====")
svc = svm.LinearSVC(class_weight='balanced', dual=False)
%time scores = cross_val_score(estimator=svc,\n    X=Xtrs, y=ytr, cv=2, scoring=balanced_acc)
print("Validation bACC:%.2f" % scores.mean())
#CPU times: user 1.01 s, sys: 39.7 ms, total: 1.05 s
#Wall time: 112 ms #Validation
bACC:0.67
svc_rbf = svm.SVC(kernel='rbf', class_weight='balanced')
%time scores = cross_val_score(estimator=svc_rbf,\n    X=Xtrs, y=ytr, cv=2, scoring=balanced_acc)
print("Validation bACC:%.2f" % scores.mean())
#CPU times: user 10.2 s, sys: 136 ms, total: 10.3 s
#Wall time: 10.3 s #Test
bACC:0.71
svc_lasso = svm.LinearSVC(class_weight='balanced', penalty='l1', dual=False)
%time scores = cross_val_score(estimator=svc_lasso,\n    X=Xtrs, y=ytr, cv=2, scoring=balanced_acc)
print("Validation bACC:%.2f" % scores.mean())
#CPU times: user 4.51 s, sys: 168 ms, total: 4.68 s
#Wall time: 544 ms #Test
bACC:0.67
print("=====") print("== SVM
CV Grid search ==")
print("=====")
Cs = [0.001, .01, .1, 1, 10, 100, 1000]
param_grid = {'C':Cs}
print("-----") print("-- SVM Linear L2 --")
print("-----")
svc_cv = GridSearchCV(svc, cv=3, param_grid=param_grid, scoring=balanced_acc, n_jobs=-1) # What
are the best parameters ?
%time svc_cv.fit(Xtrs, ytr).best_params_
#CPU times: user 211 ms, sys: 209 ms, total: 421 ms
#Wall time: 1.07 s #{'C': 0.01}
scores = cross_val_score(estimator=svc_cv,\n    X=Xtrs, y=ytr, cv=2, scoring=balanced_acc)
print("Validation bACC:%.2f" % scores.mean())
#Validation bACC:0.67
print("-----") print("-- SVM RBF --")
print("-----")
svc_rbf_cv = GridSearchCV(svc_rbf, cv=3, param_grid=param_grid,
scoring=balanced_acc, n_jobs=-1) # What are the best parameters ?

```

```

%time svc_rbf_cv.fit(Xtrs, ytr).best_params_
#Wall time: 1min 10s
#Out[6]: {'C': 1}
# reduce the grid search svc_rbf_cv.param_grid={'C': [0.1, 1,
10]} scores = cross_val_score(estimator=svc_rbf_cv,\n    X=Xtrs, y=ytr, cv=2, scoring=balanced_acc)
print("Validation bACC:%.2f" % scores.mean())
#Validation bACC:0.71
print("-----") print("-- SVM Linear L1 --")
print("-----")
svc_lasso_cv = GridSearchCV(svc_lasso, cv=3, param_grid=param_grid,
scoring=balanced_acc, n_jobs=-1) # What are the best parameters ?
%time svc_lasso_cv.fit(Xtrs, ytr).best_params_
#CPU times: user 514 ms, sys: 181 ms, total: 695 ms
#Wall time: 2.07 s
#Out[10]: {'C': 0.1}
# reduce the grid search svc_lasso_cv.param_grid={'C': [0.1, 1, 10]}
scores = cross_val_score(estimator=svc_lasso_cv,\n    X=Xtrs, y=ytr, cv=2, scoring=balanced_acc)
print("Validation bACC:%.2f" % scores.mean())
#Validation bACC:0.67
print("SVM-RBF, test bACC:%.2f" % balanced_acc(svc_rbf_cv, Xtes, yte))
# SVM-RBF, test bACC:0.70
print("SVM-Lasso, test bACC:%.2f" % balanced_acc(svc_lasso_cv, Xtes, yte))
# SVM-Lasso, test bACC:0.67

```