SERGIO A.
ROJAS-GALEANO

MODELS OF LEARNING AND OPTIMIZATION FOR DATA SCIENTISTS

A Python hands-on approach

Models of Learning and Optimization for Data Scientists

A Python hands–on approach

Sergio Rojas-Galeano, PhD.

School of Engineering

Universidad Distrital Francisco José de Caldas

About the Author: **Sergio Rojas-Galeano, PhD.**
Biosketch: With a background in Computer Science, Software Engineering and Machine Learning, Sergio is curious about the crossroads between natural and machine intelligence. His research interests are pattern recognition, large scale online learning and bio-inspired optimisation, with applications to bioinformatics, scientific software, learning on graphs and science dissemination. He is currently Full Professor at the School of Engineering of Universidad Distrital Francisco José de Caldas, Bogotá, Colombia.
email: srojas@udistrital.edu.co

ISBN: 978-958-48-7320-0

Preface

DATA IS NOWADAYS UBIQUITOUS , voluminous and puzzling. It is not surprise that scientists are so interested in analysing it, understanding it and discovering underlying complex patterns within it. And that's the origin of what is known as *Data Science*. But as much as the scientific interest in this respect is growing, so it is practitioners curiosity about potential applications in real life and development of technological tools for Data Science in non-academic contexts. This booklet has been designed to introduce newcomers to the essentials of Data Science using a hands-on approach rather than a theoretical perspective. For this aim, it addresses two of its most important branches: Machine Learning and Metaheuristics. The booklet presents many introductory examples as well as an assortment of **challenges** with varying difficulty levels proposed to the student, to be solved using the Python programming language, the current tool–of–choice adopted by the Data Science community. These challenges (nearly 90 programming exercises) will help students to acquire skills that hopefully will foster their academic or industry interests involving data analysis for knowledge discovery.

Gartner defines a *Citizen Data Scientist* "as a person who creates or generates models that use advanced diagnostic analytics or predictive and prescriptive capabilities, but whose primary job function is outside the field of statistics and analytics". See: `gartner.com/doc/3534848`

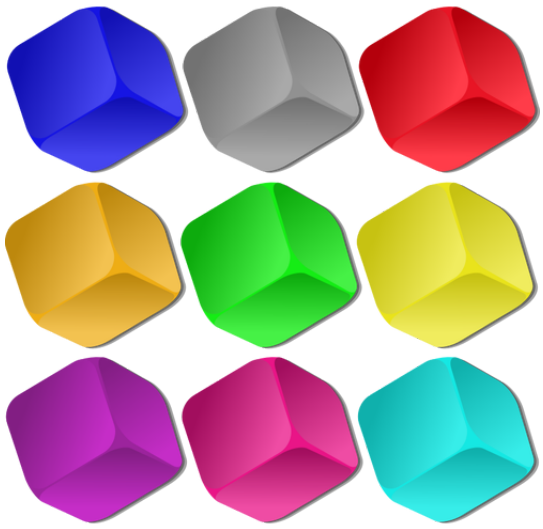
Contents

1 Introduction **5**

2 Machine Learning Models **7**
2.1 The big picture . . . 8
2.2 Exploratory data analysis . . . 10
2.3 Distance metrics . . . 15
2.4 Classification problems . . . 17
2.5 Majority vs Random class . . . 21
2.6 Nearest neighbour classifier . . . 22
2.7 Classification trees . . . 23
2.8 Choosing the best model . . . 27
2.9 Perceptron learning . . . 33
2.10 Clustering . . . 37
2.11 Suggested readings . . . 40

3 Metaheuristic Methods **41**
3.1 The big picture . . . 42
3.2 Visual insights . . . 44
3.3 Exhaustive search . . . 48
3.4 Random search . . . 53
3.5 An object-oriented approach . . . 54
3.6 Hill Climbing . . . 59
3.7 Random Walk . . . 70
3.8 Simulated Annealing . . . 71
3.9 Genetic Algorithms . . . 74
3.10 Fitness function . . . 79
3.11 Genetic operators . . . 80
3.12 Metaheuristics benchmarks . . . 88
3.13 Estimation of Distribution Algorithms . . . 92
3.14 Suggested readings . . . 96

CHAPTER 1



Introduction

WE ARE IN THE *Data Era*: almost every conceivable activity (human or otherwise) generates continuous streams of raw data originated from underlying structures, dynamics or complex patterns. Discovering such knowledge requires suitable algorithms and computation machinery to perform this task efficiently and automatically, yielding insights to decision–makers or managers about how to improve, grow or benefit from models derived from the rich information hidden in their data. The models may represent rules, groups, distributions or optimal regions of the data, explaining their nature and meaning. Knowledge about these models and how to apply them has got to be known recently as *Data Science*.

There are two clear knowledge disciplines overlapping in this modern approach to science: in the one hand Computer Science contributing the fields of Algorithmics and Computer Programming, and on the other hand Maths and Statistics, contributing fields such as Probability, Statistical Analysis, Optimisation and Linear Algebra. In fact, the intersection of these two avenues gave birth to sub-fields of Artificial Intelligence such as Machine Learning and Metaheuristics. The big novelty is the addition of a third avenue coming from the traditional science in the form of domain expertise, experimental studies, data collection and preparation, randomisation, hypothesis testing, etc.

Moreover, these fields are not newcomers to the artificial intelligence community. They have been around since many decades ago under different names: Pattern Recognition, Computational statistics, Data mining, Stochastic optimisation, Evolutionary computation, Bioinspired methods, etc.

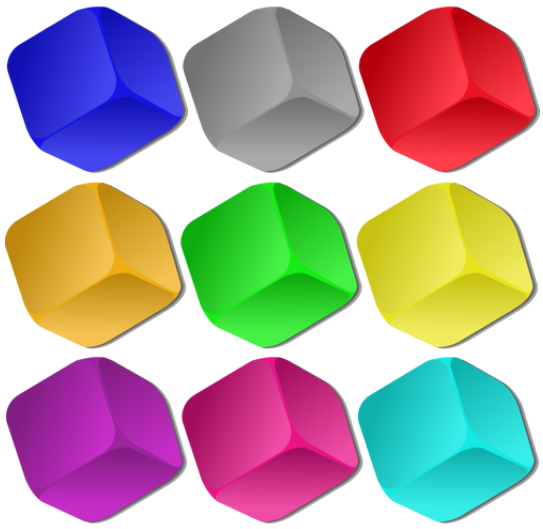
In recent years (since early 2015) we have seen the rise of Data Science as an emerging technology triggered mainly by the big advances in Deep Learning and Machine Learning. The technology is currently evolving to maturity, but it is still in its development, and moreover, is calling for more enthusiasts to grab the essentials of its concepts and tools so as to promote further advance in different scenarios of human activity.

The main motivation behind this notes, is precisely to provide an introductory level guide to graduate students from different areas of engineering to help them acquiring essential practical skills to enter a path of discovery in this novel scientific paradigm. With this aim in mind we have chosen the two (we believe) most significant branches of development in Data Science, namely Machine Learning and Metaheuristics, supported with one of the most popular and widespread programming languages nowadays, i.e Python. Well, without any further delays, we shall start our practical guided tour of “models for Data Scientists” in the next chapter.

The examples developed in this document were coded and tested in **Python v3.7**, using the **Anaconda** distribution and **Pycharm** IDE. For downloading and installing the latest versions, please visit:

```
www.python.org
www.anaconda.com
www.jetbrains.com/pycharm
```

CHAPTER 2



Machine Learning Models

THIS CHAPTER FOCUSES ON going through a variety of Machine Learning (ML) models useful in business and industrial applications; it will provide advice on concepts and pragmatic skills needed to embark on applied ML projects.

Before kicking-off, let us state our very own definition of Machine Learning, as a flavour of artificial intelligence enabling computer systems to learn from examples, data, and experience rather than to follow pre-programmed rules. In this way, "machines" (or more precisely, *algorithms*) are able to discover patterns, detect anomalies and adapt rules to perform complex tasks usually associated to human intelligence.

Recent years have seen an increasing usage of ML in common–day situations, such as image recognition systems used in medical diagnostics, video analysis systems in self-driving vehicles, voice recognition tools used by virtual personal assistants, and recommender systems deployed in online shops. As long as research progresses in this field, ML would become a disruptive technology in many areas of engineering, originating significant opportunities for social and economic development.

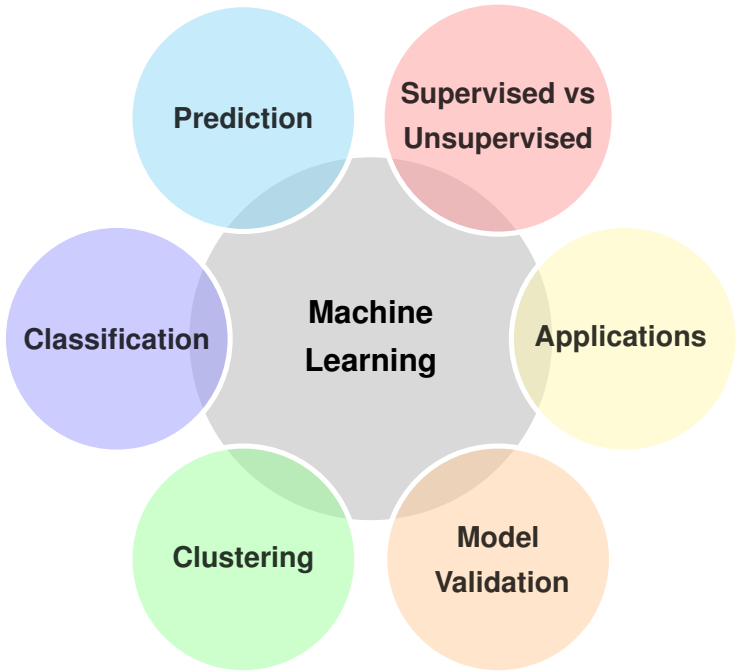
Example: EasyWine with Machine Learning

Suppose you are an entrepreneur launching EasyWine™, a new start-up focusing on a mobile–based social network for wine and spirits sales and distribution in the Bogota area. Your customers can buy, track, and even swap bottles and organise wine-tasting events.

On the basis of data you collect from your customers (profiles, purchases, location) plus external data such as weather, climate, traffic conditions, etc., Machine Learning can help your company to identify quite precisely sales campaigns, user profiles and promotions that can boost its revenue and get you ahead of competitors.

The big picture

So let's step-up one ladder in the staircase to Machine Learning. Here it is a big picture:



On the one hand Machine Learning considers mainly two models of learning: *Supervised* and *Unsupervised*. Broadly speaking, the first approach correspond in human learning to "instructional" education, whereas the second may be think of as "self-study". In other words, in supervised learning the machine is instructed with a set of previously expertly annotated (labelled) examples from which it has to discover patterns that explains their behaviour and moreover, that also identify newly unobserved examples in the future.

In contrast, the unsupervised learning scenario occurs when the machine is not instructed but just provided with a raw set of data samples from which it has to discover underlying, hidden concepts. These two approaches can be performed separately, or they can be intertwined during a particular Data Science project.

Supervised learning encompasses two main tasks: *classification* and *prediction*. In classification, the goal is to learn how to discriminate between examples labelled into two (binary classification) or more classes (multi-class classification). The machine therefore learns a *classification function* in the form of a mathematical expression or a set of rules.

On the other hand, the task of prediction attempts to anticipate the next event of a phenomenon based on the trend of past observations. In fact, classification sometimes is considered a type of prediction where the dependent variable is discrete (binary or categorical), whereas proper prediction is about forecasting a continuous–valued independent variable.

Now, regarding unsupervised learning, the main task is related to *clustering*. Here, the machine must learn the structure (arrangements, connections, order) underlying a sample of observations. Usually these observations are given without any further expert information; hence, the best the machine can do is to try to group together samples using an affinity criteria (the most similar according to a particular distance measure). Besides, sometimes the machine is provided with a few labelled samples and a lot of unlabelled samples; this mixed mode of learning has been called *Semi-supervised Learning*.

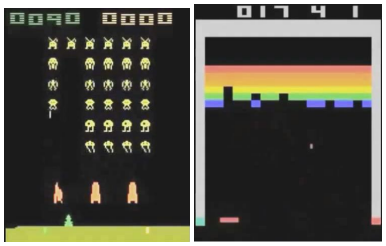
Another important issue in Machine Learning is to ensure the generalisation capabilities of the learner. That is, the machine must be able to recognise correctly unobserved data, so as to respond successfully in future situations. The tips and techniques used to achieve this goal are collectively known as *Model Validation*.

Lastly, ML of course is about real–life applications. So far we have mentioned a few, but nowadays the scope of applications are increasingly appearing in all segments of industry: banking, retail, medicine, education, environment, traffic, digital, real state, government... the list is limitless. The hype ML has gained in industry is partially credited to the recent success of *Deep Learning*, for example, the widely publicised milestone of DeepMind's machine that learned to play Atari games at an expert gamer level, or the recognition abilities of the Google's *Quick Draw* canvas engine capable of identifying handwritten doodles (even poor ones) with high accuracy.

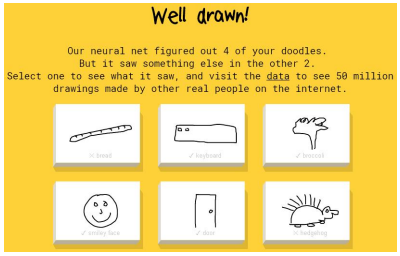
Examples of projects in *classification* are: a bank system to segment customers according to loan payment behaviour, a medical application to discriminate between healthy vs ill patients of an infectious disease, an advertisement server capable of displaying user-tailored ads in a social network.

Examples of application of *prediction* are: forecasting if a currency exchange rate will surge or drop as a consequence of political news, planning on-demand traffic routes according to weather condition and hour of day.

Examples of *clustering* are: the object recognition module of the video cameras of a self-driving car, an e-commerce retailer recommender system to suggest items for a customer based on his purchase, a credit card fraud alert monitoring application of a bank.



Screenshots of *DeepMind* playing *Space Invaders* and *Pong* as an expert gamer. See: `youtube.com/watch?v=W2CAghUiofY`



Googles's *QuickDraw* canvases. See: `quickdraw.withgoogle.com`

So let's start this tutorial with the basic tools needed to understand data and then to learn models on the data. As said before, the idea of the tutorial is to guide you through concepts and practical examples of increasing complexity as the exposition progresses. We shall start by exercising the visual exploration of the data.

Exploratory data analysis

In this section we shall introduce data visualisation tools by means of the following case-study.

A movie-recommender system: part 1

Let us assume you are the owner of a local cinema screen in your neighbourhood. You are a cinema–lover, so along with the obvious blockbuster screenings, you want your regulars to have the chance to watch the best movies in cinema history. So, you need to schedule assortments of similar award-winning movies according to recommendation given by movie-goers. Thus, this exercise is tailored towards building a movie–recommender system by using basic data–manipulation operations with the Python libraries `numpy` and `matplotlib`. The aim in the first part of this exercise is to conduct a exploratory analysis of the history of Oscar–winning movies.

For illustration purposes, we will work with a database containing information about the winners of the Oscar to the best movie from 1927 to 2018. Let's have a sneak peek into the dataset:

The dataset file can be downloaded from:
`https://goo.gl/MaqRKQ`

name	year	nominations	rating	duration	genre1	genre2	release
Shape of Water	2018	13	7.4	123	Fantasy	Romance	August
Moonlight	2017	8	7.5	111	Drama		November
Spotlight	2016	6	8.1	128	Crime	Drama	November
Birdman	2015	9	7.8	119	Comedy	Drama	November
12 Years Slave	2014	9	8.1	134	Biography	Drama	November
Argo	2013	7	7.8	120	Biography	Drama	October
The Artist	2012	10	8	100	Comedy	Drama	October
The King Speech	2011	12	8	118	Biography	Drama	December
The Hurt Locker	2010	9	7.6	131	Drama	History	July
(...)							

Dataset credit: Shehroz S. Khan @ U of Toronto

The following lines of code import the aforementioned Python libraries:

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

Now let's start–off by loading the data into a table:

```
# Data loading
movies = np.loadtxt('best-pictures.csv'
        , dtype={'names': ('name','year','nominations','rating','duration','genre1','
             genre2','release','synopsis')
        , 'formats': ('S30','u2','u1','f2','u2','S10','S10','S10','S255')}, delimiter='
             ,', skiprows=1)
# Show first rows of table
print('\n---------- First rows of table ----------')
print(movies[:][1:5])
```

```
Out[1]:

---------- First rows of table ----------
[(' Spotlight', 2015, 6, 8.1, 128, 'Crime', 'Drama', 'November', 'The true story of how the...')
 (' Birdman', 2014, 9, 7.8, 119, 'Comedy', 'Drama', 'November', 'Illustrated upon the progress...')
 (' 12 Years a Slave', 2013, 9, 8.1, 134, 'Biography', 'Drama', 'November', 'In the antebellum United...')
 (' Argo', 2012, 7, 7.8, 120, 'Biography', 'Drama', 'October', 'Acting under the cover of...')]
```

Next, using Python is straightforward to obtain basic descriptive statistics such as duration of the movies:

```
print('\n---------- Basic statistics ----------')
avg_duration = movies['duration'][:].mean()
std_duration = movies['duration'][:].std()
print('Duration average: {0:.0f}+-{1:.0f} mins'.format(avg_duration, std_duration))
```

```
Out[2]:

---------- Basic statistics ----------
Duration average: 138+-31 mins
```

Challenge 2.1

Compute descriptive statistics (maximum, minimum, average and standard deviation) of the number of nominations, user ratings and durations. Find those movies (and years) with maximum and minimum numbers.

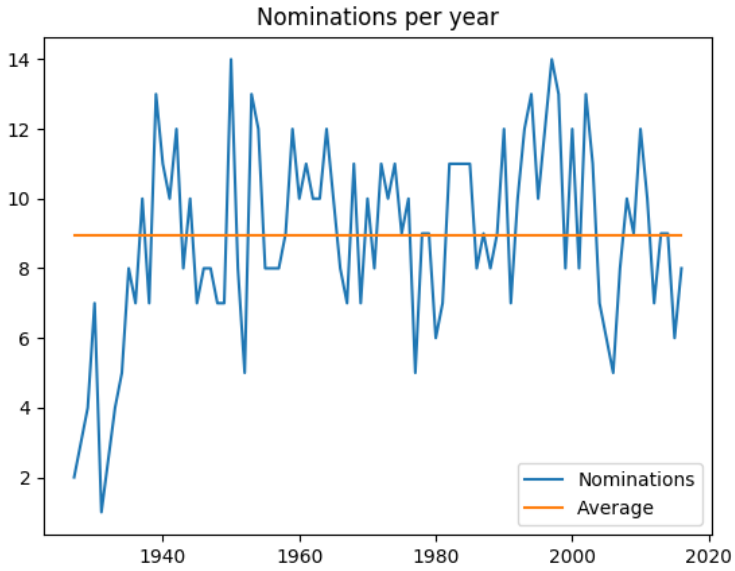
Challenge 2.2

Which are the two most frequent Oscar-winning genres? Which one is the least frequent?

Now let's see the number of nomination both in text and in a graphical plot:

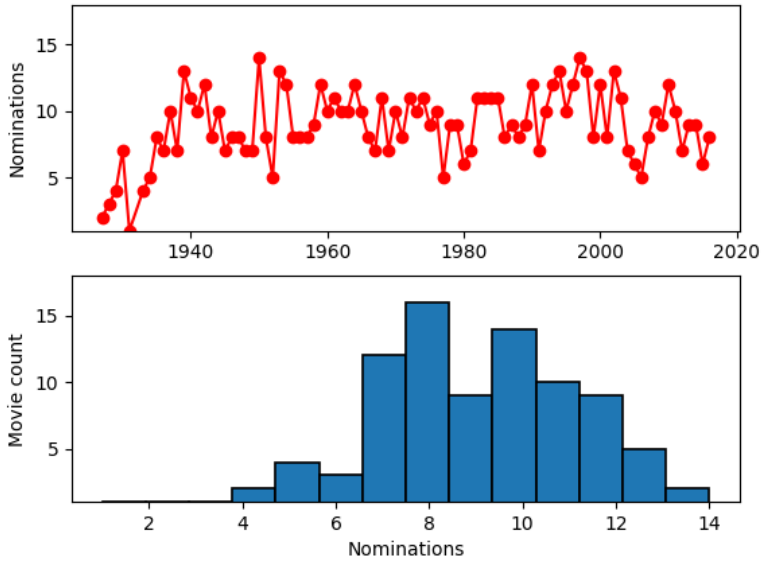
```
# Plot nominations per year
x = movies['year'][::-1]
y = movies['nominations'][::-1]
plt.plot(x, y)
avg_y = np.array([y.mean() for i in xrange(len(y))])
plt.plot(x, avg_y)
plt.legend(('Nominations', 'Average'), loc='lower right')
plt.title('Nominations per year')
plt.show()
print('\n---------- Nominations per year ----------')
print(x, y, sep='\n\n')
```

```
Out[3]:

---------- Nominations per year ----------
[1927 1928 1929 1930 1931 1933 1934 1935 1936 1937 1938 1939 1940 1941
 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955
 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 ...]

[ 2  3  4  7  1  4  5  8  7 10  7 13 11 10 12  8 10  7  8  8  7  7 14  8
  5 13 12  8  8  8  9 12 10 11 10 10 12 10  8  7 11  7 10  8 11 10 ...]
```



Let's do some additional visual exploration, this time the histogram of nominations history:

```
# Plot nominations trend vs histogram
fig, axs = plt.subplots(2, 1)
axs[0].plot(x, y, 'ro-')
axs[0].set_ylim([1, 18])
axs[1].hist(y, bins=14, edgecolor='black', linewidth=1.2)
axs[1].set_ylim([1, 18])
axs[1].set_xlabel('Nominations')
axs[1].set_ylabel('Movie_count')
plt.show()
```



Challenge 2.3

Show the historical trend and mean value of user-ratings and duration of movies along years, *both in the same plot* (you might need to figure out how to keep the range of values within the same scale).

Challenge 2.4

Plot the curve of average time length of Oscar–winning movies discriminated per genre.

Challenge 2.5

Plot the histogram of genre distribution.

NB: *Notice that a movie may belong to more than one genre.*

Now let's move forward to the concept of *valence*, which is related to the perceived emotion of a situation, or a sentiment polarity (goodness vs. badness). Here, a movie–goer may be interested in knowing the valence of a movie before going to watch it. One possible source to perform sentiment analysis is to look at the synopsis, which may indicate the attractiveness (positive valence) or aversiveness (negative valence) of a movie. A naive approach to obtain valence of a piece of text is by aggregating the valence of individual words within it. So, for this exercise you will need a valence dictionary of individual English words (valence is a number in the range $[-5, 5]$):

```
abandon         -2      backed          1       calm            2       damage  -3
abandoned       -2      backing         2       calmed          2       damages -3
abandons        -2      backs           1       calming         2       damn    -4
abducted        -2      bad             -3      calms           2       damned  -4
abduction       -2      badass          -3      cant stand      -3      damnit  -4
abductions      -2      badly           -3      cancel          -1      danger  -2

 (...)
```

The valence dictionary can be downloaded from: `https://goo.gl/s4qRMv`

In Python it is easy to tokenise (split in single words) a fragment of text:

```
text = "The␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog"
text.split()
```

```
Out[3]:

['The','quick','brown','fox','jumps','over','the','lazy','dog.']
```

It is also straightforward to load the valence dictionary:

```
valence = dict(map(lambda (k,v): (k,int(v)),
                   [ line.split('\t') for line in open("valence-dict.txt") ]))
print ("valence(\"good\")␣=␣" + str(valence["Good".lower()]))
print ("valence(\"bad\")␣=␣" + str(valence["Bad".lower()]))
```

```
Out[4]:

valence("Good") = 3
valence("Bad") = -3
```

Challenge 2.6

Compute the sentiment of each Oscar–winning movie as the aggregated valence of its respective synopsis. Add a column to the table with the name "Sentiment".

Hint: *You may want to try the* `np` *function* `c_` *to concatenate.*

Challenge 2.7

Show the 5 most attractive and 5 most aversive movies, including their posters.

Challenge 2.8

Plot the average, maximum and minimum sentiment per genre.

NB: *Notice that a movie may belong to more than one genre.*

A movie-recommender system: part 2

Now that you are familiar with the movie database, the second stage of this exercise is to build a mechanism to recommend movies to users. There are two approaches to this end (we shall use the first one):

- **Content-Based Recommender:** makes recommendations based on the description and attributes of similar items previously consumed by a user.
- **Collaborative Recommender:** makes recommendations based on collecting preferences from many users with similar tastes.

Notice that, per definition, we need to compute similarity between different items. For this purpose it is easier to work with numerical attributes only, but in our case we have two categorical attributes: genre and month of release. One way of converting these to numerical values is the so–called *one-hot encoding*, which maps the categorical features to a binary column in a one-of-K array. So, you can do something like this:

```
import numpy as np
import pandas as pd
# Data loading #
movies = np.loadtxt('best-pictures.csv'
        , dtype={'names': ('name','year','nominations','rating','duration','genre1','
             genre2','release','synopsis')
        , 'formats': ('U30','u2','u1','f2','u2','U10','U10','U10','U255')}, delimiter='
             ,', skiprows=1)
# Show first rows of table #
print('\n---------- Genre column ------')
print(movies["genre1"][0:5])
df = pd.DataFrame(movies[:]["genre1"])
onehot_genres = pd.get_dummies(df)
print("---------- One hot encoding of genre1 ----------")
print(onehot_genres[:6][:5])
genre1_matrix = onehot_genres.values
print("---------- genre1 as a matrix ----------")
print(genre1_matrix[:5])
```

```
Out[5]:

---------- genre1 column -----------
['Drama' 'Crime' 'Comedy' 'Biography' 'Biography' 'Comedy' 'Biography'
 'Drama' 'Drama' 'Crime' 'Crime' 'Drama' 'Drama' 'Adventure' 'Comedy'...]
---------- One hot encoding of genre1 -----------
   0_Action  0_Adventure  0_Biography  0_Comedy  0_Crime  0_Drama  0_Musical  \
0         0            0            0         0        0        1          0
1         0            0            0         0        1        0          0
2         0            0            0         1        0        0          0
3         0            0            1         0        0        0          0
(...)
---------- genre1 as a matrix -----------
[[0 0 0 0 0 1 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 1 0 0 0 0] ...]
```

Challenge 2.9

Modify the *movies* dataset to incorporate binary columns corresponding to the one–hot coding for genres.

NB: *Notice that a movie may belong to more than one genre.*

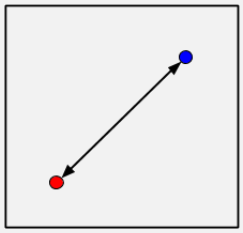
Challenge 2.10

Modify the *movies* dataset to also incorporate the one–hot coding for release month; additionally replace the synopsis column by its aggregated valence value. At this point your dataset must consists of a matrix with only numerical attributes (except its "name").

Distance metrics

Similarity indicates how much two data items are alike. In the context of ML similarity can be measured by means of the distance of vectors representing features of the items. The closer the distance the higher the similarity of two vectors can be (distance is usually measured in the range $[0,1]$). Some distance metrics used to measure similarity are: *Euclidean distance*, *Cosine similarity* and *Manhattan distance*.

Euclidean distance



Given two data items, $\mathbf{x}, \mathbf{z} \in \mathbb{R}^d$ i.e. $\mathbf{x} = (x_1, \ldots, x_d)$ and $\mathbf{z} = (z_1, \ldots, z_d)$, the Euclidean distance is obtained as:

$$D_E = \sqrt{\sum_{i=1}^{d} (x_i - z_i)^2}$$

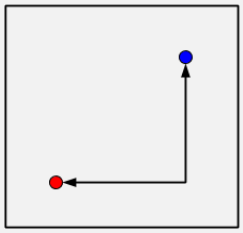
Or using `numpy`:

```
def euclidean_dist(x,z):
    return np.sqrt(np.sum((x-z)**2))
print(euclidean_dist([0,1 ],[1,0]))  # should return 1.4142135...
```

Challenge 2.11

Using all the attributes on your dataset and Euclidean distance, which is the most similar movie to "Titanic"? Which is the most similar to "Forrest Gump"? And which one is the most similar to "The Godfather"?

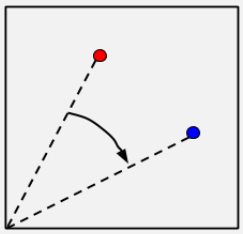
Manhattan distance



Given two data items, $\mathbf{x}, \mathbf{z} \in \mathbb{R}^d$ i.e. $\mathbf{x} = (x_1, \ldots, x_d)$ and $\mathbf{z} = (z_1, \ldots, z_d)$, the Manhattan distance is obtained as:

$$D_M = \sum_{i=1}^{d} |x_i - z_i|$$

Cosine similarity



Given two data items, $\mathbf{x}, \mathbf{z} \in \mathbb{R}^d$ i.e. $\mathbf{x} = (x_1, \ldots, x_d)$ and $\mathbf{z} = (z_1, \ldots, z_d)$, the the Cosine similarity is obtained as:

$$D_C = \frac{\mathbf{x}^T \mathbf{z}}{\|\mathbf{x}\| \|\mathbf{z}\|}$$

The Cosine similarity is not a proper metric since it does not comply with the triangle inequality property; hence, this measure gives a notion of how similar two items are, as opposed to how far are them from each other. For further details see: `https://en.wikipedia.org/wiki/Cosine_similarity`

Challenge 2.12

Same as **Challenge 2.11** but using Manhattan and Cosine distances. Is there any noticeable difference in the results?

Challenge 2.13

Write a recommender function that gets as input the name of a movie and returns as a result the list of the three most similar movies as found by each of the three distances.

Challenge 2.14

Repeat **Challenge 2.13**, but this time retrieve the 5 most similar movies with each method and return an aggregated list of titles with no duplicated items.

Challenge 2.15

Precompute a 89×89 matrix of movie similarity using Euclidean distance. Use such matrix to redefine your recommender function when finding the most similar movies instead of computing distances on-the-fly for each query. Demonstrate that the matrix–based method is faster in a large number of random user queries (say $k \geq 1000$ times).

Classification problems

In this exercise you will build up a classifier system to identify different classes of flowers, using the concepts of similarity.

The Iris flower data set consists of a collection of 50 samples from each of the three species of the Iris flower (**Iris setosa**, **Iris virginica** and **Iris versicolor**). Four attributes (a.k.a *features*) were measured from each sample: the length and the width of the sepals and petals, in centimetres.

Image credit: `http://en.wikipedia.org/wiki/Iris_flower_data_set`

The Iris dataset was introduced by the British statistician and biologist Ronald Fisher in his 1936 paper *The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis*, see:

`en.wikipedia.org/wiki/Ronald_Fisher`

This dataset is among a few others that is available in Python, so you can load it and get to grips with its contents very easily:

```
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
import pandas as pd

## Import the dataset ##
iris = datasets.load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['Species'] = iris['target']

## Scatter plot ##
X = iris.data[:, :2]  # we only take the first two features.
y = iris.target
plt.figure()
#plt.plot(X[:,0], X[:,1],'o')
plt.scatter(X[:,0], X[:,1], c=y)
plt.legend(("Setosa","Versicolour","Virginica"), loc="lower_right")
plt.show()
```

Alternatively you can fetch the data from the original UCI repository:

```
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Species']
iris = pandas.read_csv(url, names=names)
```

Most ML projects start–off by initially getting a broad idea of how the data looks like. We can begin by having a peek to the firsts iris samples:

```
## Peek at head rows using data frame ##
print('\n------- Iris data head rows ------')
print(df.head(8))
print('Dataset dimensions: ', df.shape)
```

```
Out[1]:

------- Iris data head rows ------
   sepal-length  sepal-width  petal-length  petal-width      Species
0           5.1          3.5           1.4          0.2  Iris-setosa
1           4.9          3.0           1.4          0.2  Iris-setosa
2           4.7          3.2           1.3          0.2  Iris-setosa
3           4.6          3.1           1.5          0.2  Iris-setosa
4           5.0          3.6           1.4          0.2  Iris-setosa
5           5.4          3.9           1.7          0.4  Iris-setosa
6           4.6          3.4           1.4          0.3  Iris-setosa
7           5.0          3.4           1.5          0.2  Iris-setosa
Dataset dimensions:  (150, 5)
```

Then we may list some descriptive statistics for this dataset:

```
## Basic statistics ##
print('\n------- Iris data statistics -------')
print(df.describe(),'\n')
print(df.groupby('Species').size())
```

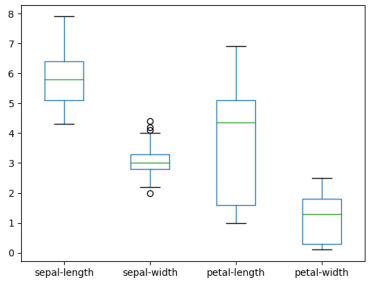
```
Out[2]:

       sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)     Species
count         150.000000        150.000000         150.000000        150.000000  150.000000
mean            5.843333          3.054000           3.758667          1.198667    1.000000
std             0.828066          0.433594           1.764420          0.763161    0.819232
min             4.300000          2.000000           1.000000          0.100000    0.000000
25%             5.100000          2.800000           1.600000          0.300000    0.000000
50%             5.800000          3.000000           4.350000          1.300000    1.000000
75%             6.400000          3.300000           5.100000          1.800000    2.000000
max             7.900000          4.400000           6.900000          2.500000    2.000000

Species
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
```

Challenge 2.16

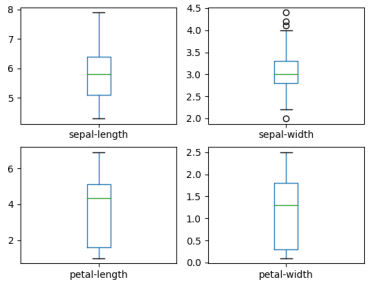
Report intra–group descriptive statistics (i.e. the above statistics for *Iris-setosa*, *Iris-versicolor* and *Iris-virginica* separately).

Nice, uh? Well, it gets better as we picture this information visually using *box–and–whisker* plots, either in single or separated panels per feature:

```
## Box and whisker plots ##
df.plot(kind='box', sharex=False, sharey=False)
df.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
```



or



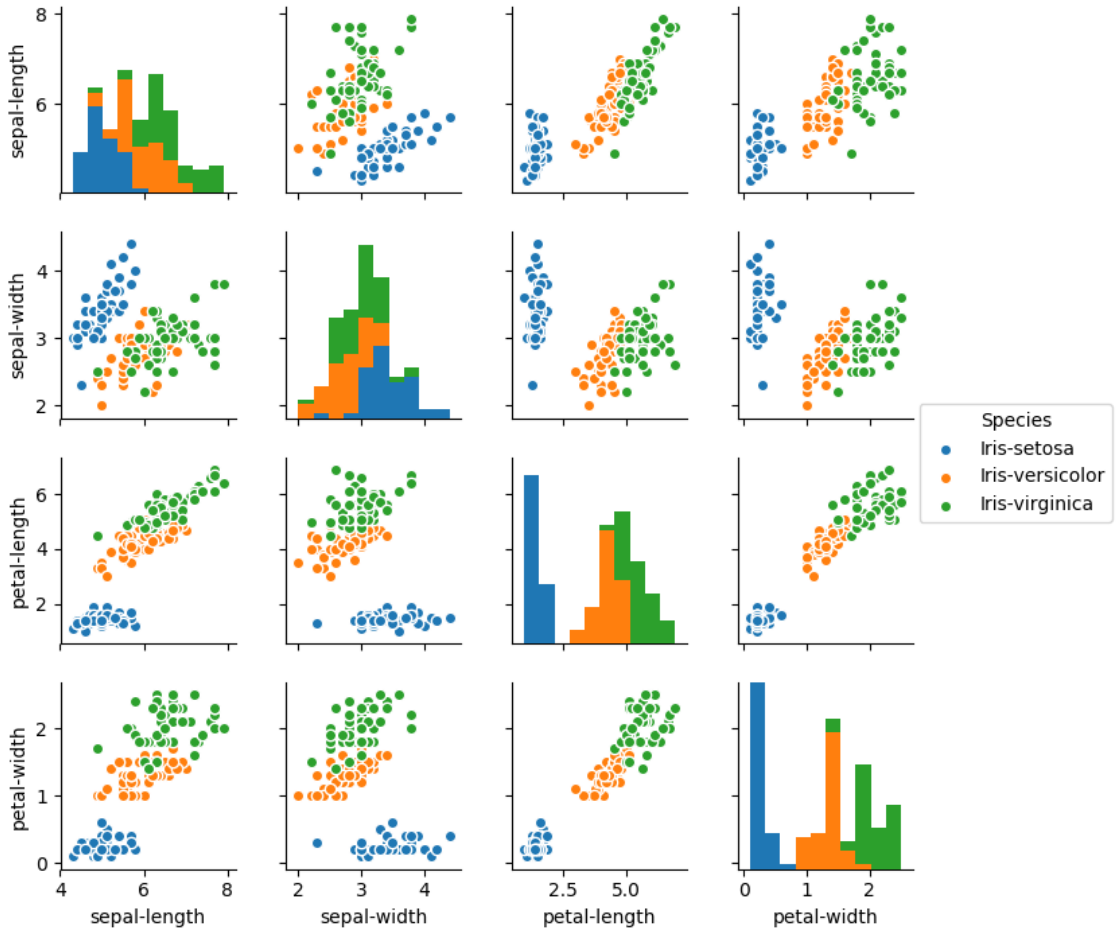
Challenge 2.17

Produce *box–and–whisker* plots per feature across species separately. Which feature seems to encompass a pattern (i.e. if-then-else rule) to accurately discriminate between species?

Hint: *Try using* `df.boxplot(by='Species', figsize=(8, 6))`

Furthermore, we can conduct multivariate analysis with a scatter plot matrix:

```
## Multivariate scatter plot ##
import seaborn as sb
sb.pairplot(df, hue="Species", size=1.8).fig.show()
```

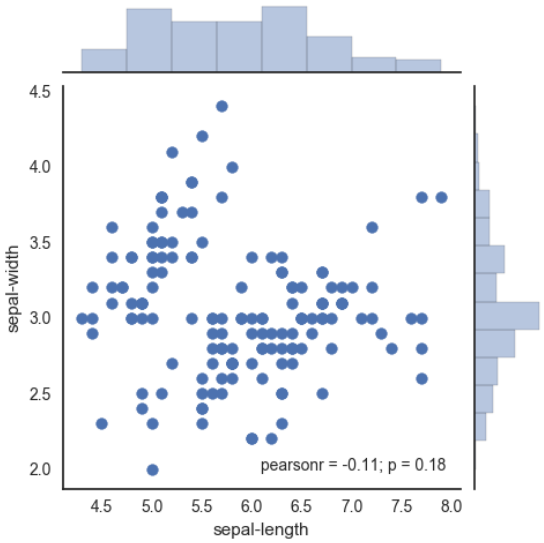


Challenge 2.18

Refine the decision rules inferred in **Challenge 2.17**, this time using the scatter plot matrix. Which class of species can be discriminated perfectly by means of which pair of features? What set of if-then-else rules are able to do it so?

Another useful tool are joint scatter plots, which combines bivariate, univariate and Pearson correlation statistic in one single plot:

```
sb.jointplot(x='sepal-length', y='sepal-width', data=df, size=5)
```



Challenge 2.19

Show a matrix of Pearson correlation coefficients between all-pairs of features. Which features are significantly most strongly correlated?

Now that we have a rough understanding of the dataset, let's build some classifiers to automatically discriminate between the three species. For this purpose we initially need to split the data into two subsets, one for learning the classifier (a.k.a **training** subset), and the other one for validating the effectiveness of the classifier (a.k.a **test** subset). This is because we want our classifier to discriminate future incoming observations, that is, data that it has not seen previously; thus, we set apart the **test** subset to evaluate such ability. So, let's call τ the proportion of samples assigned to training (usually, $\tau \geq 0.7$) and split the dataset with the following script:

```
## Split the dataset ##
tau = .7
train = df.sample(frac=tau, random_state=int(time.time()))
test = df.drop(train.index)
print('\n-------␣Training␣set␣distribution␣------')
print(train.groupby('Species').size())
print('\n-------␣Test␣set␣distribution␣------')
print(test.groupby('Species').size())
```

```
Out[3]:

------- Training set distribution ------
Species
Iris-setosa        40
Iris-versicolor    29
Iris-virginica     36

------- Test set distribution ------
Species
Iris-setosa        10
Iris-versicolor    21
Iris-virginica     14
```

Challenge 2.20

Notice that depending on the random training/test split, the resulting distributions may become unbalanced (there may be more samples from one species than for the others in each subset). Run the above script n times ($n \in \{10, 100, 1000\}$) and report average distributions to confirm this. Are the splits balanced in the average?

Challenge 2.21

Modify the script to ensure you will always get a balanced distribution of training/test samples across species in any single execution.

Majority vs Random class

Right. So now we have the dataset split into **training** and **test**, meaning we are all set and ready to go into our classification experiment! For the purpose of illustration, let's start with two very naive classifiers. The first one predicts all samples in the **test** set, assuming they come from the majority class in the **training** set (*Majority class classifier*). The second one predicts by just randomly "guessing" the labels of the **test** set (*Random guess classifier*):

```
## Use a "majority class" classifier ##
def majority_clf(Xtrn, Ytrn, Xtst):
    unique_labels, counts = np.unique(Ytrn, return_counts=True)
    majority_label = unique_labels[np.argmax(counts)]
    Ypred = np.repeat(majority_label, len(Xtst))
    return Ypred

## Use a "random guess" classifier ##
def random_clf(Xtrn, Ytrn, Xtst):
    unique_labels = np.unique(Ytrn)
    Ypred = np.random.choice(unique_labels, len(Xtst))
    return Ypred
```

Since we know the actual labels of the **test** set, we can measure the effectiveness of these (or any) classifier by computing its *accuracy* on the predicted labels, as the rate of correct predictions vs. total predictions:

$$acc_{\text{clf}} = \frac{\sum_i^N (y_i == \hat{y}_i)}{N},$$

where y_i are the actual labels on the **test** set, $\hat{y}_i$ are the predictions given by the classifier, and N is the number of samples on the **test** set.

```
## Assign data to arrays ##
Xtrn = train.drop(['Species'], axis=1).values
Ytrn = train['Species'].values
Xtst = test.drop(['Species'], axis=1).values
Ytst = test['Species'].values

## Test the classifiers ##
Ymaj = majority_clf(Xtrn, Ytrn, Xtst)
acc_maj = (float) (np.sum(np.equal(Ytst, Ymaj)))/len(Ytst)
print('\n------ Majority class classifier predictions on test set -----\n', Ytst ==
      Ymaj)
print('Accuracy rate of majority rule classifier: %.2f' % acc_maj)

Yrnd = random_clf(Xtrn, Ytrn, Xtst)
acc_rnd = (float) (np.sum(Ytst == Yrnd))/len(Ytst)
print('\n------ Random guess classifier predictions on test set -----\n', Ytst == Yrnd)
print('Accuracy rate of random classifier: %.2f' % acc_rnd)
```

```
Out[4]:

------- Majority class classifier predictions on test set ------
 [False False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False  True  True  True
  True  True  True  True  True  True  True  True  True]
Accuracy rate of majority rule classifier: 0.24

------- Random guess classifier predictions on test set ------
 [ True False False False False False False False False  True  True False
  True False False False  True False False False  True False  True False
 False False False False False  True False False False  True  True False
 False  True  True  True  True  True False False False]
Accuracy rate of random classifier: 0.33
```

Challenge 2.22

Notice that depending on the random sampling, the resulting accuracy rates may vary between different executions. Run the experiment 100 times and report average accuracy. Which classifier, *Majority class* or *Random guessing* is more accurate in average? Which is more consistent?

Challenge 2.23

What's the effect of varying the proportion of training rate? Repeat **Challenge 2.22** with different values of $\tau \in \{0.5, 0.7, 0.9, 0.99\}$. What can you say about accuracies in each case?

Nearest neighbour classifier

It is obvious that the two previous strategies perform pretty bad (a desired accuracy of good classifier should be above 90%). So, let's try to build two smarter classifiers. The first one shall use the rules you inferred from the plots produced in **Challenge 2.17** and **Challenge 2.18** (namely, *Rule based classifier)*. The second one shall predict the label of any instance in the **test** set, with the same label of the most similar instance in the **training** set (namely, *Nearest Neighbour or NN classifier*). This one can yet been refined to assign the label of the majority of the k nearest neighbours (namely, *k-NN classifier*).

```
## Use a "rule-based" classifier ##
def rule_clf(Xtrn, Ytrn, Xtst):
    ### fill-in with your code here to obtain YPred
    return Ypred

## Use a "k Nearest Neighbours" classifier ##
def kNN_clf(Xtrn, Ytrn, Xtst, k):
    ### fill-in with your code here to obtain YPred
    return Ypred
```

Challenge 2.24

Implement the *Rule–based* and *1-NN* (Euclidean–based) classifiers above mentioned. What can you say about their average accuracies? To what extent they improve with respect to the other naive classifiers (*Majority class* and *Random guessing*)? All–in–all, which classifier will you recommend as better suited to discriminate the Iris–flower dataset? Consider using plots to support your answers.

NB. *For 1-NN consider the distances measures you studied in* ***Section 2.3***

Challenge 2.25

Conduct a deeper study of the $k-NN$ classifier, using different combinations of the following running parameters: $k \in \{1,3,5,11,21\}$, $\tau \in \{0.5,0.7,0.9,0.95,0.99\}$, Euclidean, Manhattan and Cosine distances, and balanced/unbalanced splits. What can you say about the general performance of this classifier in terms of both, effectiveness (i.e. accuracy) and efficiency (i.e runtime)? Use accuracy and runtime plots or tables as evidence.

NB1. *Reporting average results of at least 30 repetitions of your experiments, instead of single executions, provides more solid evidence to support your answers.*

NB2. *Consider using the following trick to collect runtimes (e.g. for the* Majority class *classifier):*

```
import time
t = time.time()
Ymaj = majority\_clf(Xtrn, Ytrn, Xtst)
print("Time elapsed: %.4f secs" \% (time.time() - t))
```

Classification trees

A classification tree can be think of a hierarchical splitting of the data by applying a series of "decision tests" over a subset of different features in the dataset. The decision test on each node of the tree produces a partition of the data that is recursively split in smaller datasets as the tree grows in depth, until no more partitions can be made when the data remaining in the leaves of the tree are of the same class.

The `Auto-MPG` dataset contains data about attributes of different car brands, including gas consumption, manufacturer, cylinders, etc. The dataset can be used for example, to discriminate between efficient vs high-consuming vehicles. For more information and download see: `https://archive.ics.uci.edu/ml/datasets/auto+mpg`

So, the idea of a tree classifier is to perform a decision test on an attribute and observe how good the data is classified according to a label variable. Let's take as an example a dataset of vehicles manufacturing information, and say we are interested in classifying the car brands according to their fuel consumption in terms of miles per gallon (mpg). Thus, firstly we shall load the data and add a label indicating the car is efficient or not (fuel consumption $\geq$ 23 mpg):

```
import pandas as pd
import numpy as np
import time

## Load Auto-MPG dataset ##
filename = 'auto-mpg.data'
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower',
                'weight', 'acceleration', 'year', 'origin', 'name']
df = pd.read_csv(filename, delim_whitespace=True, names=column_names)

print("Count of efficient cars: " + str(df[df.mpg >= 23].shape[0]))
print("Count of inefficient cars: " + str(df[df.mpg < 23].shape[0])+"\n")

## Add a class label column ##
df["label"] = (df.mpg >= 23)

## Have a peek at the data ##
pd.set_option('display.width', 320)
print(df.head(20))
```

Using the "efficiency" label we can see the dataset is fairly balanced (201 vs. 197 samples):

```
Out[5]:

Count of efficient cars: 201
Count of inefficient cars: 197

     mpg  cylinders  displacement horsepower  weight  acceleration  year  origin                       name  label
0   18.0          8         307.0      130.0  3504.0          12.0    70       1  chevrolet chevelle malibu  False
1   15.0          8         350.0      165.0  3693.0          11.5    70       1          buick skylark 320  False
2   18.0          8         318.0      150.0  3436.0          11.0    70       1         plymouth satellite  False
3   16.0          8         304.0      150.0  3433.0          12.0    70       1              amc rebel sst  False
4   17.0          8         302.0      140.0  3449.0          10.5    70       1                ford torino  False
5   15.0          8         429.0      198.0  4341.0          10.0    70       1           ford galaxie 500  False
6   14.0          8         454.0      220.0  4354.0           9.0    70       1           chevrolet impala  False
7   14.0          8         440.0      215.0  4312.0           8.5    70       1          plymouth fury iii  False
8   14.0          8         455.0      225.0  4425.0          10.0    70       1           pontiac catalina  False
9   15.0          8         390.0      190.0  3850.0           8.5    70       1         amc ambassador dpl  False
10  15.0          8         383.0      170.0  3563.0          10.0    70       1        dodge challenger se  False
11  14.0          8         340.0      160.0  3609.0           8.0    70       1         plymouth 'cuda 340  False
12  15.0          8         400.0      150.0  3761.0           9.5    70       1      chevrolet monte carlo  False
13  14.0          8         455.0      225.0  3086.0          10.0    70       1    buick estate wagon (sw)  False
14  24.0          4         113.0       95.00 2372.0          15.0    70       3      toyota corona mark ii   True
15  22.0          6         198.0       95.00 2833.0          15.5    70       1           plymouth duster  False
16  18.0          6         199.0       97.00 2774.0          15.5    70       1                amc hornet  False
17  21.0          6         200.0       85.00 2587.0          16.0    70       1             ford maverick  False
18  27.0          4          97.0       88.00 2130.0          14.5    70       3              datsun pl510   True
19  26.0          4          97.0       46.00 1835.0          20.5    70       2  volkswagen 1131 deluxe sedan   True
```

Now see how our *Majority class* rule performs:

```
## Define a majority class classifier ##
def majority_clf(Xtrn, Ytrn, Xtst):
    labels, counts = np.unique(Ytrn, return_counts=True)
    majority_label = labels[np.argmax(counts)]
    Ypred = np.repeat(majority_label, len(Xtst))
    return Ypred

## Test the classifier ##
Ymaj = majority_clf(Xtrn, Ytrn, Xtst)
acc_maj = (float)(np.sum(np.equal(Ytst, Ymaj))) / len(Ytst)
# print('\n------ Majority class classifier predictions on test set -----\n', Ytst ==
    Ymaj)
print('Accuracy rate of majority rule classifier: %.2f' % acc_maj)
```

As expected (because the class distribution is balanced), the prediction accuracy is pretty much like guessing randomly with a coin flip (50%):

```
Out[6]:

Accuracy rate of majority rule classifier: 0.51
```

Would it be possible to improve if we first condition the prediction on the value of certain attribute, and then predict the majority class of the resulting distribution? For example, let's count how cars quantities are distributed according to label when testing the attribute *maker* and then *cylinders* (the `pandas` function `crosstab()` can help us):

```
## Count conditioned class distributions ##
print('\n------ Class distribution conditioned on \'maker\' (i.e. \'origin\') -----')
print(pd.crosstab(index=df["origin"][:], columns=df["label"]))

print('\n------ Class distribution conditioned on \'cylinders\' -----')
print(pd.crosstab(index=df["cylinders"][:], columns=df["label"]))
```

```
Out[7]:

------ Class distribution conditioned on 'maker' (i.e. 'origin') -----
label   False  True
origin
1         174     75
2          14     56
3           9     70

------ Class distribution conditioned on 'cylinders' -----
label      False  True
cylinders
3              3      1
4             20    184
5              1      2
6             73     11
8            100      3
```

Notice that these tests in fact, render our original problem into classification trees such as those shown below:

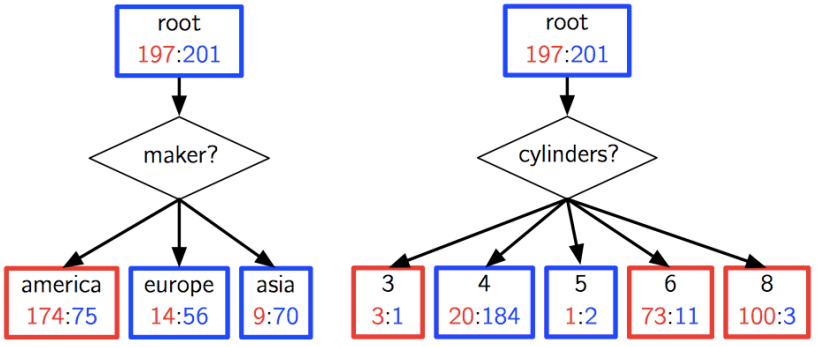
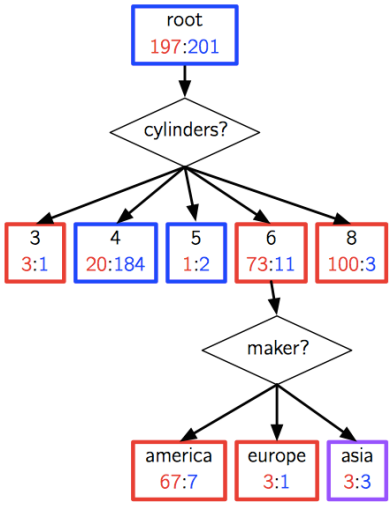


Image credit: Sham M Kakade @ cs.washington.edu

Furthermore, it is possible of course to apply an additional test on each of the categories resulting from the first test in order to obtain a more specific class distribution in lower levels of the tree (see the example next).



(c)

Image credit: Sham M Kakade @ cs.washington.edu

Challenge 2.26

Implement three different *Rule–based* classifiers (i.e. `if-then-else` rules) for the `Auto-MPG` dataset, using classification trees (a), (b) and (c), and then run experiments varying $\tau \in \{0.5, 0.8, 0.95\}$. What can you say about their average accuracies? To what extent they improve with respect to the *Majority class* classifier?

NB. *Your classifier should implement one rule per each test in the tree. For example,*

Tree (a) may look something like:

```
def treeA_clf (Xtrn, Ytrn, Xtst):
    Ypred = np.repeat(True, len(Xtst))
    for i in xrange(len(Xtst)):
        if Xtst[i, 7] == 1:     # American
            Ypred[i] = False
        elif Xtst[i, 7] == 2:   # European
            Ypred[i] = True
        else:                   # Asian
            Ypred[i] = True
    return Ypred
```

These trees can be learned automatically from data, using ML algorithms. Some of these algorithms are implemented in the `sckit-learn` library:

```
from sklearn-datasets import load_iris
iris = load_iris()
test_ids = [0, 50, 100]

## Split the data ##
Xtrn = np.delete(iris.data, test_ids, axis=0)
Ytrn = np.delete(iris.target, test_ids)
Xtst = iris.data[test_ids]
Ytst = iris.target[test_ids]

## Train and test a tree classifier ##
tree_clf = tree.DecisionTreeClassifier(max_depth=4)
tree_clf.fit(Xtrn, Ytrn)
print Ytst
print tree_clf.predict(Xtst)
```

Challenge 2.27

Train and test four different *Tree classifiers* for the `Auto-MPG` dataset, using different `max_depth` $\in \{1,2,3,4\}$ and $\tau \in \{0.5,0.9\}$. What can you say about their average accuracies? According to these results, which is the better *Tree* classifier?

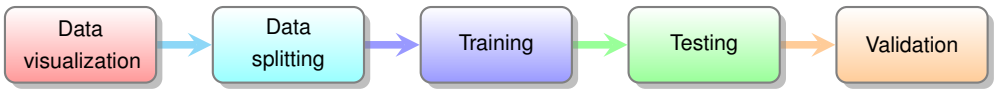
Choosing the best model

Model selection

Model selection is the task of selecting a statistical model from a set of candidate models, given data. This task involves the design of experiments to choose the best parameters of a given model. Given candidate models of similar predictive or explanatory power, the simplest model is most likely to be the best choice (Occam's razor).

Source: Wikipedia

So model selection refers to the process of choosing between different ML algorithms (e.g. k-NN, Classification Trees) or choosing between different hyperparameters or sets of features for the same ML algorithm (e.g. the value of k or `max_depth`). Model selection usually involves the following considerations: accuracy, speed, scalability, simplicity, interpretability. All these criteria can be assessed by means of the following protocol:



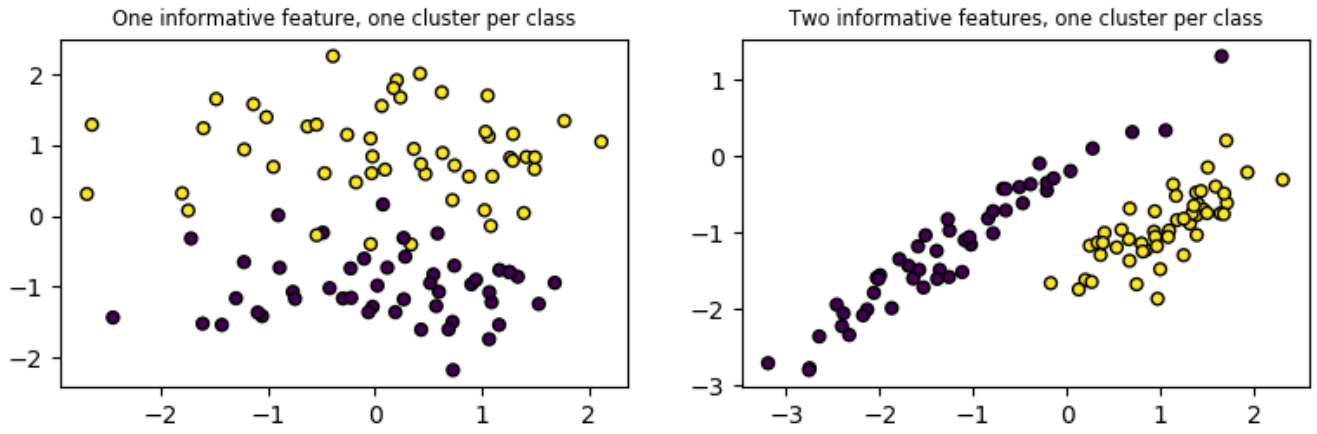
Next, we illustrate the typical steps followed in a task of model selection:

1. Create and visualise a synthetic dataset for a classification problem
2. Split the dataset into training and testing.
3. Train and test two algorithms: k-NN and Tree Classifier
4. Choose the best model

Ok, so let's start with ❶: Create and visualise a synthetic dataset...

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification

## Artificial data generation ##
dataset == "One informative, uniform"
X, Y = make_classification(n_samples=1000, n_features=2, n_informative=1, n_redundant
     =0, n_repeated=0, n_clusters_per_class=1)

## Scatter plot of the data ##
plt.figure()
plt.title(dataset)
plt.scatter(X[:,0], X[:,1], marker="o", c=Y, s=50, edgecolor="k")
plt.show()
```

... which produces the following output:



There are many artificial data generators in `sklearn`. Let's try a few ones:

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification, make_blobs,
    make_circles, make_moons, make_gaussian_quantiles
## Artificial data generation ##
dataset = "One informative, uniform"
#dataset = "Two informative, uniform"
#dataset = "Two informative, mixture"
#dataset = "Two informative, blobs"
#dataset = "Two informative, circles"
#dataset = "Two informative, moons"
#dataset = "Two informative, gaussians"

if (dataset == "One informative, uniform"):
    X, Y = make_classification(n_samples=1000, n_features=2,
        n_informative=1, n_redundant=0, n_repeated=0,
        n_clusters_per_class=1)
elif (dataset == "Two informative, uniform"):
    X, Y = make_classification(n_samples=1000, n_features=2,
        n_informative=2, n_redundant=0, n_repeated=0,
        n_clusters_per_class=1)
elif (dataset == "Two informative, mixture"):
    X, Y = make_classification(n_samples=1000, n_features=2,
        n_informative=2, n_redundant=0, n_repeated=0,
        n_clusters_per_class=2)
elif (dataset == "Two informative, blobs"):
    X, Y = make_blobs(n_samples=1000, n_features=2, centers=5)
elif (dataset == "Two informative, circles"):
    X, Y = make_circles(n_samples=1000, noise=.07, factor=.4)
elif (dataset == "Two informative, moons"):
    X, Y = make_moons(n_samples=1000, noise=.07)
elif (dataset == "Two informative, gaussians"):
    X, Y = make_gaussian_quantiles(n_samples=1000, n_classes=4, cov=.2)
```

Now, moving on to the next step: ❷ Split the dataset, there is a convenient way of randomly splitting the data using the `model_selection` module from `sklearn`:

```
import numpy as np
from sklearn.model_selection import train_test_split
X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]

X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.33, random_state=42)
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_test
[1, 4]
>>>
>>> train_test_split(y, shuffle=False)
[[0, 1, 2], [3, 4]]
```

The subsequent step is: ❸ Train and test the classifiers. So let's do this with the now familiar *Nearest Neighbour classifier* (k-NN), only that this time we will use the implementation available in `sklearn.tree`:

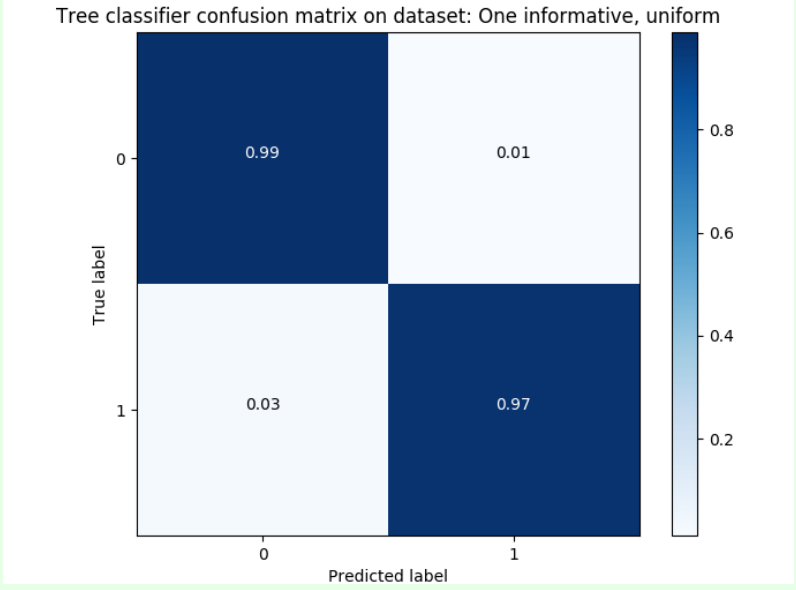
```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix, classification_report
import time
import itertools

# Generate artificial data
dataset = "One informative, uniform"
X, Y = make_classification(n_samples=1000, n_features=2, n_informative=1, n_redundant
    =0,n_repeated=0, n_clusters_per_class=1)

# Scatter plot of the data
plt.figure()
plt.title(dataset)
plt.scatter(X[:,0], X[:,1], marker="o", c=Y, s=50, edgecolor="k")
plt.show()

# Split the dataset
X_trn, X_tst, Y_trn, Y_tst = train_test_split(X, Y, test_size=0.8, random_state=int(
    time.time()))
print(X_trn.shape)

# Train and test k-NN classifier
clf = KNeighborsClassifier(n_neighbors=1)
clf.fit(X_trn, Y_trn)
acc_trn = clf.score(X_trn, Y_trn)
acc_tst = clf.score(X_tst, Y_tst)
print("kNN accuracy on training: %.2f " % acc_trn)
print("kNN accuracy on test: %.2f " % acc_tst)

Y_pred = clf.predict(X_tst)
tn, fp, fn, tp = confusion_matrix(Y_tst, Y_pred).ravel()
print("kNN sensitivity: %.2f " % float(tp/(tp+fn)))
print("kNN specificity: %.2f " % float(tn/(tn+fp)))
print("kNN confusion matrix: \n %s" % confusion_matrix(Y_tst, Y_pred))
print("kNN classification report (k=1): \n %s" % classification_report(Y_tst, Y_pred))
```

```
Out[8]:

(200, 2)
kNN accuracy on training: 1.00
kNN accuracy on test: 0.98
kNN sensitivity: 1.00
kNN specificity: 0.97
kNN confusion matrix:
 [[394  14]
 [  1 391]]
kNN classification report (k=1):
             precision    recall  f1-score   support

          0       1.00      0.97      0.98       408
          1       0.97      1.00      0.98       392

avg / total       0.98      0.98      0.98       800
```

You can also try a *Decision Tree classifier* (including a fancy confusion matrix visualisation):

```
# Train and test Tree classifier
clf = DecisionTreeClassifier(max_depth=2)
clf.fit(X_trn, Y_trn)
acc_trn = clf.score(X_trn, Y_trn)
acc_tst = clf.score(X_tst, Y_tst)
print("Tree␣accuracy␣on␣training:␣%.2f␣" % acc_trn)
print("Tree␣accuracy␣on␣test:␣%.2f␣" % acc_tst)

Y_pred = clf.predict(X_tst)
tn, fp, fn, tp = confusion_matrix(Y_tst, Y_pred).ravel()
print("Tree␣sensitivity:␣%.2f␣" % float(tp/(tp+fn)))
print("Tree␣specificity:␣%.2f␣" % float(tn/(tn+fp)))
print("Tree␣confusion␣matrix:␣\n␣%s" % confusion_matrix(Y_tst, Y_pred))
print("Tree␣classification␣report␣(max_depth=2):␣\n␣%s" %
    classification_report(Y_tst, Y_pred))

## Fancy CM visualisation ##
cm = confusion_matrix(Y_tst, Y_pred)
cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
print("Tree␣normalised␣confusion␣matrix␣(max_depth=2):␣\n␣%s" % cm)

plt.figure()
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title("Tree␣classifier␣confusion␣matrix␣on␣dataset:␣%s" % dataset)
plt.colorbar()
classes = np.unique(Y)
tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes, rotation=0)
plt.yticks(tick_marks, classes)
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], '.2f'), horizontalalignment="center"
        ,
             color="white" if cm[i, j] > thresh else "black")
plt.tight_layout()
plt.ylabel('True␣label')
plt.xlabel('Predicted␣label')
plt.show()
```

Out[9]:



Now it is time for some hands–on exercising of your model selection skills.

Challenge 2.28

Explore and understand your chosen synthetic dataset. You now know an essential toolbox of `Python` commands to do this: provide descriptive statistics, scatter plots, box and whisker plots, multivariate scatter plots, distribution histogram, correlation coefficients, etc.

NB. *Generate 3 dataset versions by varying the parameters of the chosen method:* `n_classes`, `n_features`, `n_informative`, `n_clusters_per_class`, `centers`, `factor`.

Challenge 2.29

Train and test some classifiers. Consider at least 3 classifiers: k-*NN, Decision Tree* and *Majority Class*. Report the classification score of each model on each variation of your dataset using different metrics: accuracy, TP, TN, FP, FN, specificity, sensitivity, confusion matrix, etc. Remember that the more creative and more visual are the more effective reports.

NB. *Perform hyperparameter evaluation, i.e, try different values for* `n_neighbors`, `max_depth`, *etc.*

Hint: *You may use* `sklearn.dummy.DummyClassfier()` *for the* Majority Class *classifier.*

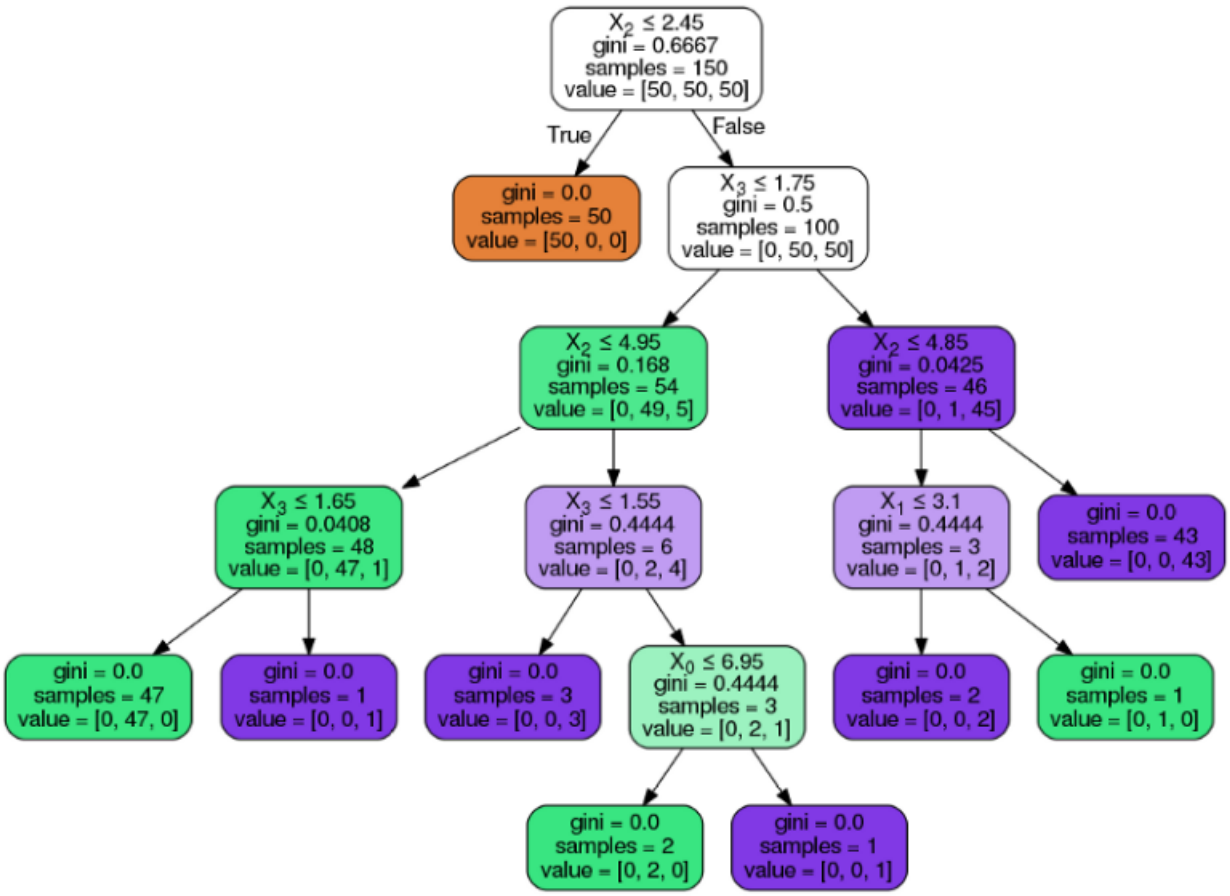
Challenge 2.30

Select a model. Based on the previous experiments, decide the fittest model for your dataset variations. Besides models' classification score, also provide evidence of their runtime performance.

On a different note, it would be interesting to be able to visualise the learned tree classifiers. Let's see how we can do it.

```
## Tree visualisation ##
import sklearn.datasets as datasets
import pandas as pd
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
from pydotplus import graph_from_dot_data

iris = datasets.load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
labels = iris.target

clf = DecisionTreeClassifier()
clf.fit(df, labels)

dot_data = StringIO()
export_graphviz(clf, out_file=dot_data)
graph = graph_from_dot_data(dot_data.get_value())

## Caution: GraphViz software is required. If not in PATH use the following 2 lines
##import os
##os.environ["PATH"] += os.pathsep + "C:\\Downloads\\graphviz-2.38\\release\\bin"

tree.write_pdf("tree.pdf")
```

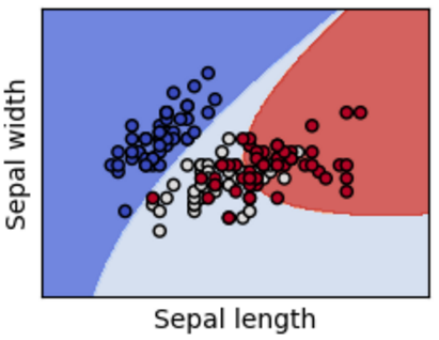
Now you can open the .pdf file and see how your tree looks like:



How about visualising what my classifier is doing? It is possible to see the discrimination function (or discrimination boundaries for that effect). Let's try it!

```
## Discrimination space ##
import numpy as np
import time
X = iris.data[:, 2:4]  # Classify using petal length and width
Y = iris.target

#clf = KNeighborsClassifier(n_neighbors=3)
clf = DecisionTreeClassifier(max_depth=5)
clf.fit(X,Y)

X1 = X[:,0]
X2 = X[:,1]
h = .1
xx, yy = np.meshgrid(np.arange(X1.min()-.5, X1.max()+.5, h),
                     np.arange(X2.min() - .5, X2.max() + .5, h))
zz = clf.predict(np.c_[xx.ravel(), yy.ravel()])
zz = zz.reshape(xx.shape)

plt.figure()
plt.contourf(xx, yy, zz, cmap=plt.cm.coolwarm)
plt.scatter(X1, X2, c=Y, edgecolors="k")
plt.show()
```

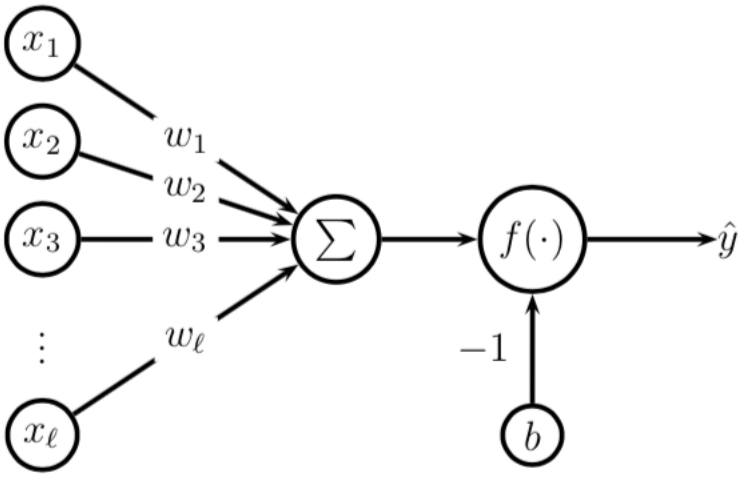
Out[10]:



As you can see, the input space was divided in three non–overlapping regions whose boundaries were learned by the machine during its training stage (here the blue, red and grey regions correspond to the three species of Iris). We remark that each learned model with different hyper–parameters would yield a different discrimination boundary.

Perceptron learning

The *Perceptron* was conceived as a metaphor of the operation of the brain to recognise patterns, or to be more precise, of a single neuron in the brain. The apparatus used by the *Perceptron* was the simple neural circuit illustrated below: a set of input cells conducting electrical impulses to a central processing unit (neuron) that triggers a signal when the total stimuli rise above certain threshold. The inputs are propagated through synaptic weights. Thus, depending on the inputs, weights and threshold, the neuron fires or stays quiet, i.e. it outputs a binary signal.

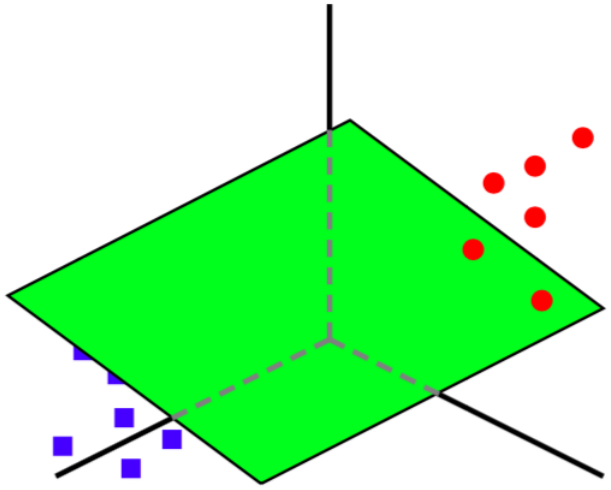


The mathematical formalism of this abstraction is the following:

$$f(v) = \begin{cases} 1 & v > 0 \\ 0 & v \leq 0 \end{cases},$$

where the potential of the neuron is denoted $v = \sum_{i=1}^{n} w_i x_i - b$. Here, the parameters $\{w_i\}$ correspond to the coefficients of a discrimination function in the input space given by a dataset $D = \{(\mathbf{x}_t, y_t)\}_{t=1,2,\ldots}$. The prediction given by the *Perceptron* would be the sign of the potential, $y_t = \texttt{sign}(v)$.

A geometric interpretation of the parameters $\mathbf{w} = \{w_i\}$ correspond to a hyperplane in $\mathbb{R}^\ell$, separating the space in to halves: one where the positive-labeled samples lie and the other one for the negative-labeled. For example, in the illustration below, a Perceptron linearly separates a dichotomy in $\mathbb{R}^3$; here, binary labels are shown in red ($y_t = +1$) or blue ($y_t = -1$). A consistent discriminant hyperplane is shown in green, which separates a dichotomy in such space.



The training stage of the *Perceptron* adjusts the hyperplane $\mathbf{w}$ every time an input data item $\mathbf{x}$ is wrongly classified $(y_t \neq \hat{y}_t)$. The learning rule is very simple, it simply shifts the hyperplane towards said misclassified input, i.e. $\mathbf{w} \leftarrow \mathbf{w} + y_t \mathbf{x}_t$. The algorithm is depicted below:

Algorithm 1: *Perceptron*

Input: A stream of patterns $\{(\mathbf{x}_t, y_t)\}_{t=1,2,\ldots}$
Output: A linear discriminant $\mathbf{w}$
$\mathbf{w} \leftarrow \mathbf{0}$
for $t = 1, 2, \ldots$ **do**
  $\hat{y}_t \leftarrow \texttt{sign}(\mathbf{w}^T \mathbf{x}_t)$
  if $(y_t \neq \hat{y}_t)$ **then**
    $\mathbf{w} \leftarrow \mathbf{w} + y_t \mathbf{x}_t$
  if convergence test met, stop.

The implementation of the algorithm in Python is listed below.

```
from random import choice
from numpy import array, dot, random, sign, arange, meshgrid, c_, ones, mean
import matplotlib.pyplot as plt
import time
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier

## Training data ##
# Comment/Uncomment for AND gate
# training_data = [ (array([0,0,1]), -1),
#                   (array([0,1,1]), 1),
#                   (array([1,0,1]), 1),
#                   (array([1,1,1]), 1), ]

# Comment/Uncomment XOR gate
training_data = [ (array([0,0,1]), -1),
                  (array([0,1,1]), 1),
                  (array([1,0,1]), 1),
                  (array([1,1,1]), -1), ]
## Initialize Perceptron weights #
random.seed(int(time.time()))
w = random.rand(3)

## Set algorithm parameters ##
errors = []
n = 100
eta = 0.1                          # Learning rate controlling the step of the update

## Train the classifier ##
for i in range(n):
    xi, yi = training_data[i % len(training_data)]     # Get one data point at a time
    yi_bar = sign(dot(w, xi))      # Compute the current prediction
    if (yi_bar*yi < 0):
        w = w + eta*(yi*xi)        # Update rule
        errors.append(yi_bar)      # Record errors
    else:
        errors.append(0)           # Record errors

## Test the classifier ##
print("Perceptron weights: ", w)
print("Perceptron test:")
for i in range(len(training_data)):
    xi, yi = training_data[i]      # Get one data point at a time
    yi_bar = sign(dot(w, xi))      # Compute prediction
    print(xi, yi, yi_bar)
```

Again, we can plot the learned classifier as follows:

```
## Plot the errors per iteration ##
plt.plot(errors)
plt.show()

## Plot the input space and decision boundary ##
h = .01
xx1, xx2 = meshgrid(arange(-0.5, 1.5, h),
                    arange(-0.5, 1.5, h))

yy = sign(dot(w, c_[xx1.ravel(), xx2.ravel(), ones(xx1.shape, dtype=int).ravel()].T))
yy = yy.reshape(xx1.shape)

plt.figure()
plt.contourf(xx1, xx2, yy, cmap=plt.cm.coolwarm)

## Plot the dataset ##
for X, y in training_data:
    plt.scatter(X[0], X[1], c="yellow" if y==1 else "white", marker="o" if y==1 else "s
        ")
plt.show()
```

Challenge 2.31

Repeat **Challenge 2.27** this time using the *Perceptron* classifier.

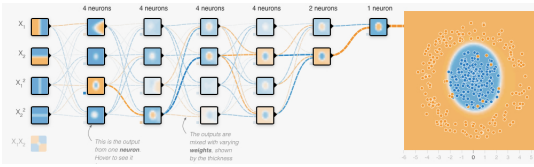
Alternatively, you can always use the implementation of the *Perceptron* and *Multi-layer Perceptron* found in `scikit-learn`:

```
## Let's try with the Iris dataset ##
iris = load_iris()
X = iris.data
y = iris.target
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=.3)

## Train and test a Single Layer ## Perceptron
clf = Perceptron()
clf.fit(Xtrain, ytrain)
ypred = clf.predict(Xtest)
print("Perceptron test accuracy: %.2f" % mean(ypred == ytest))

## Train and test a Multilayer Perceptron ##
clf = MLPClassifier(verbose=0, random_state=0, max_iter=100,)
clf.fit(Xtrain, ytrain)
ypred = clf.predict(Xtest)
print("MLP classifier test accuracy: %.2f" % mean(ypred == ytest))
```

Challenge 2.32

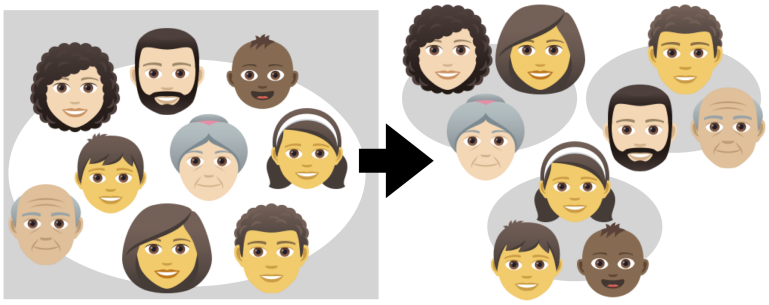
Perform model selection as in **Challenge 2.28**, **Challenge 2.29**, **Challenge 2.30**, using several synthetic datasets but this time applying the *Perceptron* classifier (choose the best value for parameters such as *learning rate* or *training epochs*).

A multi-layer Perceptron incorporates intermediate layers of neurons between the input and output neurons. You can have a look at the effects of including these new layers in the digital neural network playground found at:
`http://playground.tensorflow.org`

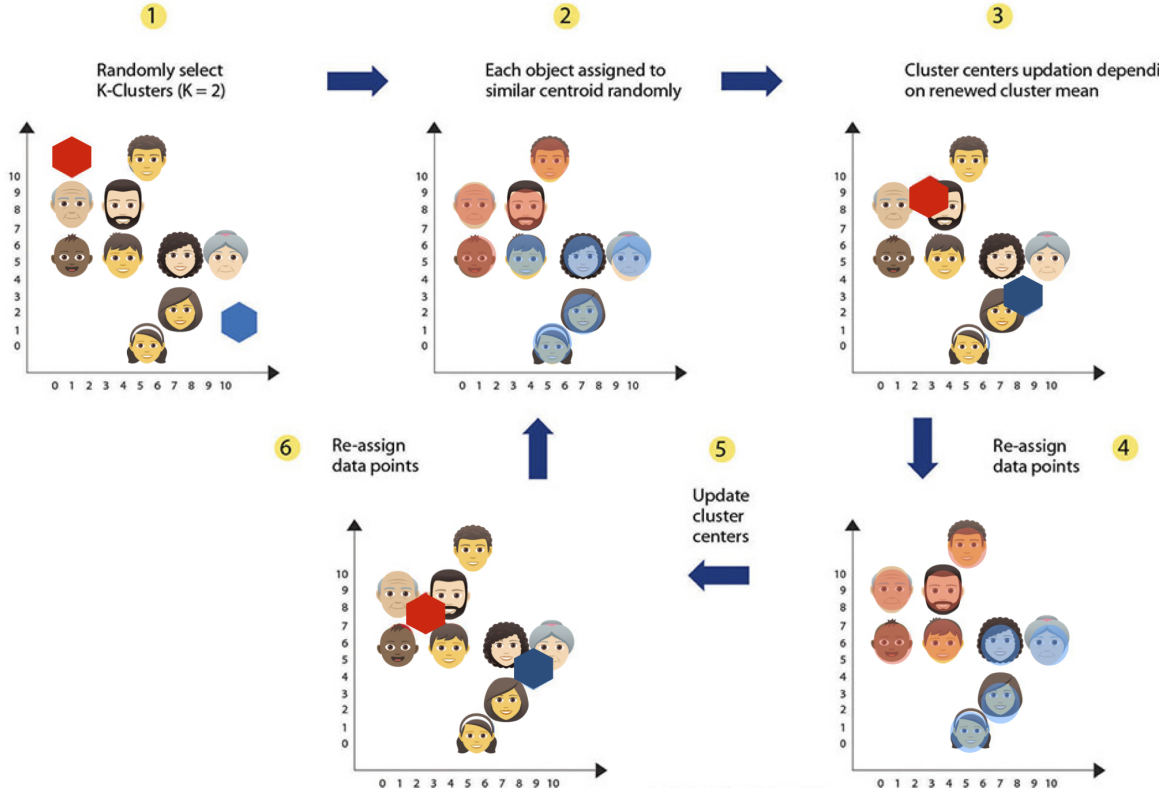


Clustering

Clustering is one of the main applications of unsupervised learning. Basically the goal is, given a raw (unlabelled) dataset, get the machine to discover clusters or groups of data items that are related by some of their intrinsic properties or attributes.



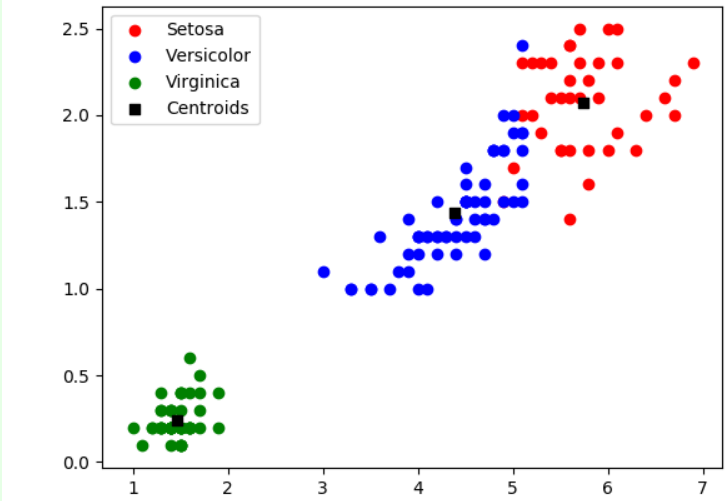
A popular clustering technique is k-*means*. This algorithm is aimed at partitioning the input space in regions by finding k representative centroids for the data. Then the elements in the dataset closest to each centroid are associated to each cluster. Let's say we represent the above faces dataset using two attributes in a 2D input space, and that we collected 9 observations; the algorithm proceeds as it is illustrated below.



k-means is ready and set to be used in `sklearn`. Let's try it on Iris:

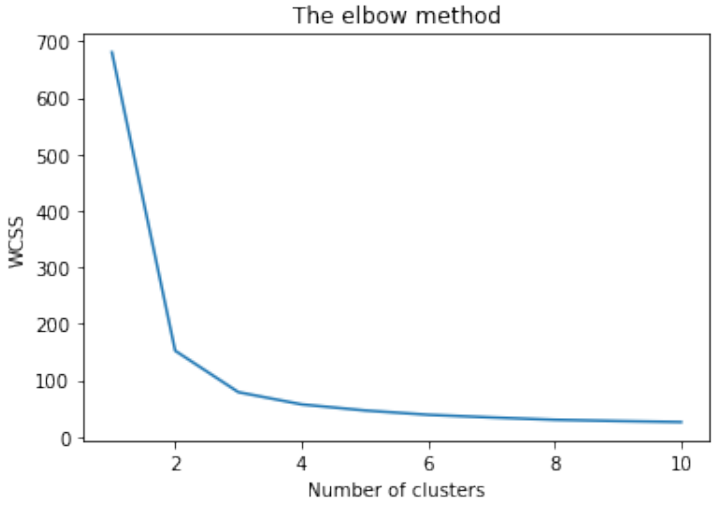
```
import sklearn.datasets as datasets
from sklearn.cluster import KMeans

## Load Iris dataset ##
iris = datasets.load_iris()
X = iris.data
Y = iris.target

## Run k-means on the Iris dataset ##
kmeans = KMeans(n_clusters = 3)
y_kmeans = kmeans.fit_predict(x)

## Visualise the discovered clusters ##
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Iris
      -setosa')
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s = 100, c = 'blue', label = '
      Iris-versicolour')
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s = 100, c = 'green', label = '
      Iris-virginica')

## Visualises also the the centroids of the clusters ##
plt.scatter(km.cluster_centers_[:,2], km.cluster_centers_[:,3], c="black", label="
      Centroids", marker="s")

## Show the plot ##
plt.legend()
plt.show()
```

Out[11]:



In this case we set $k = 3$ since we knew there were observations from three species in the dataset. What happens when we do not know the number of clusters in advance? Well, a good heuristic to choose k is the "elbow" method: choose the value of k where the within cluster sum of squares does not decrease significantly:

```
## Run different models k = 1,...,11 ##
wcss = []       #within cluster sum of squares
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 300, n_init = 10,
          random_state = 0)
    kmeans.fit(x)
    wcss.append(kmeans.inertia_)

## Model selection (choosing k) by observing the 'elbow' ##
plt.plot(range(1, 11), wcss)
plt.title('The elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



Challenge 2.33

Create at least 3 synthetic datasets with different numbers of clusters ($k \in \{2,...,5\}$). Then train k-*means* in each dataset (12 datasets in total) and visualise the discovered clusters. Report the accuracy achieved for each model in each case.

NB. *Use* `make_classification`, `make_blobs`, `make_circles` *functions to create the datasets.*

Challenge 2.34

Use the *elbow method* for model selection of $k \in \{1,...,11\}$) in the problems of **Challenge 2.33**. Report the effectiveness of the method in discovering the correct number of clusters in each case.

Suggested readings

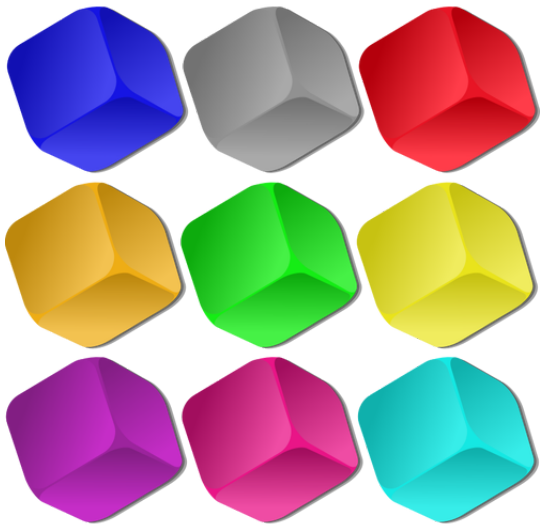
↓ Contemporary books

- Ethem Alpaydin. *Introduction to Machine Learning*, 3rd Edition, MIT Press, 2014.
- Sarah Guido & Andreas MÃijller. *Introduction to Machine Learning with Python*, O'Reilly Media, 2016.
- Sebastian Raschka & Vahid Mirjalili. *Python Machine Learning*, 2nd Edition. Packt Publishing, 2017.
- Kelleher, Mac Namee & D'Arcy. *Fundamentals of Machine Learning for Predictive Data Analytics*, MIT Press, 2015.
- Peter Harrington. *Machine Learning in Action*, Manning Publications, 2012.
- Brink, Richards & Fetherolf. *Real-World Machine Learning*, Manning Publications, 2016.

↓Classic books

- Duda, R. O., Hart, P. E., & Stork, D. G. *Pattern classification*. John Wiley & Sons, 2012.
- Bishop, C. *Pattern recognition and machine learning*. Springer, 2016.
- Shawe-Taylor, John, and Nello Cristianini. *Kernel methods for pattern analysis*. Cambridge University Press, 2004.
- Mitchell, T. *Machine Learning*. McGraw Hill, 1997.

CHAPTER 3



Metaheuristic Methods

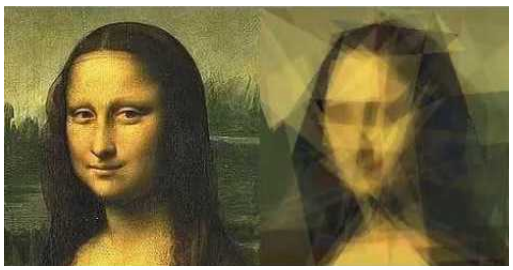
WHENEVER we face a situation where multiple choices are available, we aim to pick the best one, in order to satisfy or even exceed our expectations. That is what we mean by *optimisation*, a recurring theme occurring in our daily life, but also in many scenarios in industry, management, planning, design, engineering, medical services and logistics. Actually, any question for a superlative is an optimisation problem (what is the fastest or the cheapest or the most robust or the most valuable, the most profitable, etc.). In this chapter, we shall discuss the metaheuristic way of solving such kind of problems.

A typical example in optimisation is the *Travelling Salesman Problem (TSP)*: given a list of cities and distances, the traveller should find the best tour that visits each city only once whilst minimising a measure of interest (distance, cost, time, etc.) so as to presumably become the champion salesman in his company.

Metaheuristics are general procedures to search for reasonably well-suited approximate solutions to optimisation problems. That is, in contrast to exact methods that find the optimum of a cost function, these methods iteratively explore local regions aiming to reach a sufficiently good solution (a worthy candidate, although probably not the best). This situation is common in many engineering applications, where optimisation problems are usually combinatorial or ill–defined and hence, where exact methods are not applicable. As a matter of fact, metaheuristics is currently a relevant and hot topic of research within the field of Data Science.

On the other hand, industrial applications are facing increasingly larger volumes of data to analyse, where exact methods are difficult to apply and therefore metaheuristics can be considered as an alternative. Some of the most popular are Hill Climbing, Simulated Annealing, Evolutionary Algorithms and Estimation of Distribution Algorithms.

Exact methods rely on taking advantage of mathematical properties of the optimisation function, such as gradients, Hessians, convexity and so on. When the objective function does not comply with such well–behaved problem definitions, metaheuristics are an alternative to find an approximate solution. For example, if the problem is to generate art automatically, defining a mathematical function may prove difficult. Here a metaheuristic is trying to paint an image that strikingly resembles DaVinci's Mona Lisa, by using only coloured, overlapped triangles (see further details in: `chriscummins.cc/s/genetics/`)

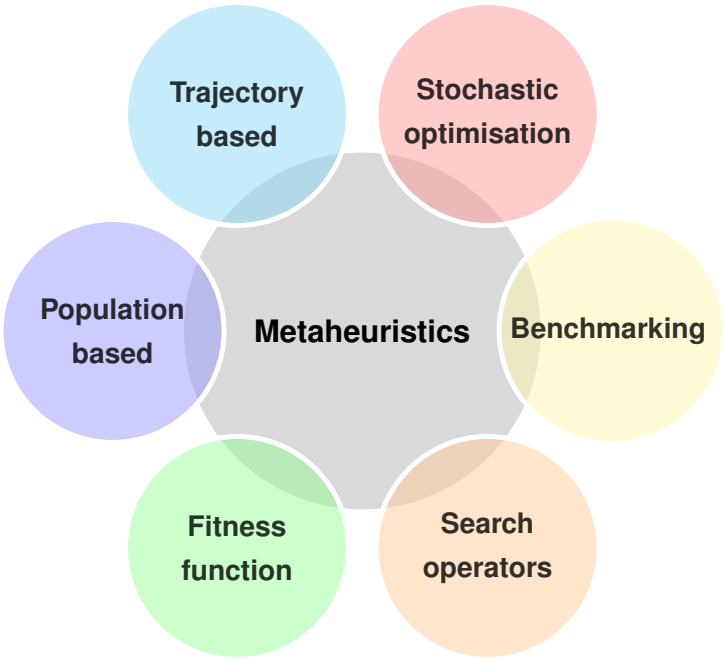


The core idea of these techniques is to randomly generate solutions that are progressively improved with variation operators; a metaheuristic performs some sort of stochastic optimisation and as such, it uses rules to guide the search over a large set of feasible solutions with less computational effort.

This is why metaheuristics are mainly empirical approaches implemented and tuned with computer experiments. Mastering their basic tools and implementation techniques would enable practitioners and researchers not only to apply them on real-world engineering problems but also to experiment with new versions of operators, representations, hybridisations or completely novel metaheuristics.

The big picture

The essential concepts we shall cover in our exposition of metaheuristics are summarised in the following diagram:



As it can be seen in the picture, roughly speaking metaheuristics are stochastic search procedures: they find solutions by randomly exploring the space of feasible candidates and then exploiting regions of promising fitness. Here *fitness* (or suitability) of a candidate solution is the only information these algorithms use to perform the search. Fitness is computed with a cost function usually corresponding to the optimisation objective that is defined in terms of the measure of efficiency the problem is trying to minimise (or maximise).

Moreover, metaheuristics approaches are categorised in two large groups: trajectory–based and population–based. The former category groups search algorithms working around a single candidate solution that moves (changes) along an optimisation trajectory. The changes are performed using local search operators, that is, an algorithmic operation that produces a new candidate, hopefully a fitter one. Some search operators are the creation of a random solution or the mutation of an existing solution. Usually these metaheuristics handle a single-point or instantaneous candidate whereas others take advantage of a memory of visited solutions in the trajectory.

On the other hand, population–based metaheuristics maintain a list of candidates that are being simultaneously evaluated as potential optimal solutions. These techniques are usually bioinspired, in the sense they are modelled after processes occurring in nature (animal behaviour, genetics, sociodynamics, astrophysics, etc.). Here again the fitness function ranks the suitability of each candidate in the population to optimise the problem. And also the search operators produce new candidates out of old ones. But in this case the operators work at the individual level (mutation, crossover) or at the population level (creation, selection).

Lastly, notice that different metaheuristics can be applied to a particular problem. The key issues when using any of them is firstly to define the variable problem domain (discrete or continuous), then to define a proper candidate solution representation (variable encoding or genotype), followed by a mapping to the actual behaviour of the solution in the problem context (variable decoding or phenotype), and finally designing appropriate search operators and fitness function. The empirical comparison of several metaheuristics in order to assess their behaviour, to adjust their parameters and to choose the best performing model is known as benchmarking, an analogous process to that of model selection in Machine Learning.

This chapter focuses on several instances of both, trajectory–based and population–based metaheuristics, including *Hill Climbing, Simulated Annealing*, *Genetic Algorithm* and *Estimation of Distribution Algorithms*. Throughout practical examples and exercises, the tutorial progresses from the essential to the more elaborate concepts. Let's begin with our first example.

Visual insights

A movie-scheduling system: part 1

Let us assume you are the owner of a local cinema screen in your neighbourhood. You are a cinema–lover, so along with the obvious blockbuster screenings, you want your regulars to have the chance to watch the best movies in cinema history. So, every week you set aside a time slot on Sundays to schedule as many as possible of these movies on your screen. Thus, this exercise is tailored towards building a movie–scheduling process by using basic data–manipulation operations with `numpy` and `matplotlib`. The aim in the first part of this exercise is to conduct a exploratory analysis of the history of Oscar–winning movies.

In this exercise we will work again with the database of Oscar winner movies from 1927 to 2018.

The dataset file can be downloaded from:
`https://goo.gl/MaqRKQ`

name	year	nominations	rating	duration	genre1	genre2	release
Shape of Water	2018	13	7.4	123	Fantasy	Romance	August
Moonlight	2017	8	7.5	111	Drama		November
Spotlight	2016	6	8.1	128	Crime	Drama	November
Birdman	2015	9	7.8	119	Comedy	Drama	November
12 Years Slave	2014	9	8.1	134	Biography	Drama	November
Argo	2013	7	7.8	120	Biography	Drama	October
The Artist	2012	10	8	100	Comedy	Drama	October
The King Speech	2011	12	8	118	Biography	Drama	December
The Hurt Locker	2010	9	7.6	131	Drama	History	July
(...)							

Dataset credit: Shehroz S. Khan @ U of Toronto

So let's start–off by loading the data into a Python table:

```
import matplotlib.pyplot as plt
import numpy as np
# Data loading
movies = np.loadtxt('best-pictures.csv'
        , dtype={'names': ('name','year','nominations','rating','duration','genre1','
             genre2','release','synopsis')
        , 'formats': ('S30','u2','u1','f2','u2','S10','S10','S10','S255')}, delimiter='
             ,', skiprows=1)
# Show first rows of table
print('\n---------- First rows of table ----------')
print(movies[:][1:5])
```

```
Out[1]:

---------- First rows of table ----------
[(' Spotlight', 2015, 6, 8.1, 128, 'Crime', 'Drama', 'November', 'The true story of how the...')
 (' Birdman', 2014, 9, 7.8, 119, 'Comedy', 'Drama', 'November', 'Illustrated upon the progress...')
 (' 12 Years a Slave', 2013, 9, 8.1, 134, 'Biography', 'Drama', 'November', 'In the antebellum United...')
 (' Argo', 2012, 7, 7.8, 120, 'Biography', 'Drama', 'October', 'Acting under the cover of...')]
```

Using Python it is easy to find out which are the longest movies:

```
longest = np.sort(movies, order='duration')
print('\n---------- Top 5 longest movies ----------')
print(longest[["name","year","duration","genre1","genre2"]] [-1:-5:-1], sep="\n")
```

```
Out[2]:

---------- Top 5 longest movies ----------
[(' Gone With the Wind', 1939, 238, 'Drama', 'Romance')
 (' Lawrence of Arabia', 1962, 216, 'Adventure', 'Biography')
 (' Ben-Hur', 1959, 212, 'Adventure', 'Drama')
 (' The Godfather: Part II', 1974, 202, 'Crime', 'Drama')]
```

Challenge 3.1

Find the top five of best–rated and most–nominated movies. Report also the bottom five of worst–rated and least–nominated movies.

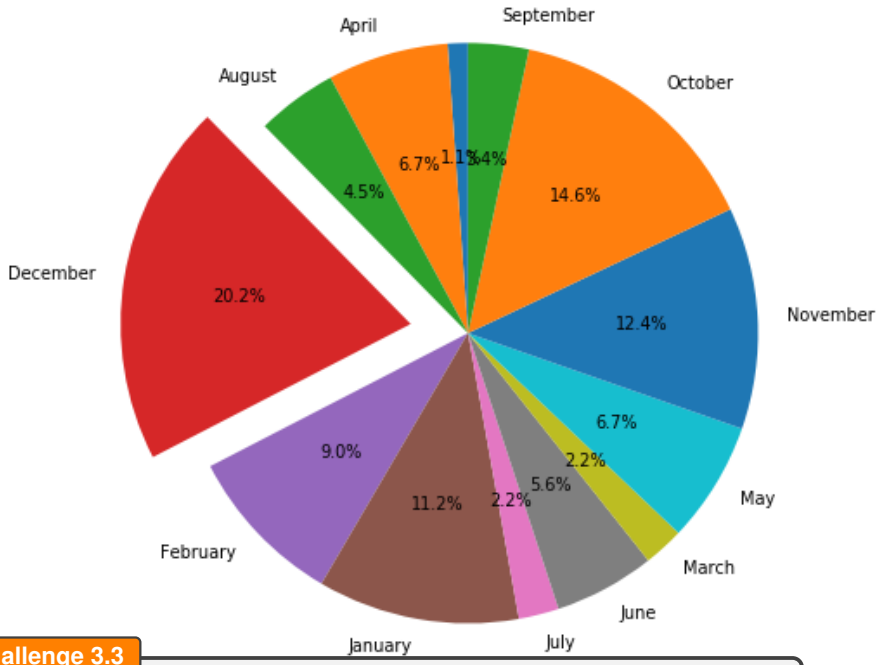
Challenge 3.2

Which are the five Oscar-winning movies with the longest titles? Which are the five with shortest titles?

Now let's visualise some information regarding month of release:

```
print('\n---------- Distribution by month of release ----------')
months, counts = np.unique(movies["release"], return_counts=True)
print(months, "\n", counts)
offset = np.zeros(len(months))
offset[3] = 0.2  # offsets the largest slice (i.e. 'December')
fig, ax = plt.subplots()
ax.pie(counts, explode=offset, labels=months, autopct='%1.1f%%', shadow=False,
      startangle=90)
ax.axis('equal')  # ensures that pie is drawn as a circle, not oval
plt.show()
```

```
Out[3]:

---------- Distribution by month of release ----------
['April' 'August' 'December' 'February' 'January' 'July' 'June' 'March'
 'May' 'November' 'October' 'September']
 [ 6  4 18  8 10  2  5  2  6 11 13  3]
```



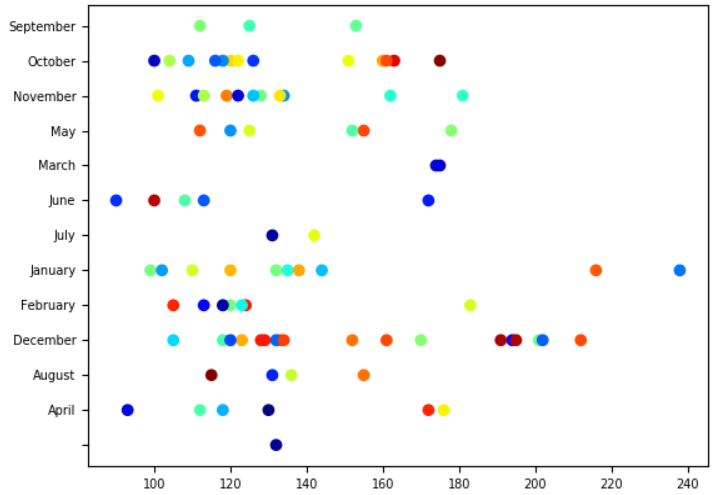
Challenge 3.3

Produce a pie plot of movie distribution by genre.

NB: *Notice that one movie may belong to more than one genre, so they should be aggregated first.*

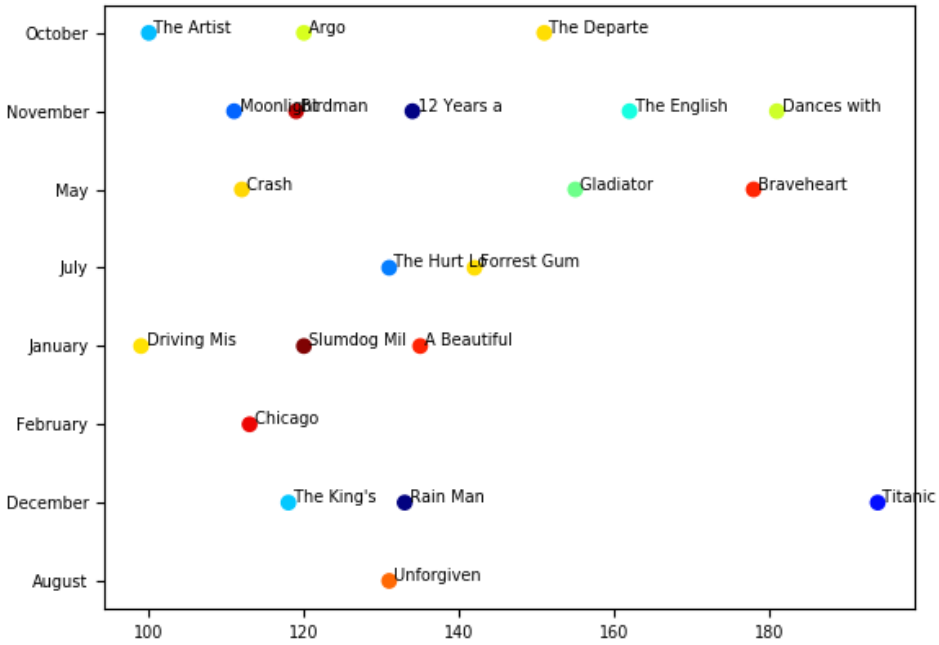
Now let's see if we can find any pattern between month of release and movie length, using a scatter plot:

```
## Scatter plot of release month vs length ##
x = movies["duration"]
y = movies["release"]
names = movies["name"]
N = len(names)
colors = np.random.rand(N)
plt.scatter(x, y,
c=colors, cmap="jet")
plt.show()
```



... not much really, but wouldn't be nicer to also show the movies' titles?

```
## Scatter plot with annotations ##
plt.scatter(x, y, c=colors,
            cmap="jet")
for i, txt in enumerate(names):
    plt.annotate(txt, (x[i],y[i]))
plt.show()
```



Challenge 3.4

Show the scatter plot of movie duration vs. nominations, movie duration vs. ratings and movie nominations vs. ratings.

Challenge 3.5

Can you think of a trick to visualise all the previous dimensions in a single plot? For example, let's say you take movie duration vs ratings as a base plot; how can you show visually the other two dimensions (nominations and release month)? Produce such enhanced scatter plot for the latest 20 best movie winners.

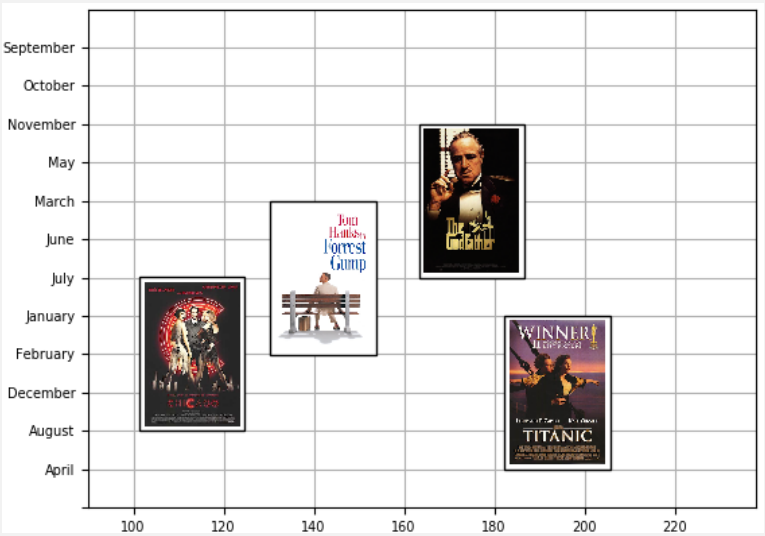
Challenge 3.6

Let's assume you plan to allocate only one slot of minimal duration for a single projection of a Oscar-winning movie in your Sunday screening times. Using similar plots as those described before, how will you choose the best-ranked, shortest movie per each genre, that will be shown on a two–month classic–movies season?

Challenge 3.7

Produce a scatter plot of your favorite 5 movies using their posters as point markers (using month of release and movie length as axes).

Hint: *see* `goo.gl/qyTCRb` *and* `goo.gl/y4xSH8` *for some guidance.*



Exhaustive search

Let's try to solve the cinema-scheduling problem with a computational approach. Instead of looking at candidate solutions using visualisation tools, one can perform an *exhaustive search* over the entire solution space. To do so, we will need a mechanism to enumerate all possible combination of candidates, in other words, a measure of *order* is needed. Then, an algorithm may try every candidate in order one after the other until the optimum is found.

Assume the cinema owner will allow up to two movies to be screened every Sunday, with the shortest running times and highest rankings. Your job is to find the combination of movies that optimise these two criteria. For this purpose, firstly we must define the search space (or *solution representation*) and a quality measure (also known as *fitness*), and then perform and exhaustive search of every possible combination. Since this strategy use no information to guide the search, it is is also known as *Brute Force*.

So, in this problem the **solution representation** or **encoding** can be defined as a vector of two dimensions, corresponding to the identifiers (or indexes) of the first and second movie to be screened, within the list of 90 movies (i.e. integer numbers in the range $0, 1, 2, \ldots, 89$). Besides, regarding the **fitness function** the criterion here can be to minimise the projection runtime of the selected two movies. Alternatively we may combine several optimisation criteria, e.g. minimise duration while maximising user ratings.

The above mentioned elements are included in the following script:

```
import pandas as pd

## Load movies dataset ##
movies = pd.read_csv("best-pictures.csv")
print(movies.head(5))
print(movies["synopsis"][0])
print(movies.shape)

## Fitness function. Criterion: total combined duration of x = [i, j]) ##
## (solution representation is x = [i, j]; i, j are in [0,89])        ##
def fitnessDuration(x):
    return movies.duration[x[0]] + movies.duration[x[1]]

## Fitness function. Criterion: minimise duration and maximise rating (raw values) ##
def fitness(x):
    return movies.duration[x[0]] + movies.duration[x[1]] - movies.rating[x[0]] - movies
          .rating[x[1]]
```

Now, most of the times we need to transform the way an arbitrary candidate is coded in the solution space into a representation that provides meaningful information to the final user; we will call such a transformation a *genotype-phenotype mapping* (`GPM()` in our code), taking inspiration on the mechanism living organisms use to map their genetic information into visible phenotypical traits. In our example, this mapping could simply be the printed list of titles of the scheduled movies along with the total running time. From here, a nested loop will suffice to enumerate all the possible combinations:

```
## Genotype-phenotype mapping function ##
def GPM(x):
    phenotype = [movies["name"][x[0]], movies["name"][x[1]],
                 "Total length: " + str(movies.duration[x[0]]+movies.duration[x[1]])]
    return phenotype

# Loop over all possible combinations (exhaustive enumeration) ##
for i in range(0,89):
    for j in range(i+1, 89):
        x = [i, j]
        print(GPM(x))
```

Challenge 3.8

Obtain the optimal solution to the movie scheduling problem for the combined duration and rating criteria, using exhaustive search.

Hint 1: *Modify the enumeration loop shown above to keep track of the best solution found so far.*

Hint 2: *Instead of using raw data, consider normalising the values of the components of the fitness function. What strategy does obtain better results?*

```
## Preprocess data: normalise values ##
movies["duration_norm"] = movies["duration"]/max(movies["duration"])
movies["rating_norm"] = movies["rating"]/max(movies["rating"])

## Re-define fitness function with normalised values ##
def fitness(x):
    return movies.duration_norm[x[0]] + movies.duration_norm[x[1]] - movies.
          rating_norm[x[0]] - movies.rating_norm[x[1]]
```

Challenge 3.9

Now repeat **Challenge 3.8** but this time produce a list of classic movies to schedule in the next month (4 Sundays). For this purpose you may try storing the intermediate results in a 90×90 matrix and then find those solutions with the 4 lowest values, taking care of not scheduling the screening of the same movie more than once.

Challenge 3.10

Define a different fitness function and run the experiment again. Is the solution found different? Is there any difference in the search procedure?

NB. *Consider using runtime arguments to support your answer.*

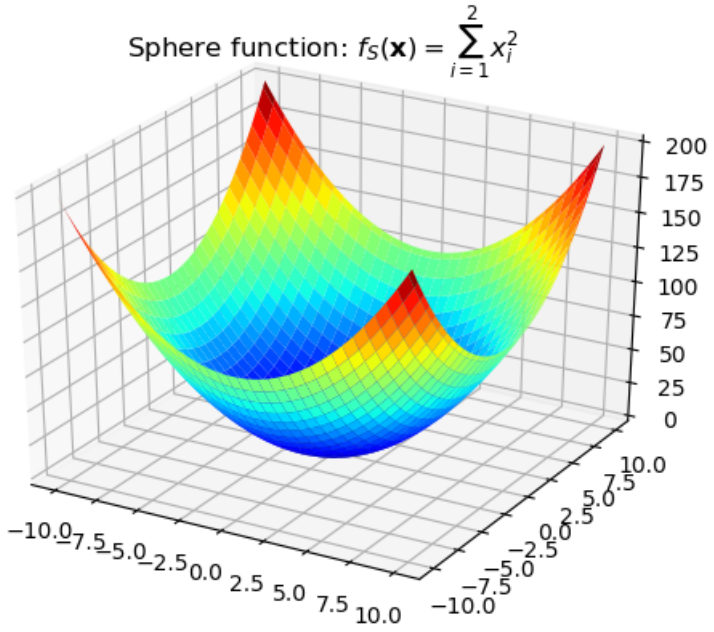
The previous exercise dealt with a problem where the variables of the solution space were discrete. Now we shall see how to address optimisation of continuous domain cost functions, using *Brute Force* in comparison to the simplest metaheuristic of *Random Search*.

For this purpose let's first consider the **Sphere** fitness function defined over a vector $\mathbf{x} = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$, as:

$$f_S(\mathbf{x}) = \sum_{i=1}^{d} x_i^2, \qquad -10 \leq x_i \leq 10,$$

which in $\mathbb{R}^2$ can be visualised using the following Python script:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from mpl_toolkits.mplot3d import Axes3D

x1 = np.linspace(-10, 10, 101)
x2 = np.linspace(-10, 10, 101)
X1, X2 = np.meshgrid(x1, x2)
F = X1**2 + X2**2
fig = plt.figure()
axis = fig.gca(projection='3d')
surf = axis.plot_surface(X1, X2, F, cmap=cm.jet)
plt.title("Sphere function: $f_S(\mathbf{x}) = \sum_{i=1}^2 x_i^2$")
plt.show()
```



Challenge 3.11

Produce 3D plots of the modified **Sphere** function with constant *offset* $\mathbf{c} \in \mathbb{R}^d$:

$$\widehat{f}_S(\mathbf{x}, \mathbf{c}) = \sum_{i=1}^{d} (x_i - c_i)^2.$$

For example, in $\mathbb{R}^2$ ($d = 2$), use $\mathbf{c} = [5, 5]$, $\mathbf{c} = [-8, -8]$ and $\mathbf{c} = [2.5, -2.5]$.

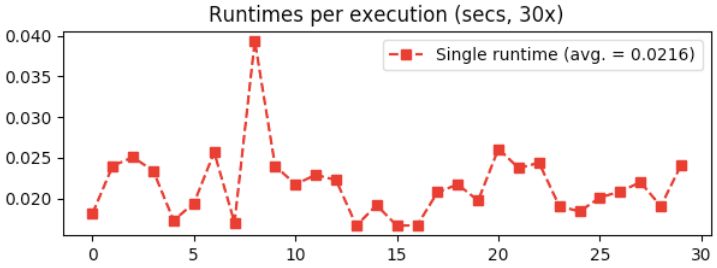
So now let's find the minimum of this function by *Brute Force*. The search space in this case would be $\mathrm{X} = [-10, 10]^2 \subset \mathbb{R}^2$, and the cost function would be the **Sphere** function, f_S. One way of doing this, is looping over the entire set of values in coordinates $\mathbf{x}_1$ and $\mathbf{x}_2$ (here a tiny tweak was included for timekeeping of execution times):

```
## Sphere cost function ##
def f_sphere(x1, x2):
    return x1**2 + x2**2

## Exhaustive search over coordinate arrays ##
print('\n--- Exhaustive search over coordinate arrays ---')
import time
t = time.time()
min_f = float('inf')
for a in x1:
    for b in x2:
        f_ab = f_sphere(a, b)
        if f_ab < min_f:
            argmin_f = [a, b]
            min_f = f_ab
print("The minimum of function Sphere is: ", min_f, "\nI found it at location: ",
      argmin_f)
print("Time elapsed: %.4f secs" % (time.time() - t))
```

```
Out[4]:

--- Exhaustive search over coordinate arrays ---
The minimum of function Sphere is:  0.0
I found it at location:  [0.0, 0.0]
Time elapsed: 0.0193 secs
```

Not bad, uh? In fact, you can verify that this is the global minimum by visual inspection of the 3D plot.

Notice that single executions may result in different run times depending on processor overload at each run; hence, it is recommended to report the average over many repetitions (at least 30 runs):



Another way of doing it would be to exhaust all the possible combinations of pairs within the 2D coordinate mesh-grid:

```
## Exhaustive search over coordinate mesh grid ##
print('\n--- Exhaustive search over coordinate mesh grid ---')
t = time.time()
min_f = float('inf')
for a, b in zip(X1,X2):
    f_ab = f_sphere(a, b)
    if np.min(f_ab) < min_f:
        ind = np.argmin(f_ab)
        argmin_f = [a[ind], b[ind]]
        min_f = np.min(f_ab)
print("The minimum of function Sphere is: ", min_f, "\nFound at location: ", argmin_f)
print("Time elapsed: %.4f secs" % (time.time() - t))
```

Challenge 3.12

Which implementation strategy runs faster: searching over the coordinate arrays or searching over the mesh grid? Support your answer with evidence of average running times of 30 executions per strategy.

Challenge 3.13

Using *Brute Force* with coordinate array search, show the *search trajectory* of problems in **Challenge 3.11**, i.e., the path joining the different candidate solutions found during the execution of the algorithm.

NB. *Consider using a contour plot for visualisation purposes.*

Now let's see if this metaheuristic is able to find the global minimum of the offset **Sphere**. For this aim you may use the following function definition:

```
### Sphere offset cost function ###
def f_hat_sphere(x1, x2, c1, c2):
    return (x1-c1)**2 + (x2-c2)**2
```

Challenge 3.14

What are the minima of $\widehat{f}_S(\mathbf{x}, [5,5])$, $\widehat{f}_S(\mathbf{x}, [-8,-8])$ and $\widehat{f}_S(\mathbf{x}, [2.5,-2.5])$? Which implementation strategy is faster? Verify your answers using the 3D plots of **Challenge 3.11** for the former question, and average runtimes plots for the latter.

Challenge 3.15

The generalised **Sphere** function with *offset* $\mathbf{c} \in \mathbb{R}^d$ and *cutoff* $b \in \mathbb{R}$ is defined as:

$$\widetilde{f}_S(\mathbf{x}, \mathbf{c}, b) = \widehat{f}_S(\mathbf{x}, \mathbf{c}) + b.$$

Repeat **Challenge 3.14**, but this time find the minimum and runtimes for $\widetilde{f}_S(\mathbf{x}, \mathbf{c}, b)$ with $b \in \{7, -1\}$.

Challenge 3.16

Describe at least two reasons why in continuous domains *Exhaustive Search* may fail to achieve the optimum. Support your answer with examples (plots, running time, etc.).

Random search

Now let's have a look at the other simplistic metaheuristic known as *Random Search*, which, as its name implies, blindly explores X by guessing repetitively at chance new candidate solutions. The basic algorithm is shown next.

Algorithm 2: *Random Search*

Input: $\text{fitness}(\mathbf{x})$, a cost function to optimise, $\mathbf{x} \in \mathbb{R}^d$
Output: $\mathbf{x}_{\text{best}}$, the best found minimum
repeat
  $\mathbf{x}_{\text{best}} \leftarrow \text{create}()$
  if $\text{fitness}(\mathbf{x}_{\text{new}}) < \text{fitness}(\mathbf{x}_{\text{best}})$ **then**
    $\mathbf{x}_{\text{best}} = \mathbf{x}_{\text{new}}$
until runtime or evaluations budget exhausted

In this case, in order to `create()` at chance one candidate solution we can use the Python pseudo–random number generator `random`. Recall that for the **Sphere** cost function, our search space would be $\mathrm{X} = [-10, 10]^2 \subset \mathbb{R}^2$, therefore we can use the following code (the stopping criterion here is a maximum number of iterations, `max_iter`):

```
## Random search over [-10, 10]^2 ##
print('\n--- Random search over [-10, 10]^2 ---')
t = time.time()
min_f = float('inf')
max_iter = 10000
for i in xrange(max_iter):
    a = np.random.uniform(-10, 10)
    b = np.random.uniform(-10, 10)
    f_ab = f_sphere(a, b)
    if f_ab < min_f:
        argmin_f = [a, b]
        min_f = f_ab
print("The minimum of function Sphere is: %.4f"  % min_f)
print("I found it at location: ", argmin_f)
print("Time elapsed: %.4f secs" % (time.time() - t))
```

```
Out[5]:

--- Random search over [-10, 10]^2 ---
The minimum of function Sphere is: 0.0047
I found it at location:  [-0.02898027155331384, -0.062019730019658326]
Time elapsed: 0.0388 secs
```

We can see that the solution found is approximate (although it would be correct at a precision level of two significant digits).

Challenge 3.17

How many iterations are needed for *Random Search* to find the global zero minimum of f_S with a rounding precision of four significant digits? Support your answer by plotting the average minimum over 30 repetitions with increasing number of `max_iter`, contrasted to their corresponding average runtimes.

Challenge 3.18

Repeat **Challenge 3.13**, **Challenge 3.14** and **Challenge 3.15** but this time finding minima using *Random Search*.

An object-oriented approach

Since from now on we will study different variations of metaheuristic algorithms, it may be convenient to modularise and encapsulate the common elements related to an optimisation procedure in a single structure, which then can be specialised to particular techniques; this is known as a *class*. Although a *class* is a more elaborated computer concept related to *Object-Oriented Programming*, for our purposes you may think of it as a template where the basic operations needed to run a metaheuristic are defined, so as to be able to create many copies (instances) to try it on different problems.

In order to do so, firstly we shall define the name of the class (in this case `class BruteForce` for *Exhaustive Search*), along with a function `__init__` intended to define variables the object will use to perform the optimisation as well as their initial values:

```
## Brute force metaheuristic class definition ##
class BruteForce:

    ## Initialization of object attributes ##
    def __init__(self, x1_range, x2_range, fitness, samples=100, max_eval=1000):
        self.x1_range = x1_range  # Domain of first coordinate
        self.x2_range = x2_range  # Domain of second coordinate
        self.fitness = fitness    # Fitness function to be optimised
        self.max_eval = max_eval  # Max number of fitness evaluations allowed
        self.samples = samples    # Granularity of coordinate sampling
        self.xmin = []            # The candidate solution found
        self.fmin = np.Inf        # The cost (fitness) of candidate solution
        self.xmins= []            # History of old candidate solutions (trajectory)
        self.toc = 0              # Timing counter
```

Then the remainder operations related with the metaheuristic are implemented. In this case we need two functions, one to perform the genotype-to-phenotype mapping (which we called `gpm()`), and the other one to perform the actual exhaustive search (which we called `optimise()`). Finally we may include an operation just to report the results found by the metaheuristic (here called `summary()`). The entire script for the `class BruteForce` is listed below.

```
import numpy as np
import time

## Brute force metaheuristic class definition ##
class BruteForce:

    ## Initialization of object attributes ##
    def __init__(self, x1_range, x2_range, fitness, samples=100, max_eval=1000):
        self.x1_range = x1_range  # Domain of first coordinate
        self.x2_range = x2_range  # Domain of second coordinate
        self.fitness = fitness    # Fitness function to be optimised
        self.max_eval = max_eval  # Max number of fitness evaluations allowed
        self.samples = samples    # Granularity of coordinate sampling
        self.xmin = []            # The candidate solution found
        self.fmin = np.Inf        # The cost (fitness) of candidate solution
        self.xmins= []            # History of old candidate solutions (trajectory)
        self.toc = 0              # Timing counter

    ## Genotype-phenotype mapping function ##
    def gpm(self):
        return "X = [%.2f, %.2f]. f(X) = %.5f" % (self.xmin[0], self.xmin[1], self.fmin
             )

    ## Optimisation procedure ##
    def optimise(self):
        x1 = np.linspace(self.x1_range[0], self.x1_range[1])
        x2 = np.linspace(self.x2_range[0], self.x2_range[1])
        tic = time.time()
        for a in x1:
            for b in x2:
                xnew = [a, b]
                fnew = self.fitness(xnew)
                if fnew < self.fmin:
                    self.xmin, self.fmin = xnew, fnew   ## New candidate solution
        self.toc = time.time() - tic

    ## Report results ##
    def summary(self):
        print('\n[%s] Candidate minimum of <%s>: %s' % (self.__class__.__name__, self.
             fitness.__name__, self.gpm()))
        print('[%s] Runtime: %.4f s. Updates: %d. Evaluations left: %d' % (self.
             __class__.__name__, self.toc, len(self.xmins), self.max_eval))

## End of class ##
```

We are ready to use the class to create two copies (or *objects* in programming terms) that solve two different problems, **Sphere** $f_S(\mathbf{x})$ y offset **Sphere** $\widehat{f}_S(\mathbf{x}, [2.5, -2.5])$:

```
## Problem definitions: Sphere and offset Sphere ##
def f_sphere(x):
    return np.sum(np.power(x, 2))

c = [2.5, -2.5]  # Modify to define functions with different offsets
def f_sphere_offset(x):
    return (x[0]-c[0])**2 + (x[1]+c[1])**2

## Main program that creates two brute force objects, and have them solving the
      problems ##
bf1 = BruteForce([-10, 10], [-10, 10], f_sphere)
bf1.optimise()
print('[BruteForce] Candidate minimum of <%s> is %s' % (bf1.fitness.__name__, bf1.gpm()
      ))
print('Runtime: %.4f s\n' % bf1.toc)

bf2 = BruteForce([-10, 10], [-10, 10], f_sphere_offset)
bf2.optimise()
print('[BruteForce] Candidate minimum of <%s> is %s' % (bf2.fitness.__name__, bf2.gpm()
      ))
print('Runtime: %.4f s\n' % bf2.toc)
```

```
Out[6]:

[BruteForce] Candidate minimum of <f_sphere> is X = [-0.20, -0.20]. f(X) = 0.08330
Runtime: 0.0134 s

[BruteForce] Candidate minimum of <f_sphere_offset> is X = [2.65, 2.65]. f(X) = 0.04686
Runtime: 0.0029 s
```

Challenge 3.19

Notice that *Brute Force* finds the most proximate solution to the actual minimum coordinates, up to the degree of precision allowed for the sampling of the coordinate vectors. Modify the previous class definition to allow the user to define the sampling precision with a variable called `samples` in the initialisation of the class. How does sampling precision relates to execution time? Support your answer with graphical plots.

Challenge 3.20

Modify also the *Brute Force* class definition to keep track of the *optimisation trajectory* as well as the number of updates that were made during such trajectory.

Challenge 3.21

Modify additionally the *Brute Force* class definition to make the algorithm to stop until certain number `max_eval` of evaluations of the fitness function has been reached.

An interesting feature of *Object Oriented Programming* is the possibility of reusing previously implemented code to add new functionality to your programs. This is accomplished with a *inheritance* mechanism, which likewise it happens in living organisms, means that a "child" class can inherits the traits and behaviour of a "mother" class. Given that in *Random Search* we need to use similar variables (`range`, `max_eval`, `xmin`, `toc`, etc.) and functions (`init()`, `gpm()`, `summary()`, but a different search procedure (`optimise()`) compared to *Brute Force*, we shall resort to the inheritance mechanism to re-implement it in an object-oriented fashion. In Python this is accomplished during the class definition by indicating in parenthesis the base class from which the new class is deriving, as it can be seen in the next script.

```
import numpy as np
import time

from bruteforce import BruteForce

## Random Search class definition: Derives from BruteForce (inheritance) ##
class RandomSearch(BruteForce):

    ## Creation of a random solution ##
    def create(self):
        x1 = np.random.uniform(self.x1_range[0], self.x1_range[1])
        x2 = np.random.uniform(self.x2_range[0], self.x2_range[1])
        return [x1, x2]

    ## Optimisation procedure ##
    def optimise(self):
        self.xmin = self.create()
        self.fmin = self.fitness(self.xmin)
        tic = time.time()
        while self.max_eval > 0:
            xnew = self.create()                    ## The core RS exploring operator
            fnew = self.fitness(xnew)
            self.max_eval -= 1
            if fnew < self.fmin:
                self.xmin, self.fmin = xnew, fnew  ## New candidate solution update
                self.xmins.extend([self.xmin])     ## Trajectory update
        self.toc = time.time() - tic

## End of class ##
```

Here, a novel exploring operator called `create()`, which does not exist in the base `BruteForce` class, was defined with the purpose of randomly sampling a new candidate from the solution space. Notice also that the optimisation procedure differs a little bit from `BruteForce`, as it loops searching for solutions until the `max_eval` number of evaluations is exhausted, and for this reason we need to re-write it (or overload it, in programming jargon). Now, let's use our two object-oriented metaheuristics (`BruteForce`, `RandomSearch`) to solve the same offset **Sphere** $\widehat{f}_S(\mathbf{x}, [2.5, -2.5])$ problem, and see the differences:

```
bf = BruteForce([-10, 10], [-10, 10], f_sphere_offset, samples=20, max_eval=100000)
bf.optimise()
bf.summary()

for i in range(3):
     rs = RandomSearch([-10, 10], [-10, 10], f_sphere_offset, max_eval=100000)
     rs.optimise()
     rs.summary()
```

```
Out[7]:

[BruteForce] Candidate minimum of <f_sphere_offset>: X = [2.65, -2.65]. f(X) = 0.04686
[BruteForce] Runtime: 0.0056 s. Updates: 183. Evaluations left: 95000

[RandomSearch] Candidate minimum of <f_sphere_offset>: X = [2.49, -2.51]. f(X) = 0.00024
[RandomSearch] Runtime: 0.3881 s. Updates: 15. Evaluations left: 0

[RandomSearch] Candidate minimum of <f_sphere_offset>: X = [2.50, -2.48]. f(X) = 0.00057
[RandomSearch] Runtime: 0.4236 s. Updates: 8. Evaluations left: 0

[RandomSearch] Candidate minimum of <f_sphere_offset>: X = [2.51, -2.48]. f(X) = 0.00065
[RandomSearch] Runtime: 0.3860 s. Updates: 9. Evaluations left: 0
```

Observe that we have run a `RandomSearch` object several times, since because of the stochastic nature of its search procedure, it produces different results in each execution. It is clear that `BruteForce` exhausted all the coordinate combinations at the given sampling precision while saving some fitness evaluations (at that same precision). On the other hand, `RandomSearch` exhausted all the fitness evaluations allocated, consuming more runtime along the way. Besides, the number of updates of candidate solutions (or trajectory) is different in each metaheuristic, as it is so the final resulting mininum.

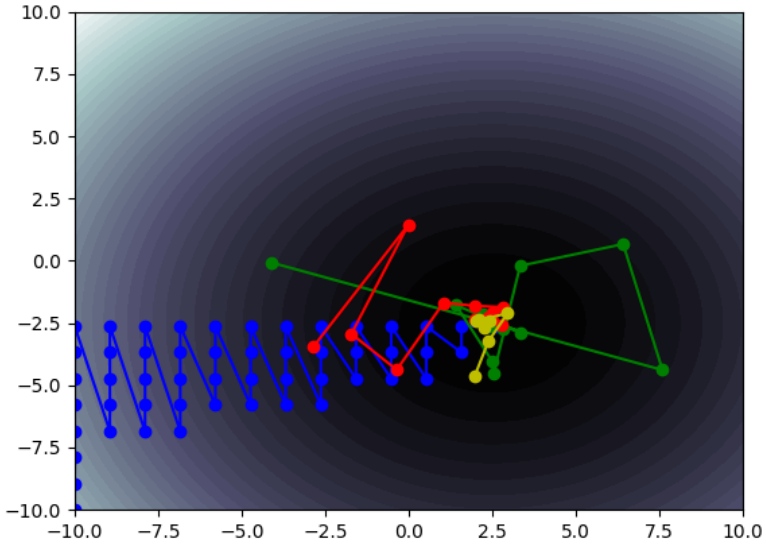
Challenge 3.22

Fiddle around with the parameters of the objects `BruteForce` and `RandomSearch` so as to get the global zero minimum of f_S with a rounding precision of four significant digits. Contrast the performance of runtime and fitness evaluations of both metaheuristics as you experiment with such parameters.

We can actually plot the trajectory of the optimisation search with the following script:

```
X, Y = np.meshgrid(np.linspace(-10, 10, 101), np.linspace(-10, 10, 101))
Z = (X - 2.5)**2 + (Y + 2.5)**2
c = ['b', 'g', 'r', 'y', 'm', 'c', 'k', 'w']    ## Trajectory colors
plt.figure()
plt.contourf(X, Y, Z, 40, cmap=cm.bone)
xmins = np.array(bf.xmins)                      ## Convert BF trajectory list to np
    array
plt.plot(xmins[:, 0], xmins[:, 1], marker='o', c=c[0])
xmins = np.array(rs.xmins)                      ## Convert RS trajectory list to array
plt.plot(xmins[:, 0], xmins[:, 1], marker='o', c=c[i+1])
plt.show()
```

The following plot show the trajectories of the `BruteForce` object (blue) and the three `RandomSearch` objects (red, green, yellow). The search space landscape is shown in 2D as a contour plot, where darker areas correspond to lower values of the offset **Sphere** $\widehat{f}_S(\mathbf{x}, [2.5, -2.5])$ problem. It is clear that in fact, the four objects converged to the global minimum located at $[2.5, -2.5]$.



Hill Climbing

So far we have addressed the problem of finding an optimum of a function by evaluating the fitness of a candidate in a solution space and then moving forward to search for another, better candidate. For this reason this methods are called **single point** or **trajectory–based** metaheuristics. The two methods we have seen, however, are not useful for practical purposes: *Brute Force* would become unfeasible in large solution spaces or high–dimensional fitness functions, whereas *Random Search* attempts to blindly search the space without actually using any structure information given by the fitness function; if lucky enough it would just "guess" an approximate good solution. In the following, we will study an smarter approach.

Hill Climbing, as its names suggests, is a metaheuristic taking inspiration from the sport of alpinism, that is, the activity of climbing mountains. Facing the challenge of ascending to the summit of a big mountain (see illustration below), what is the approach taken by alpinists?

Assuming they have no map or instruments to guide them through their route apart from an altimeter (an instrument that measures the altitude they are located above ground level), surely the best they can do starting from the initial point of their route, is to jump up to the next reachable peak. Then, to look around if there is another bigger peak (one with higher altitude) around and climb up to it. And then keep doing this climbing-to-the-next-nearest-bigger-peak until the reach the summit or they are exhausted and have to stop.

Without a map or GPS, alpinists try to reach the summit by increasingly jumping from lower peaks to the next higher reachable peak.

Well, this is exactly what the *Hill Climbing* algorithm does. It keeps making short jumps from one candidate solution to the next, by looking around the neighbourhood of the current point, whilst using the fitness function as altimeter to measure the next point where to climb to. This "looking around" operation is also known as *local search*. The basic algorithm is shown next.

Algorithm 3: *Hill Climbing*

Input: $\text{fitness}(\mathbf{x})$, a cost function to optimise, $\mathbf{x} \in \mathbb{R}^d$
Output: $\mathbf{x}_{\text{best}}$, the best found minimum
$\mathbf{x}_{\text{best}} \leftarrow \text{create}()$
repeat
  $\mathbf{x}_{\text{new}} \leftarrow \text{mutate}(\mathbf{x}_{\text{best}})$
  if $\text{fitness}(\mathbf{x}_{\text{new}}) < \text{fitness}(\mathbf{x}_{\text{best}})$ **then**
    $\mathbf{x}_{\text{best}} = \mathbf{x}_{\text{new}}$
until runtime or evaluations budget exhausted

So, the *Hill Climbing* algorithm involves using a local search operator that is denoted as *mutate()*. This operator basically cause to jump around the current solution towards other candidate solutions within its neighbourhood. In a continuous search space such as $\mathbb{R}^d$, said operation can be done by moving forwards or backwards a small step in each d dimension of the current solution. We can therefore add a positive or negative small random value, for example generated with a Gaussian normal distribution. The Python script implementing this metaheuristic is the following, where again we have taken advantage of the inheritance mechanism:

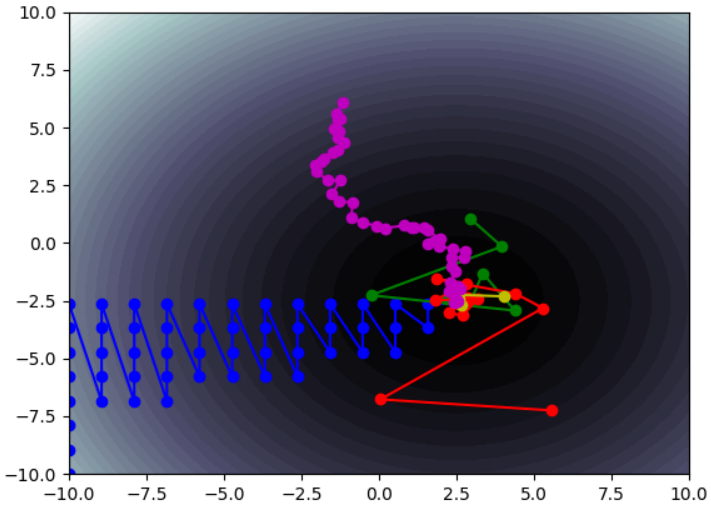
```
import numpy as np
import time
from rs import RandomSearch

## HillClimbing class definition: Inherits from RandomSearch ##
class HillClimbing(RandomSearch):

    ## Initialization of class attributes ##
    def __init__(self, x1_range, x2_range, fitness, samples=100, max_eval=1000, step
          =0.1):
        RandomSearch.__init__(self, x1_range, x2_range, fitness, samples, max_eval)
        self.step = step          # Length of the mutation step

    ## Tweaks a little bit a candidate solution randomly ##
    def mutate(self, x):
        return x + np.random.randn(2)*self.step

    ## Optimisation algorithm ##
    def optimise(self):
        self.xmin = self.create()
        self.fmin = self.fitness(self.xmin)
        tic = time.time()
        while self.max_eval > 0:
            xnew = self.mutate(self.xmin)          ## The core HC exploring operator
            fnew = self.fitness(xnew)
            self.max_eval -= 1
            if fnew < self.fmin:
                self.xmin, self.fmin = xnew, fnew  ## New candidate solution update
                self.xmins.extend([self.xmin])     ## Trajectory update
        self.toc = time.time() - tic

## End of class ##
```

So let's see how `HillClimbing` compares to `BruteForce` and `RandomSearch` in solving the offset **Sphere** $\widehat{f}_S(\mathbf{x}, [2.5, -2.5])$ problem:

```
bf = BruteForce([-10, 10], [-10, 10], f_sphere_offset, samples=20, max_eval=100000)
bf.optimise()
bf.summary()

for i in range(3):
    rs = RandomSearch([-10, 10], [-10, 10], f_sphere_offset, max_eval=100000)
    rs.optimise()
    rs.summary()

for i in range(3):
    hc = HillClimbing([-10, 10], [-10, 10], f_sphere_offset, max_eval=1000, step=.25)
    hc.optimise()
    hc.summary()
```

```
Out[8]:

[BruteForce] Candidate minimum of <f_sphere_offset>: X = [2.63, -2.63]. f(X) = 0.03463
[BruteForce] Runtime: 0.0009s. Updates: 51. Evaluations left: 99600

[RandomSearch] Candidate minimum of <f_sphere_offset>: X = [2.38, -2.48]. f(X) = 0.01388
[RandomSearch] Runtime: 0.0481s. Updates: 16. Evaluations left: 0

[RandomSearch] Candidate minimum of <f_sphere_offset>: X = [2.59, -2.55]. f(X) = 0.01009
[RandomSearch] Runtime: 0.0532s. Updates: 6. Evaluations left: 0

[RandomSearch] Candidate minimum of <f_sphere_offset>: X = [2.47, -2.57]. f(X) = 0.00665
[RandomSearch] Runtime: 0.0496s. Updates: 6. Evaluations left: 0

[HillClimbing] Candidate minimum of <f_sphere_offset>: X = [2.49, -2.50]. f(X) = 0.00015
[HillClimbing] Runtime: 0.0067s. Updates: 42. Evaluations left: 0

[HillClimbing] Candidate minimum of <f_sphere_offset>: X = [2.49, -2.50]. f(X) = 0.00012
[HillClimbing] Runtime: 0.0090s. Updates: 20. Evaluations left: 0

[HillClimbing] Candidate minimum of <f_sphere_offset>: X = [2.50, -2.51]. f(X) = 0.00019
[HillClimbing] Runtime: 0.0080s. Updates: 36. Evaluations left: 0
```

It can be seen in the previous output that the different executions of `HillClimbing` manage to achieve a better solution both closer to the actual optimum and more efficient in runtime. The behaviour is more clear if we plot the trajectories (`HillClimbing` in magenta); evidently, it is exploiting the structure given by the fitness function:



Challenge 3.23

How does the choice of `max_eval` and `step` affect the behaviour of `HillClimbing`? Try different values for `max_eval` $\in \{10^3, 10^4, 10^5, 10^6\}$ and `step` $\in \{10^{-2}, 10^{-1}, 10^0, 10^1\}$ and draw your conclusions using plot evidence.

A variation of *Hill Climbing* checks for the highest peak around the current solution before doing the next jump, i.e. the *Steepest Ascent Hill Climbing*:

Algorithm 4: *Steepest Ascent Hill Climbing*

Input: $\text{fitness}(\mathbf{x})$, a cost function to optimise, $\mathbf{x} \in \mathbb{R}^d$, n number of tweaks
Output: $\mathbf{x}_{\text{best}}$, the best found minimum
$\mathbf{x}_{\text{best}} \leftarrow \text{create}()$
repeat
  $\mathbf{x}_{\text{new}} \leftarrow \text{mutate}(\mathbf{x}_{\text{best}})$
  repeat n **times**
    $\mathbf{x}_{\text{aux}} \leftarrow \text{mutate}(\mathbf{x}_{\text{best}})$
    if $\text{fitness}(\mathbf{x}_{\text{aux}}) < \text{fitness}(\mathbf{x}_{\text{new}})$ **then**
      $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{aux}}$
  if $\text{fitness}(\mathbf{x}_{\text{new}}) < \text{fitness}(\mathbf{x}_{\text{best}})$ **then**
    $\mathbf{x}_{\text{best}} = \mathbf{x}_{\text{new}}$
until runtime or evaluations budget exhausted

In Python using inheritance, we can implement *Steepest Ascent Hill Climbing* as follows.

```
## Steepest Ascent Hill Climbing metaheuristic ##
class SteepestHC(HillClimbing):

    ## Initialization of class attributes ##
    def __init__(self, x1_range, x2_range, fitness, samples=100, max_eval=1000, step
          =0.1, n_tweaks=10):
        HillClimbing.__init__(self, x1_range, x2_range, fitness, samples, max_eval,
             step)
        self.n_tweaks=n_tweaks

    ## Optimisation algorithm ##
    def optimise(self):
        self.xmin = self.create()
        self.fmin = self.fitness(self.xmin)
        tic = time.time()
        while True:
            xnew = self.mutate(self.xmin)
            fnew = self.fitness(xnew)
            self.max_eval -= 1
            for i in range(self.n_tweaks):          ## Best nearby peak look-around
                xaux = self.mutate(self.xmin)
                faux = self.fitness(xaux)
                self.max_eval -= 1
                if faux < fnew:
                    xnew, fnew = xaux, faux
                if (self.max_eval) < 1:             ## Max_eval exceeded ending
                    self.toc = time.time() - tic
                    return
            if fnew < self.fmin:
                self.xmin, self.fmin = xnew, fnew  ## New candidate solution update
                self.xmins.extend([self.xmin])     ## Trajectory update

## End of class ##
```

Here's the outcome of `HillClimbing` and `SteepestHC` in solving the **Offset sphere** $\widehat{f}_S(\mathbf{x}, [2.5, -2.5])$ problem:

```
for i in range(3):
    hc = HillClimbing([-10, 10], [-10, 10], f_sphere_offset, max_eval=1000, step=.25)
    hc.optimise()
    hc.summary()

for i in range(3):
    steephc = SteepestHC([-10, 10], [-10, 10], f_sphere_offset, max_eval=2000, step
          =.25, n_tweaks=40)
    steephc.optimise()
    steephc.summary()
```

```
Out[9]:


[HillClimbing] Candidate minimum of <f_sphere_offset>: X = [2.50, -2.50]. f(X) = 0.00003
[HillClimbing] Runtime: 0.0087s. Updates: 48. Evaluations left: 0

[HillClimbing] Candidate minimum of <f_sphere_offset>: X = [2.50, -2.50]. f(X) = 0.00001
[HillClimbing] Runtime: 0.0063s. Updates: 67. Evaluations left: 0

[HillClimbing] Candidate minimum of <f_sphere_offset>: X = [2.51, -2.48]. f(X) = 0.00035
[HillClimbing] Runtime: 0.0091s. Updates: 61. Evaluations left: 0

[SteepestHC] Candidate minimum of <f_sphere_offset>: X = [2.50, -2.50]. f(X) = 0.00002
[SteepestHC] Runtime: 0.0165s. Updates: 27. Evaluations left: 0

[SteepestHC] Candidate minimum of <f_sphere_offset>: X = [2.50, -2.50]. f(X) = 0.00001
[SteepestHC] Runtime: 0.0132s. Updates: 24. Evaluations left: 0

[SteepestHC] Candidate minimum of <f_sphere_offset>: X = [2.50, -2.50]. f(X) = 0.00000
[SteepestHC] Runtime: 0.0121s. Updates: 17. Evaluations left: 0
```

Challenge 3.24

Find a choice of parameters which allow `SteepestHC` to clearly outperform `HillClimbing`. Support your answer using runtime plots.

The local search loop of *Steepest Ascent Hill Climbing* favours *exploitation* instead of *exploration*, that is, it looks around the neighbourhood of the current candidate to choose a higher peak to climb on, assuming no further regions of higher peaks exists further away. A variation of this algorithm would be to allow the climber to explore other intermediate peaks even if it moves away from the current highest solution; in order to do not get ashtray and forgive the best found solution, it can leave a mark in the highest visited peak, that is, keep it in memory while it explores other promising peaks. We shall call this variation *Steepest Ascent Hill Climbing with Replacement*:

Algorithm 5: *Steepest Ascent Hill Climbing with Replacement*

Input: $\text{fitness}(\mathbf{x})$, a cost function to optimise, $\mathbf{x}\in\mathbb{R}^d$, n number of tweaks
Output: $\mathbf{x}_{\text{best}}$, the best found minimum
$\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_{\text{new}} \leftarrow \text{create}()$
repeat
  $\mathbf{x}_{\text{tmp}} \leftarrow \text{mutate}(\mathbf{x}_{\text{new}})$
  repeat n **times**
    $\mathbf{x}_{\text{aux}} \leftarrow \text{mutate}(\mathbf{x}_{\text{new}})$
    if $\text{fitness}(\mathbf{x}_{\text{aux}}) < \text{fitness}(\mathbf{x}_{\text{tmp}})$ **then**
      $\mathbf{x}_{\text{tmp}} \leftarrow \mathbf{x}_{\text{aux}}$
  $\mathbf{x}_{\text{new}} \leftarrow \mathbf{x}_{\text{tmp}}$
  if $\text{fitness}(\mathbf{x}_{\text{new}}) < \text{fitness}(\mathbf{x}_{\text{best}})$ **then**
    $\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_{\text{new}}$
until runtime or evaluations budget exhausted

Challenge 3.25

Implement the class *Steepest Ascent Hill Climbing with Replacement* (`ReplacementHC`) and show how its behaviour differs from the other versions of *Hill Climbing* on solving the **Generalised Sphere** problem.

A final variation of *Hill Climbing* is intended to avoid premature convergence to a local minimum due to a poor choice of the initial random candidate. The idea would be then to restart the climbing route (exploration) after certain epoch of local search (exploitation). An implementation of this new variation should add a new parameter N representing an epoch (number of function evaluations) of exploitation before the exploration restarts, as follows:

Algorithm 6: *Hill Climbing with Restarts*

Input: $\text{fitness}(\mathbf{x})$, a cost function to optimise, $\mathbf{x} \in \mathbb{R}^d$
Input: n number of tweaks, N epoch length, $N \gg n$
Output: $\mathbf{x}_{\text{best}}$, the best found minimum
$\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_{\text{new}} \leftarrow \text{create}()$
$i \leftarrow N$
repeat
  repeat n **times**
    $\mathbf{x}_{\text{aux}} \leftarrow \text{mutate}(\mathbf{x}_{\text{new}})$
    $i \leftarrow (i-1)$
    if $\text{fitness}(\mathbf{x}_{\text{aux}}) < \text{fitness}(\mathbf{x}_{\text{new}})$ **then**
      $\mathbf{x}_{\text{new}} \leftarrow \mathbf{x}_{\text{aux}}$
  if $\text{fitness}(\mathbf{x}_{\text{new}}) < \text{fitness}(\mathbf{x}_{\text{best}})$ **then**
    $\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_{\text{new}}$
  if $i \leq 0$ **then** `/* restart condition */`
    $\mathbf{x}_{\text{new}} \leftarrow \text{create}()$
    $i \leftarrow N$
until runtime or evaluations budget exhausted

Challenge 3.26

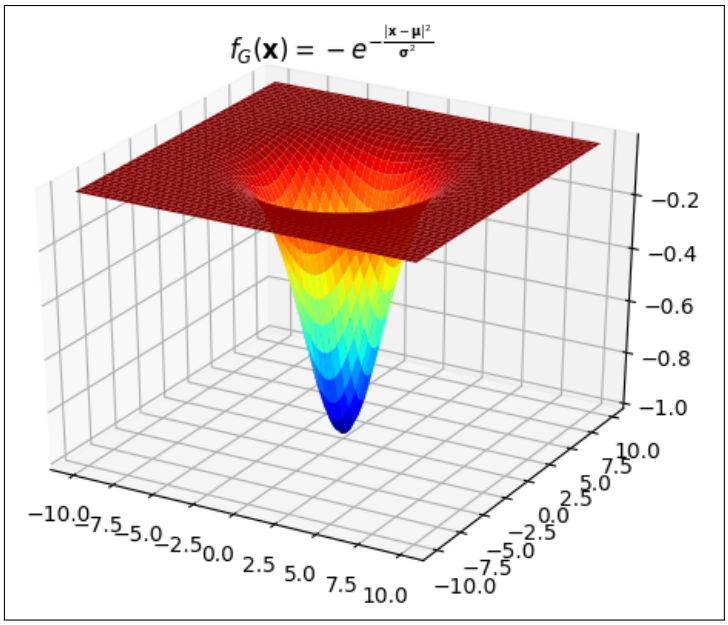
Implement the class *Hill Climbing with Restarts* (name it `RestartHC`) and show how its behaviour differs from the other versions of *Hill Climbing* on solving the **Generalised Sphere** problem.

The family of **Sphere** problems are well-behaved functions with a nice and sharp gradient that is conveniently exploited by the *Hill Climber*-type meta-heuristics. What would happen with other slightly more difficult problems?

Let's consider the **Inverted Hat** problem The mathematical definition of the **Inverted Hat** is given by a *Gaussian* cost function defined over a vector $\mathbf{x} = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$, with mean $\boldsymbol{\mu} \in \mathbb{R}^d$ and standard deviation $\sigma \in \mathbb{R}$, as follows:

$$f_{\check{\mathbf{G}}}(\mathbf{x}) = -e^{-\frac{\|\mathbf{x}-\boldsymbol{\mu}\|^2}{\sigma^2}}$$

The **Inverted Hat** problem looks like the following picture.



To implement this function in Python, we can use the built-in operations of the `numpy` library to compute the exponential function (`np.exp()`) and the norm of an array (`np.linalg.norm()`):

```
## Inverted Gaussian function definition (unimodal) ##
def f_gaussian(x, mu=[0, 0], sigma=1):
    return -np.exp(-(np.linalg.norm(np.array(x)-mu)/sigma)**2)
```

Defined in this way, `f_gaussian(x)` evaluates the function on a single point $\mathbf{x}$. Alternatively, we can take advantage of the `numpy` capability to evaluate the function on multiple points if passed as a bi-dimensional matrix $\mathbf{X}$; in this case, we need to specify in which dimension `np.linalg.norm()` will operate, using the parameter `axis=0` to indicate rows or `axis=1` to indicate columns. The modified Python definition of the **Inverted Hat** function will look like it follows:

```
## Inverted Gaussian function definition (unimodal) ##
def f_gaussian(x, mu=[0, 0], sigma=1):
    axis = 0 if (np.size(x) == 2) else 1  # this is the axis numpy uses to compute norm
    return -np.exp(-(np.linalg.norm(np.array(x)-mu, axis=axis)/sigma)**2)
```

The latter definition can now be used for plotting purposes, as we can reshape the meshgrid coordinates into a matrix to be passed as the `x` parameter of the `f_gaussian()` function in order to evaluate all the **Inverted Hat** function values within the $[-10, 10]^2$ subspace; this time, we will use the meshgrid operations available in the `numpy` library, as it is shown next. When executed, this script produces the plot showed previously.

```
## Let's plot the inverted Gaussian ##
X = np.mgrid[-10:10:.1, -10:10:.1]      # Coordinates meshgrid
Xl = X.T.reshape(-1,2)                  # Convert to a coordinates 2D matrix
Zl = f_gaussian(Xl, sigma=3)            # Evaluates unimodal Gaussian on coordinates
Z = Zl.reshape(X[0].shape)              # Convert back function values to meshgrid
fig = plt.figure()
axis = fig.gca(projection='3d')
surf = axis.plot_surface(X[0], X[1], Z, cmap=cm.jet)
plt.title("$f_G(\mathbf{x}) = -e^{-\\frac{\|\mathbf{x}-\mathbf{\mu}\|^2}{\mathbf{\sigma
     }^2}}$")
plt.show()
```

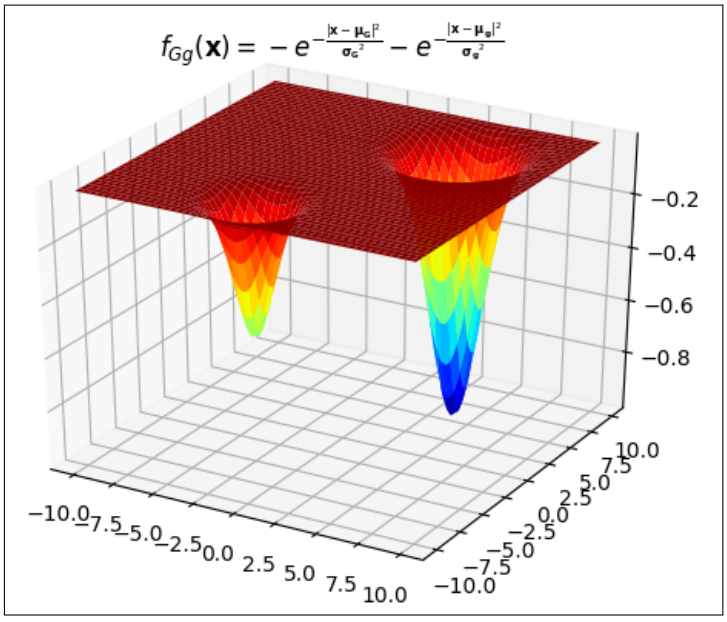
Let's move on to define a more complicated optimisation function on the basis of the **Inverted Hat**. This time our problem will include two local minima, let's say an inverted hat with **Two Peaks** (also known as inverted bimodal *Gaussian*):

$$f_{\breve{\mathbf{G}}\breve{\mathbf{g}}}(\mathbf{x}) = -e^{-\left(\frac{1}{\sigma_G^2}\|\mathbf{x}-\boldsymbol{\mu}_G\|^2\right)} - e^{-\left(\frac{1}{\sigma_g^2}\|\mathbf{x}-\boldsymbol{\mu}_g\|^2\right)}$$

We can define the **Two Peaks** function easily in Python, by invoking twice the previously defined inverted Gaussian (unimodal) with different mean and standard deviation values:

```
## A bimodal inverted Gaussian: landscape with two peaks ##
def f_two_peaks(x):
    return f_gaussian(x, mu=[5.01, 4.09], sigma=2) + 0.5*f_gaussian(x, mu=[-6.55,
         -1.48], sigma=1.5)
```

The corresponding plot is the following:



Challenge 3.27

Write and execute the Python script to produce the previous **Two Peaks** function plot.

Challenge 3.28

Define several versions of three, four and five-peaks **Inverted Hat** functions and visualise them using Python.

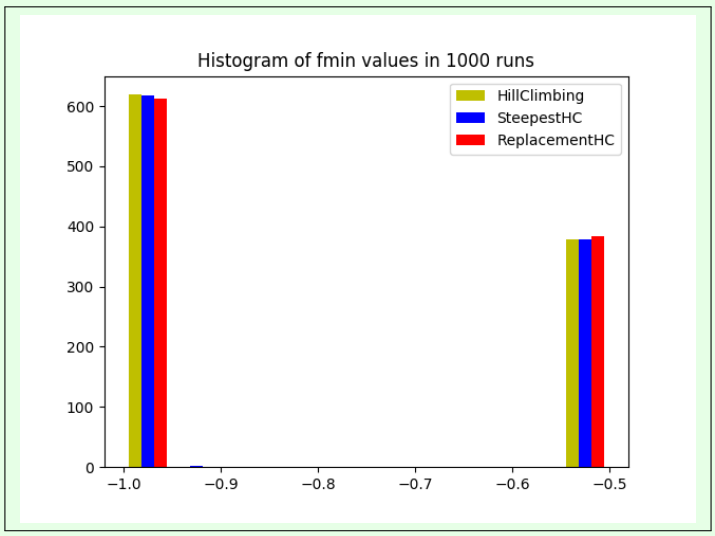
Now let's see if our *Hill Climbers* metaheuristics are good at solving the **Two Peaks** problem as efficiently as the **Sphere** problems. Remember that because of the stochastic nature of this algorithms one single execution is not sufficient to obtain a definite answer, so we need to repeat the experiments several times and report average behaviour. We shall keep the results of these repetitions in auxiliary lists `stats` for function minima and number of updates, and `xstar` for the actual minimum location found in each execution.

```
## Now let's see the HillClimbers in action!!! ###
reps = 1000                             # Number of repetitions
stats, xstar = [], [];                  # Auxiliary lists to store results
for i in range(reps):
    hc1 = HillClimbing([-10, 10], [-10, 10], f_two_peaks)
    hc1.optimise()
    hc1.summary()
    stats.append([hc1.fmin, len(hc1.xmins)])
    xstar.append(hc1.xmin)

for i in range(reps):
    hc2 = SteepestHC([-10, 10], [-10, 10], f_two_peaks)
    hc2.optimise()
    hc2.summary()
    stats.append([hc2.fmin, len(hc2.xmins)])
    xstar.append(hc2.xmin)

for i in range(reps):
    hc3 = ReplacementHC([-10, 10], [-10, 10], f_two_peaks)
    hc3.optimise()
    hc3.summary()
    stats.append([hc3.fmin, len(hc3.xmins)])
    xstar.append(hc3.xmin)
```

One convenient way of comparing how each algorithm performed (apart from examining the `summary()` reports printed in the console), is to plot the statistics as stacked histograms:

```
## Now plot the fmin stats ##
stats = np.array(stats)
fmins = stats[:,0].reshape(reps, 3)
updates = stats[:,1].reshape(reps, 3)
labels = ["HillClimbing", "SteepestHC", "ReplacementHC"]
colors = ['y','b','r']

plt.figure()
plt.hist(fmins, color=colors, label=labels)
#plt.yticks(range(reps))
plt.legend()
plt.title("Histogram of fmin values in 1000 runs")
plt.show()
```

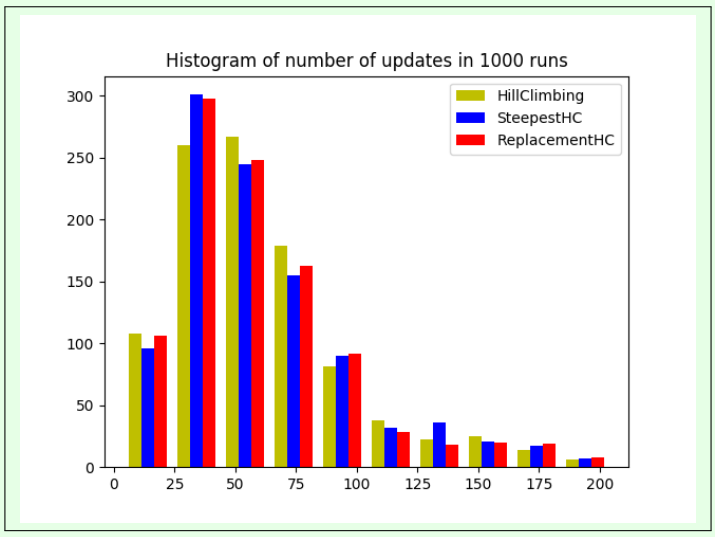
Out[10]:



We can see here that basically the three algorithms perform very similar. They are able to discover both minima of the function (with values -1.0 and -0.5). However, about 40% of the times they got stuck into the local minimum (0.5), whereas 60% of the times they successfully reached the global minimum (1.0). Now let's have a look at the number of updates:

```
## Now updates stats ##
plt.figure()
plt.hist(updates, color=colors, label=labels)
#plt.yticks(range(reps))
plt.legend()
plt.title("Histogram␣of␣number␣of␣updates␣in␣1000␣runs")
plt.show()
```

Out[11]:



The latter plot suggests that `SteepestHC` and `ReplacementHC` are in fact more efficient in obtaining the solution with a fewer number of updates (≤ 50).

Challenge 3.29

Now compare the accuracy of the *Hill Climbers* algorithms in finding the solution of `f_two_peaks()`. Notice that in this example the local minimum is located in $\mu_g = [-6.55, -1.48]$, whereas the global minimum is in $\mu_G = [5.01, 4.09]$. How can you nicely report the precision obtained by each algorithm (that is, the degree of proximity achieved)?

Random Walk

A *Random Walk* (also known as Drunkard's walk, you will see why later), is a search strategy much like *Random Search* with a little improvement in the form of a memory. Here, the algorithm basically searches around the current candidate blindly, moving forwards, backwards or sideways randomly accepting every new candidate solution with no questions asked, but taking care of remembering the best-so-far visited peak. The algorithm is defined below, with its Python implementation further down (yes, inheritance again!).

Algorithm 7: *Random Walk*

Input: $\text{fitness}(\mathbf{x})$, a cost function to optimise, $\mathbf{x} \in \mathbb{R}^d$
Output: $\mathbf{x}_{\text{best}}$, the best found minimum
$\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_{\text{new}} \leftarrow \text{create}()$
repeat
  $\mathbf{x}_{\text{new}} \leftarrow \text{mutate}(\mathbf{x}_{\text{new}})$
  if $\text{fitness}(\mathbf{x}_{\text{new}}) < \text{fitness}(\mathbf{x}_{\text{best}})$ **then**
    $\mathbf{x}_{\text{best}} = \mathbf{x}_{\text{new}}$
until runtime or evaluations budget exhausted

```
## Random Walk class implementation ##
class RandomWalk(HillClimbing):

    ## Optimisation procedure ##
    def optimise(self):
        xnew = self.create()
        fnew = self.fitness(xnew)
        self.xmin, self.fmin = xnew, fnew
        tic = time.time()
        while self.max_eval > 0:
            xnew = self.mutation(xnew)
            fnew = self.fitness(xnew)
            self.max_eval -= 1
            if fnew < self.fmin:
                self.xmin, self.fmin = xnew, fnew  ## New candidate solution update
                self.xmins.extend([self.xmin])     ## Trajectory update
        self.toc = time.time() - tic

## End of class ##
```

Challenge 3.30

Perform again these experiments, but for the *multi-peak* functions you defined in **Challenge 3.28**. This time use the full battery of metaheuristics you know up to this point: `BruteForce`, `RandomSearch`, `HillClimbing`, `SteepestHC`, `ReplacementHC`, `RestartHC`, `RandomWalk`. What overall conclusion can you draw from your results?

NB. *In order to obtain a fair comparison make sure you use the same running parameters for all the algorithms.*

Challenge 3.31

Come up with a new variation of *Hill Climbing* intended to escape more efficiently from local minima. Name it in your honour, implement it in Python and show how this new metaheuristic is advantageous compared to the other *Hill Climbers* of **Challenge 3.30**. You may use again for this aim, the multi-peak functions you defined in **Challenge 3.28**.

Simulated Annealing

Simulated Annealing is a metaheuristic inspired by the metallurgy industry where metal pieces are melted seeking to obtain a more robust structure resistant to cracking. Roughly speaking, the process starts by heating the material at high temperatures, allowing its molecules to acquire a large amount of kinetic energy that causes them to move around randomly and quickly, adopting multiple coupling configurations. Later on, the process continues by allowing the material to cool down slowly; with this decrease in temperature, the molecules calm down their movement adopting configurations that minimise the energy and maximise its stability. The final structure assumed by the molecules at the end of the process guarantees to improve the strength properties of the metal minimising the risks of susceptibility to fractures.

The cooling mechanism of the annealing process combines the two concepts applied in metaheuristics to search the space of candidate solutions: *exploration* and *exploitation*. The exploration phase occurs when the temperature is high, allowing the algorithm to evaluate candidates even in sub-optimal regions of the search space. The exploitation phase occurs in the cooling down phase, when new configurations are accepted only with a decreasing probability depending on the temperature. In algorithmic terms, this is implemented using a probability of accepting a suboptimal candidate depending on a variable that simulates the behaviour of the annealing temperature. Thus, the temperature is varied according to a cooling schedule that starts with a high value and decreases subsequently with time. In this way, the lower the temperature the smaller chances a suboptimal candidate will be accepted.

The pseudocode of the *Simulated Annealing* algorithm is showed next.

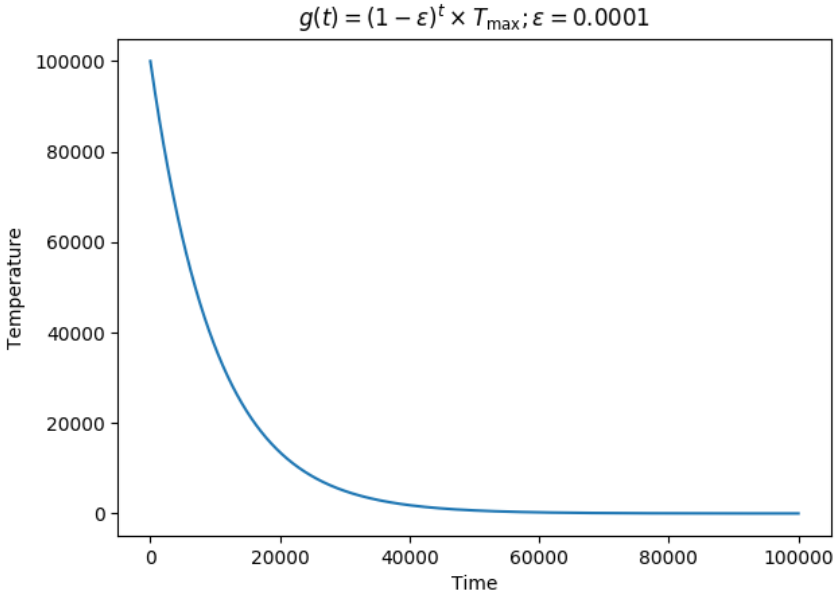
Algorithm 8: *Simulated Annealing*

Input: $\text{fitness}(\mathbf{x})$, a cost function to optimise, $\mathbf{x} \in \mathbb{R}^d$
Input: $\text{cooldown}(T)$, a temperature schedule $T \leq T_{\max} \in \mathbb{R}$
Output: $\mathbf{x}_{\text{best}}$, the best found minimum
$T \leftarrow T_{\max}$
$\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_{\text{new}} \leftarrow \text{create}()$
repeat
  $\mathbf{x}_{\text{aux}} \leftarrow \text{mutate}(\mathbf{x}_{\text{new}})$
  $\Delta E \leftarrow \text{fitness}(\mathbf{x}_{\text{aux}}) - \text{fitness}(\mathbf{x}_{\text{new}})$
  $p \leftarrow \text{random}(0, 1)$
  if $\Delta E < 0$ **or** $p < e^{-\frac{\Delta E}{T}}$ **then**
    $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{aux}}$
  $T \leftarrow \text{cooldown}(T)$
  if $\text{fitness}(\mathbf{x}_{\text{new}}) < \text{fitness}(\mathbf{x}_{\text{best}})$ **then**
    $\mathbf{x}_{\text{best}} \leftarrow \mathbf{x}_{\text{new}}$
until runtime or evaluations budget exhausted

Here the $\text{cooldown}()$ schedule can be simulated with any monotonous decreasing function of time. Take for example the following function:

$$g(t) = (1 - \epsilon)^t \times T_{\max}$$

If we set $\epsilon = 0.0001$, the temperature will cool down as shown in the following plot.



Challenge 3.32

Plot other alternative temperature schedule functions. Consider for example the logarithmic schedule:

$$g(t) = T_{\max} / \log t,$$

or the linear schedule:

$$g(t) = (1 - t/T_{\max}) \times T_{\max}$$

Moreover, propose your own decreasing schedule.

Challenge 3.33

Implement the `SimulatedAnnealing` class for this metaheuristic. Evaluate its performance in the multi–peak functions of **Challenge 3.28**.

Challenge 3.34

Recall the problem of scheduling a cycle of Oscar–winning movies in your cinema screen. Let's say you decided to set apart a time slot of 6 hours on next Sunday to screen a Oscar–winning movies marathon. You want of course, to optimise this time slot to schedule as many as possible best–rated and most–nominated movies.
Start–off by defining a search space X, candidate encoding and cost function $f(\mathbf{x}), \mathbf{x} \in$ X that allow you to optimise this problem. Then use all the *Hill Climbers, Random Walk* and *Simulated Annealing* metaheuristics to report a solution. Notice the order in which movies are screened *does not* matter.

***NB1**: Since all but* Brute Force *are stochastic metaheuristics, their results may vary depending on the chosen initial candidate solution. Therefore you should repeat the experiments several times before reporting your final answers.*

***NB2**: The `create()` and `mutation()` functions in the class definitions should be adjusted to the discrete nature of the search space of this problem.*

Challenge 3.35

Use the metaheuristics in **Challenge 3.34** to report the optimum to the problem $\widetilde{f}_S(\mathbf{x}, [5.56, -7.77], 100.93)$. Are all metaheuristics able to find the actual minimum to a rounding precision of two significant digits? If the answer is not, try tuning the execution parameters to make them more effective (*hint:* use different values for `samples`, `max_eval`, `n_repeats`). What conclusion do you reach regarding which metaheuristic is better? (more precise, more runtime efficient, etc.).

***NB**: Support your answers using plots (runtime vs parameters, trajectories) and tables (evaluations, success rate)*

Genetic Algorithms

Genetic Algorithms are optimisation techniques inspired in the Darwinian ideas of the origin of species: (1) organisms within a species reproduce and their offspring phenotypes may exhibit tiny variations with respect to those of their parents; and (2) variations that increase their chances of survival are favoured by nature since as a result, such organisms are able to live longer and reproduce more.

Natural selection as mechanism of natural evolution: character variation is inherited from parents to offspring; these diversity is filtered by the selective pressure of nature (predators, accidents, climate catastrophes) so as to favour variations that let the organisms to adapt and survive to their living conditions.

In the above illustration, the characteristic trait of brown beetles gives them an evolutionary advantage to survive over their green cousins which are food source of hungry birds and were also affected by unfortunate events. As a result, the brown beetle population manage to survive. Images credit: University of California Museum of Palaeontology, see: `evolution.berkeley.edu`

This metaphor led to the design of the computational method for optimisation known as the *Genetic Algorithm* (GA). A GA's goal is to minimise (or maximise) a fitness function by means of a population that evolves during many generations, and whose individuals reproduce, recombine, mutate and die, while finding adaptations (solutions) to the optimisation problem along the way. Hence the evolutionary undertone of the algorithm.

From a computational point of view, an individual is represented as an array of "genes" or encoding variables ("chromosome") affecting the fitness function. By applying genetic operators (crossover, mutation) to a population of these chromosomes, an offspring is produced and the fittest individuals are selected to continue to the next generation. Better adaptations (that is, more suitable solutions) yielding better chances of surviving in the environment (that is, obtaining higher fitness scores) are found, as the "selection of the fittest" pressure guides the search process.

John Holland and his students in Michigan invented the GAs; David Goldberg and collaborators in Illinois are recognised pioneers in the field. For more information, see: `http://illigal.org`

In the GA, "nature" or the evolutionary pressure representing predators, climate changes, limitation of food, disasters, cosmic rays, etc., must be carefully designed within the cost function that evaluates the fitness of a particular candidate to solve the optimisation problem

More specifically the algorithm loops through the following steps: (i) Encode the initial population, (ii) Evaluate population fitness using a cost function, (iii) Apply genetic operators, (iv) Repeat until termination criteria is met, (v) Return the best solution emerged. The pseudocode of the GA algorithm is shown below. Applications of GA range from automated industrial design to protein structure prediction to gene expression analysis to job-shop scheduling to neural nets training to automatic software testing, just to name a few.

We remark that GA are not global optimisers; instead they are able to find "good enough" local optima, depending on the design of the chromosome, genetic operators, number of generations, etc. Moreover, they are stochastic in the sense that multiple runs of the algorithm may lead to different resulting solutions.

Algorithm 9: *Genetic Algorithm*

Input: n: population size, m: generations, p_s, p_c, p_m: selection, crossover and mutation rates.
Input: $\text{fitness}(\mathbf{x})$: a cost function to optimise, $\mathbf{x} \in \mathcal{P}$: a chromosome enconding, $\mathcal{P} \subset \mathbb{R}^\ell$: a population of chromosomes
Output: $\mathbf{x}_{\text{best}}$: the best found minimum
$\mathcal{P} \leftarrow \text{initialise}(n, \ell)$;
$\mathcal{F} \leftarrow \text{fitness}(\mathcal{P})$
$\mathbf{x}_{\text{best}} \leftarrow \text{create}()$;
repeat m **generations**
  $\mathcal{P}' \leftarrow \text{selection}(\mathcal{P}, \mathcal{F}, p_s)$
  $\mathcal{P}^\dagger \leftarrow \text{crossover}(\mathcal{P}', p_c)$
  $\mathcal{P}^\ddagger \leftarrow \text{mutation}(\mathcal{P}^\dagger, p_m)$
  $\mathcal{P} \leftarrow \mathcal{P}^\ddagger$;
  $\mathcal{F} \leftarrow \text{fitness}(\mathcal{P})$
  $\mathbf{x}_{\text{best}} \leftarrow \text{update\_best}(\mathcal{P}, \mathcal{F}, \mathbf{x}_{\text{best}})$

Let's translate the pseudocode into an initial template of the `GA` class in Python. So, firstly we need to handle the running parameters:

```
import numpy as np
import time

## Genetic Algorithm class implementation ##

class GeneticAlgorithm():

    ## Initialization of algorithm parameters ##
    def __init__(self, fitness, pop_size=100, max_gen=100, sol_len=10, ps=0.5, pc=0.8,
          pm=0.01, max_eval=10000):
        self.fitness = fitness    # Fitness function to be optimised
        self.max_eval = max_eval  # Max number of fitness evaluations allowed
        self.pop_size = pop_size  # Population size (number of chromosomes)
        self.max_gen = max_gen    # Max number of generations to evolve
        self.sol_len = sol_len    # Chromosome length (number of genes)
        self.ps = ps              # Selection rate (proportion of survival population)
        self.pc = pc              # Crossover rate (probability of breeding)
        self.pm = pm              # Mutation rate (probability of mutating)
        self.xbest = []           # The best found solution
        self.fbest = np.Inf       # The fitness of best solution
        self.xolds = []           # History of older bests solutions
        self.toc = 0              # Timing counter

(...)
```

Now let's write the main search procedure of the GA. Observe that in accordance with the biological inspiration of the algorithm, we have renamed such routine from `optimise()` to `evolve()`. In this case the stopping criterion involves two conditions, either running out of the fitness evaluations budget, or achieving the maximum allowed number of generations:

```
class GeneticAlgorithm():
(...)
    ## Optimisation algorithm ##
    def evolve(self):
        self.pop = self.createPop()
        self.fit = self.fitness(self.pop)
        self.xmin = self.create()
        self.fmin = self.fitness(self.xmin)
        tic = time.time()
        while self.max_eval > 0 and self.max_gen > 0:
            newpop = self.selection(self.pop, self.fit, self.ps)
            newpop = self.crossover(newpop, self.pc)
            newpop = self.mutation(newpop, self.pm)
            self.pop = newpop
            self.fit = self.fitness(self.pop)
            self.update_best()
            self.max_eval -= self.pop_size
            self.max_gen -= 1
        self.toc = time.time() - tic
(...)
```

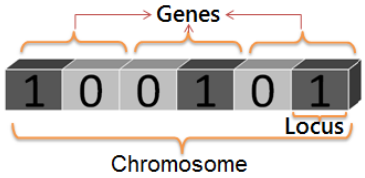
So, the main search routine is ready, what is left to code is the implementation of the genetic operators. We shall defer this task until some of the techniques proposed in the literature, are explained later in the text.

```
class GeneticAlgorithm():
(...)
    ## Genetic operators template ##
    def createPop(self):
        # ToDo: create a random initial population #
        pass

    def selection(self, pop, fit, ps):
        # ToDo: selection operator #
        pass

    def crossover(self, pop, pc):
        # ToDo: crossover operator #
        pass

    def mutation(self, pop, pm):
        # ToDo: mutation operator #
        pass

    def update_best(self):
        # ToDo: update routine #
        pass

## End of class ##
```

Solution representation

As it was alluded before, the variables defining the optimisation problem must be encoded in an artificial chromosome. The chromosome consists of a set of genes (variables) located at different positions (locus), each one holding a candidate value for the variable they represent.

A common representation strategy uses binary arrays as chromosomes; they codify for boolean variables. Other strategies encode solutions as arrays of integers or decimal numbers. An in-between scheme for problems with numerical variables is to represent them as binary–coded integers or decimals with either fixed or floating point formats. An example of a binary chromosome is shown below.



Each gene in a chromosome $\mathbf{c} = [c_1, c_2, ..., c_\ell]$ will have a locus (index) indicating its position in the array, with ℓ denoting the chromosome length.

Binary encoding

This is the most common solution representation, where variables are boolean and are represented as an array of 0's and 1's. This was also the original representation proposed by Holland; most of the theory behind GAs are based on fixed-length binary chromosomes. Many of these theories have been extended to real encoding, although they have not been studied as thoroughly as the binary encodings. Additionally, the heuristics to determine appropriate GA parameters (such as crossover and mutation rates), are generally developed in the context of binary strings. So, let's see how to create and inspect a binary string in Python:

```
c = np.random.randint(2, size=sol_len)
print "A random binary chromosome: " + str(c)
print "Locations with bit on:", c==1
print "Number of bits on:", (c==1).sum()
print "Number of bits off:", (c==0).sum()
```

```
Out[12]:

A random binary chromosome: [1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1 0 0 1 0]
Locations with bit on: [ True  True False False  True  True False  True  True  True
True False True  True  True  True False False  True False]
Number of bits on: 13
Number of bits off: 7
```

Sometimes it is difficult to find an appropriate mapping of the solution space to a binary representation, thus it may be more convenient to consider the use of other types of encodings, such as discrete or real.

Real encoding

When the variables in the problem are defined over a continuous domain, the most natural way of encoding a candidate solution is a direct mapping to real values.

Notice that a binary string may code for a real value using a floating point representation. Their precision, however, is limited by the number of significant digits one uses during encoding. It is suggested to utilise the type of encoding that is most natural for the problem, either binary or real-number encoding.

Challenge 3.36

Write a Python script to generate a random chromosome with 10 real-valued numbers as genes, within the range $[-15, 15]$. The script should also report the number of positive and negative genes.

Challenge 3.37

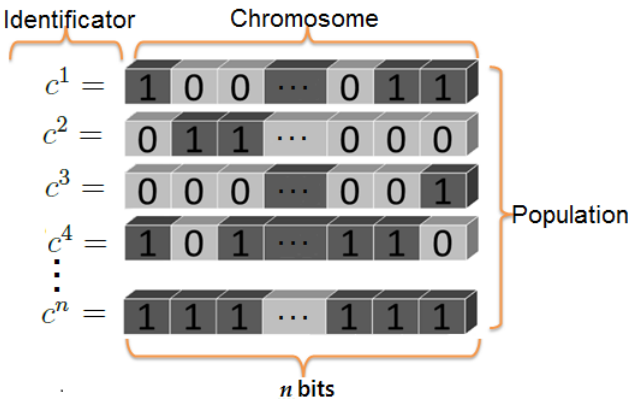
Repeat **Challenge 3.36** but this time represent the genes as integer numbers in the range $[-15, 15]$ with a binary encoding (for example, the integer number 7 is coded as "0111" in binary, the integer 12 is "1100"). What would be the length of the chromosome?

***NB**: Here the genetic information is written in an alphabet that must be translated to a phenotypical readable representation. Hence, it can be useful to define a genotype-to-phenotype mapping for this task (recall the function* `gpm(c)`*).*

Initial population

The first generation of a GA consist of a spontaneously-created random population of chromosomes. So, a number of m chromosomes are generated as uniform distributed random values according to the chosen encoding. Other techniques include biasing the generation towards promising regions of the solution region known beforehand, or seeding some chromosomes as archetypes or attractors of the random generation process.

An example of a binary–coded initial population is illustrated below.



Similarly to the way we generated a single chromosome as a `numpy` array, we may obtain a population of chromosomes using a binary matrix:

```
pop_size = 5
pop = np.random.randint(2, size=(pop_size, sol_len))
print "A random binary population: \n\n", pop
```

```
Out[13]:

A random binary population:

[[1 0 1 0 1 0 1 0 1 1]
 [1 0 0 1 1 0 0 1 0 1]
 [0 1 1 0 1 1 1 0 0 0]
 [0 0 1 0 0 0 1 1 0 1]
 [0 0 0 0 1 1 1 0 0 0]]
```

Challenge 3.38

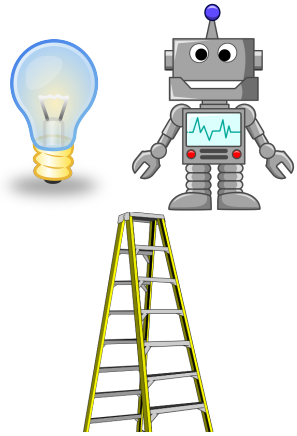
Repeat **Challenge 3.36** and **Challenge 3.37**, this time generating a population of 100 chromosomes. Demonstrate that the gene variation across the population is uniform.

***Hint**: Plot the histogram of gene distribution per chromosome locus.*

Fitness function

Also known as cost function or optimisation function, the *fitness function* plays the key role of being the selective pressure in the artificial environment of the GA, which ultimately guides the evolution of the algorithm. By evaluating this function on a given chromosome, the GA obtains its quality at solving the optimisation problem. For this purpose, the function firstly decodes the underlying variables of the chromosome and then evaluates its value.

Let's have a look at a simple example. Think of a robot designed to change broken light bulbs on the ceiling of a room. A 7-step ladder is located below the bulb. The robot must climb up to the top of the ladder, otherwise he would not be able to reach the bulb socket. The problem consists of programming the series of 7 moves needed for the robot to climb up the ladder. Assume that our simple robot recognises only two commands: climb-up and step-down. Let's code each command in binary: climb-up (1) and step-down (0).



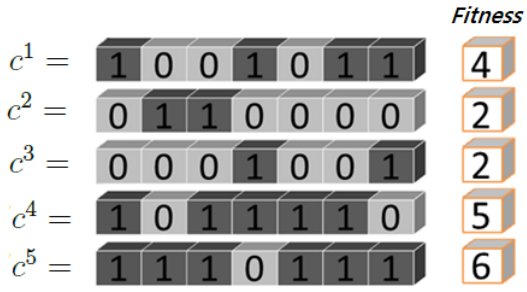
Although simple at first sight, both climbing up one step or staying put over the ground on a ladder, would involve extremely complex tasks for a biped robot: that of maintaining the equilibrium so as not to fall down. These aspects are not considered in our abstract example.

For a human programmer the task is trivial: the solution to the problem is the set of moves $\mathbf{c}^\star = \{1,1,1,1,1,1,1\}$, since any other sequence would fail to reach the top of the ladder. But let's see now how a GA will solve it. For this purpose we need to design a cost function for the problem. Let us call an arbitrary sequence of moves (or candidate solution) as $\mathbf{c} = \{c_1, c_2, \ldots, c_7\}$, with $c_i \in \{0,1\}$ being the variables of the problem. Thus, the optimisation function can be:

$$f(\mathbf{c}) = \sum_{i=1}^{7} c_i$$

Obviously a GA is not the recommended method to solve this trivial problem. Here we are using it for the sake of a didactic illustration of the algorithm.

The latter is actually a widely–used benchmark function in the GA community, known as **OneMax** or **BitCount** cost function. It basically counts the number of bits set in the sequence; its optimal maximum is achieved when all bits are set to one. Let's say the GA generates a random initial population of $m = 5$ candidates with $\ell = 7$ variables. The following picture shows one such population, with fitness for each candidate evaluated using $f(\mathbf{c}^k)$:



In this case, one may think of candidates $\{c^1, c^2, c^3\}$ as the sub–population of green beetles, whereas candidates $\{c^4, c^5\}$ as the sub–population of brown beetles in the didactic illustration at the beginning of the section.

This particular fitness function is actually very easy to implement using Python:

```
fit = pop.sum(axis=1)
print pop, fit
```

```
Out[14]:

[[1 0 1 0 1 0 1 0 1 1]
 [1 0 0 1 1 0 0 1 0 1]
 [0 1 1 0 1 1 1 0 0 0]
 [0 0 1 0 0 0 1 1 0 1]
 [0 0 0 0 1 1 1 0 0 0]] [6 5 5 4 3]
```

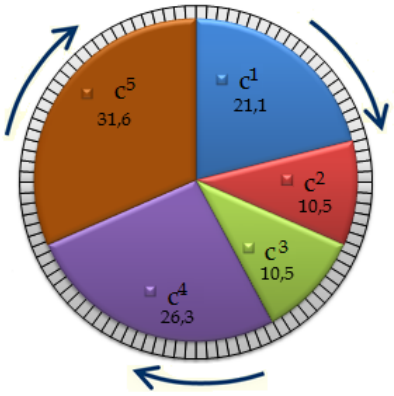
Genetic operators

The genetic operators are the exploration and exploitation techniques used to conduct the search for better solutions. These techniques are inspired in the ideas of the Darwinian theory of natural selection. They are mainly grouped in three operators: selection, reproduction (or crossover) and mutation.

Selection

The natural selection principle states that the fittest individuals in a population are able to reproduce more successfully because their adaptation skills allow them to outlive less skilful individuals. In a GA, the latter means that the fitter the candidate is at optimising the problem (i.e. the higher value it obtains in the fitness function), the better chances it will have to mate with other fit candidates to produce offspring. Such a selection process can be simulated with different techniques that are described below.

- **Tournament:** Two candidates are chosen randomly in a number of rounds (this number is usually half the population size, i.e. $\frac{n}{2}$). In each round, they compete by evaluating their fitness function: the winner is selected and promoted to a temporary pool of parents; the loser is dismissed.
- **Roulette wheel:** As it happens in a casino roulette game, candidates bet for slots in a simulated roulette wheel; slots are allocated in proportion to their fitness (fitter candidates can bet on more slots). Then, the algorithm spins the wheel n times; at every turn the lucky winner is selected and promoted to the pool of parents. It is clear that candidates with bigger slices of the roulette will be at better odds of being selected (a candidate is selected as many times as he wins). To illustrate this method, below is a picture of the roulette wheel corresponding to the binary candidates for the **OneMax** problem showed before.



Here you can see that candidates c^4 and c^5 share the biggest slices as their fitness is closest to that of the optimal solution, $(f(\mathbf{c}^4) = 5, f(\mathbf{c}^5) = 6, f(\mathbf{x}_{\text{best}}) = 7)$

Notice that the size of the slots slices were allocated proportional to the fitness of each candidate, as it can be seen in the following table:

In fact, the proportions in the roulette wheel correspond to probabilities of being selected, as they are computed using the following rule:

$$p(\mathbf{c}^k) = \frac{f(\mathbf{c}^k)}{\sum_{l=1}^{n} f(\mathbf{c}^l)}$$

Name	Chromosome	Fitness	Roulette slice
$\mathbf{c}^1$		4	21.1 %
$\mathbf{c}^2$		2	10.5%
$\mathbf{c}^3$		2	10.5%
$\mathbf{c}^4$		5	26.3%
$\mathbf{c}^5$		6	31.6%
	Sum	19	100%

We can replicate the table above using Python as follows:

```
import pandas as pd
df = pd.DataFrame(pop)
df['fitness'] = fit
df['slice'] = (1.0*fit)/fit.sum()
print "--- Population data ---"
print df
print "--- Sum of data columns ---"
print df.sum(axis=0)
```

```
Out[15]:

--- Population data ---
   0  1  2  3  4  5  6  7  8  9  fitness     slice
0  1  0  1  0  1  0  1  0  1  1        6  0.260870
1  1  0  0  1  1  0  0  1  0  1        5  0.217391
2  0  1  1  0  1  1  1  0  0  0        5  0.217391
3  0  0  1  0  0  0  1  1  0  1        4  0.173913
4  0  0  0  0  1  1  1  0  0  0        3  0.130435

--- Sum of data columns ---
0          2.0
1          1.0
2          3.0
3          1.0
4          4.0
5          2.0
6          4.0
7          2.0
8          1.0
9          3.0
fitness   23.0
slice      1.0
dtype: float64
```

Finally, an easy way to simulate the **Roulette wheel** selection strategy is the `random.choice()` function, which choose randomly with replacement, a number of elements from a set, with a non-uniform probability distribution. For example, the following script chooses 10 colours with a preference towards a warm palette:

```
colors = ['red', 'orange', 'yellow', 'green', 'blue', 'white']
weights = [.3, .25, .25, .1, .05, .05]
np.random.choice(colors, 10, p=weights)
```

```
Out[16]:

array(['red', 'orange', 'green', 'yellow', 'orange', 'white', 'yellow',
       'orange', 'blue', 'red'], dtype='|S6')
```

Thus, selecting the fittest candidates that would be able to breed the next generation can be done as follows (in this example, chromosome number 0 has a slightly greater chance of being selected because its higher fitness value):

```
newpop_id = np.random.choice(pop_size, pop_size, p=df["slice"])
print "Id of selected chromosomes according to fitness distribution:"
print newpop_id
```

```
Out[17]:

Id of selected chromosomes according to fitness distribution:
array([0, 0, 4, 1, 2])
```

Challenge 3.39

Write Python code to simulate the **Tournament** selection strategy.

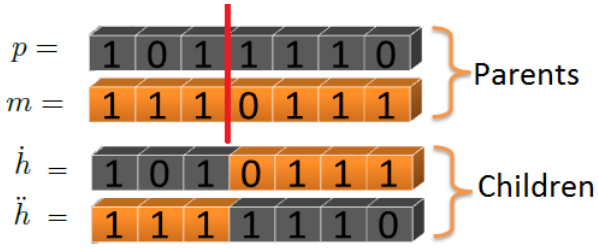
Crossover

The crossover operator aims at combining the genetic material of the selected candidates, in a similar way living organisms reproduce by combining the genes in the chromosomes of their parents. Its motivation is exploitative: selected parents are among the fittest to solve the problem, thus it is expected their offspring to perform equally or even better.

The operator works as follows: two candidates are taken from the pool of parents that were previously promoted by the selection process, as described before. Then, taking into account that mating in real life occurs depending on numerous circumstances (flirting, courting, casual encounters, birth control and so on), the operator is applied conditional on a crossover probability (p_c). This parameter can be fine–tuned in order to obtain different evolutionary results. A value $p_c = 0.7$ is usually a good guess to start experimenting with.

Let us denote an arbitrary pair of father and mother as $p = c^i$ and $m = c^j$ respectively, and assuming mating occurs, their offspring as $\dot{h}$ and $\ddot{h}$. For binary string chromosomes, two popular crossover techniques are the following:

- **One–point crossover:** A random locus in the parents chromosome is chosen, then the resulting segments are exchanged to create two children.



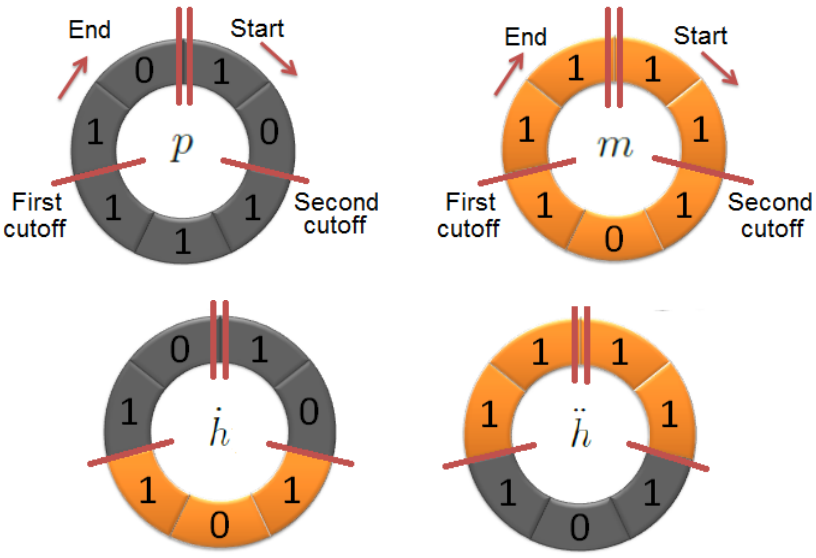
Now, let's write some code in Python to implement **One-point** crossover. Assuming the mother and father are chromosomes number 1 and 2 respectively, we choose a random locus and mix the arrays to produce the offspring:

```
## Choose father, mother and cutoff point ##
father, mother = pop[[1,2]]
loc = np.random.randint(sol_len)

## Breed by mixing the genetic material ##
child1 = np.concatenate((father[:loc], mother[loc:]))
child2 = np.concatenate((mother[:loc], father[loc:]))

## Show crossover results ##
print "Parents  :", father, mother
print "Cutpoint :", loc
print "Offspring:", child1, child2

newpop = np.vstack((mother, father, child1, child2))
newfit = newpop.sum(axis=1)
print "New population and fitness:\n", newpop, newfit
```

```
Out[18]:

Parents  :  [1 0 0 1 1 0 0 1 0 1] [0 1 1 0 1 1 1 0 0 0]
Cutpoint :  7
Offspring:  [1 0 0 1 1 0 0 0 0 0] [0 1 1 0 1 1 1 1 0 1]

New population and fitness:
[[0 1 1 0 1 1 1 0 0 0]
 [1 0 0 1 1 0 0 1 0 1]
 [1 0 0 1 1 0 0 0 0 0]
 [0 1 1 0 1 1 1 1 0 1]] [5 5 3 7]
```

It can be seen that by choosing gene 7 as cut point, `child2` was born a fitter chromosome (its fitness is 7 compared to 5 of their parents). Hence, in the next generation `child2` presumably will have better chances to reproduce than his sibling `child1` which was born with a lower fitness of 3.

- **Two–point crossover:** This operator randomly chooses two cut-off loci, exchanging the genetic string that falls within these two points.



Two–point crossover can be seen as special case of one–point crossover if the flat strings are folded as rings by joining together their end points; the difference being, this operator can combine two not contiguous genetic regions to produce the children.

Challenge 3.40

Write Python code to implement the **Two-point crossover** technique.

Mutation

In contrast to the crossover operator, the aim of mutations is to ensure diversity within the genetic pool of the population. Its motivation is exploratory: small changes are induced in the genetic material of promising candidates in the hope of finding better adaptations. The positive or negative impact of these variations on the fitness of the mutants would be filtered by the selection operator; if they are advantageous for improving the chances of survival, the population would keep the new adaptations when reproducing parents inheriting the mutated traits to their children.

The operator works as follows: a very few candidates from the offspring are selected at random; then a slight variation to some of their genes is performed. Likewise the real life where mutations are rare but happen due to strange circumstances (cosmic rays, sequencing errors during DNA replication and so on), the application of the operator is conditioned on a parameter known as mutation probability (p_m). Again, this is another parameter that can be fine–tuned in order to obtain different evolutionary results. A value $p_m = 0.01$ is usually a good guess to start experimenting with.

For binary string chromosomes, two popular mutation techniques are the following:

- **One bit mutation:** A soon-to-be mutant is chosen with probability p_m, and a bit at a locus picked at random is flipped.
- **Multi–bit mutation:** The entire chromosome of the soon-to-be mutant is traversed, while each locus is flipped or not according to probability p_m.

Let's try a possible Python implementation of the **Multi–bit mutation**. We can use a feature called *binary masking* for this purpose. It works by testing a condition in every location of the array; then only those positions satisfying the condition would be updated. For example, mutation for chromosome number 4 in our population is illustrated below:

```
pm = 0.01
mutant = pop[4]
print "Mutant:␣", mutant
print "Inverted␣mutant:␣", 1-mutant
mask = np.random.uniform(0, 1, size = sol_len) < pm
print "Mask:␣", mask
mutant[mask] = 1-mutant[mask]
print "Updated␣mutant:␣", mutant
```

```
Out[19]:

Mutant:  [0 0 0 0 1 1 1 0 0 0]
Inverted mutant:  [1 1 1 1 0 0 0 1 1 1]
Mask:  [False False False False False  True False  True False False]
Updated mutant:  [0 0 0 0 1 0 1 1 0 0]
```

We can actually take advantage of binary masking to apply **Multi–bit mutation** to the entire population in a single go:

```
pm = 0.1
print "Original␣population:␣\n", pop
mask = np.random.uniform(0,1,size=(pop_size, sol_len)) < pm  # Choose mutant genes
print "Mask:␣\n", mask
newpop = pop                    # Copy  mutant population
newpop[mask] = 1-pop[mask]      # Perform actual mutation
print "Mutated␣population:␣\n", newpop
```

```
Out[20]:

Original population:
[[1 0 1 0 1 0 1 0 1 1]
 [1 0 0 1 1 0 0 1 0 1]
 [0 1 1 0 1 1 1 0 0 0]
 [0 0 1 0 0 0 1 1 0 1]
 [0 0 0 0 1 0 1 1 0 0]]

Mask:
[[False False  True False False False  True False  True False]
 [False False False False False  True  True False  True False]
 [False False False False False  True False  True False  True]
 [False False False  True False False False False  True False]
 [False False False  True False  True False False False False]]

 Mutated population:
[[1 0 0 0 1 0 0 0 0 1]
 [1 0 0 1 1 1 1 1 1 1]
 [0 1 1 0 1 0 1 1 0 1]
 [0 0 1 1 0 0 1 1 1 1]
 [0 0 0 1 1 1 1 1 0 0]]
```

Challenge 3.41

Write Python code to implement the **One bit mutation** technique.

Stopping criteria

The criterion to stop the evolution of a GA is generally determined by the maximum number of generations. There are other more sophisticated techniques tracking population indicators such as the loss of diversity (stagnation) or allocating a maximum number of fitness evaluations, or a combination of these criteria.

Replacement strategies

This concept is related to how the ageing population is replaced by the younger offspring. The are usually two strategies:

- **Generational:** a new generation of individuals is produced from the previous population. This implies that n new children are obtained from n parents by repeated application of selection, crossover and mutation.
- **Steady-state:** Here just a few individuals in each generation are replaced. Typically, a percentage of the total population corresponding to the worst subset of candidates are replaced.

Challenge 3.42

Up to this point we have seen all the conceptual elements and technical ideas needed to develop the GA metaheuristic. Your task is now to complete the code that implements the class `GA`.
Test your class on the **OneMax** problem. Try it with different values of $\ell \in \{10^1, 10^2, 10^3, 10^4, 10^5, 10^6\}$ using the default values for the algorithm parameters. What can you say about runtime behaviour? Or about evolution generations? Or about number of fitness evaluations?

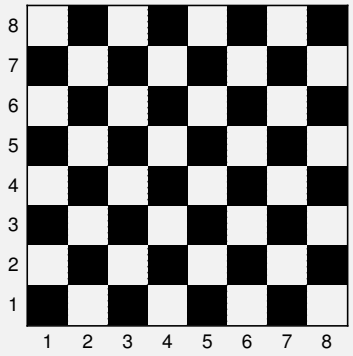
***Hint**: This is a maximisation problem that must be converted to a minimisation problem.*

Challenge 3.43

Similar to **Challenge 3.42**, this time study the effect of the algorithm parameters on **OneMax** problems $\ell \in \{10^3, 10^6\}$. How does the *GA* behaves with varying settings of n, p_c and p_m?

Challenge 3.44

Say you want your *GA* algorithm to design from scratch an official checkers board. The board is actually a 2D matrix with this pattern:



which can be represented in 1D array as follows:

- Define the corresponding fitness function and solve the problem.
- Experiment with different genetic operators and parameter values.

***Hint**: Associate each cell in the board with a bit in the chromosome.*

Challenge 3.45

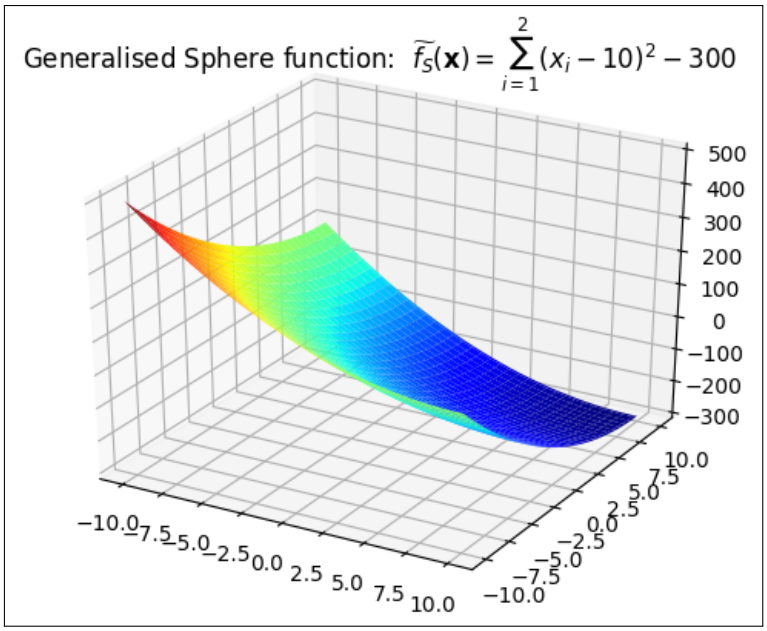
Implement the GA class for numeric domains. Define the solution encoding and genetic operators *Selection*, *Crossover* and *Mutation* for this kind of domain. Use function generalised **Sphere** as a testbed.

Metaheuristics benchmarks

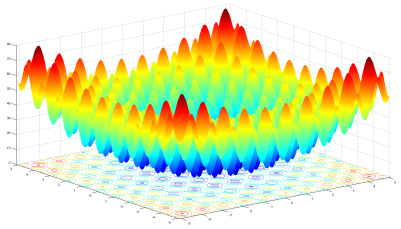
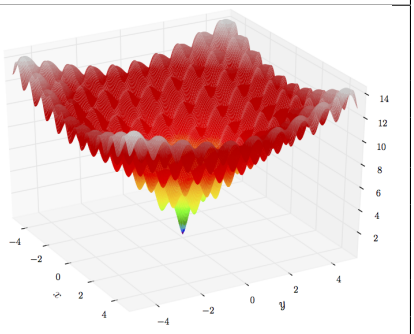
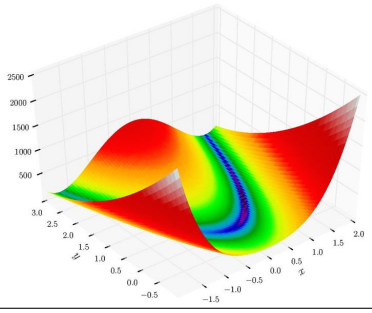
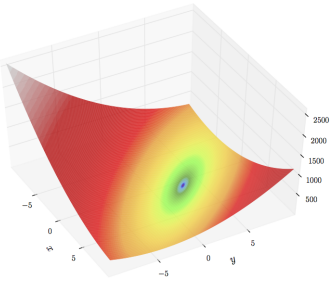
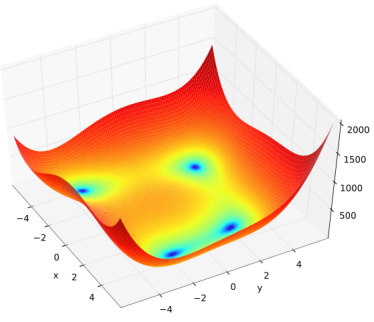
In this section your aim is to learn how to evaluate the behaviour of different metaheuristics in standard optimisation problems, and to compare their performances in terms of effectiveness and efficiency to solve such problems. Let's begin by recalling the generalised **Sphere** cost function definition:

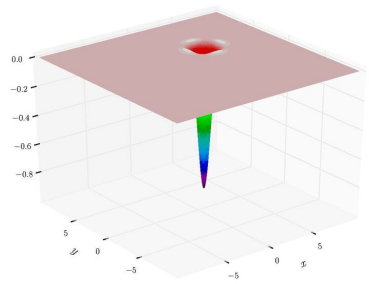
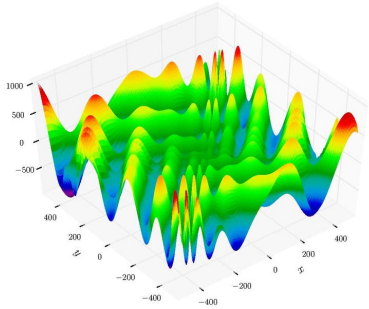
$$\tilde{f}_S(\mathbf{x}, \mathbf{c}, b) = b + \sum_{i=1}^{d} (x_i - c_i)^2.$$

A plot visualising a particular instance of this function is shown below.



We previously verified that even the generalised **Sphere** is a simple problem easily solvable using metaheuristics. Now we will consider more difficult problems. A standard benchmark set for continuous–domain problems is listed in the tables below.

Problem	Definition / Minimum	Plot for $\mathbf{x} \in \mathbb{R}^2$
Rastrigin $-5.12 \leq x_i \leq 5.12$	$f_R(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left(x_i^2 - 10\cos(2\pi x_i)\right)$ $f_R^\star(0,\ldots,0) = 0$	
Ackley $-5 \leq x_1, x_2 \leq 5$	$f_A(\mathbf{x}) = -20\exp\left(-0.2\sqrt{0.5\left(x_1^2 + x_2^2\right)}\right)$ $-\exp\left[0.5\left(\cos 2\pi x_1 + \cos 2\pi x_2\right)\right] + e + 20$ $f_A^\star(0,0) = 0$	
Rosenbrock $-\infty \leq x_i \leq \infty$	$f_K(\boldsymbol{x}) = \sum_{i=1}^{d-1} \left(100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2\right)$ $f_K^\star(1,\ldots,1) = 0$	
Booth $-10 \leq x_1, x_2 \leq 10$	$f_B(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$ $f_B^\star(1,3) = 0$	
Himmelblau $-5 \leq x_1, x_2 \leq 5$	$f_H(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$ $f_H^\star(3.0, 2.0) = 0; \quad f_H^\star(-2.8, 3.1) = 0;$ $f_H^\star(-3.7, -3.2) = 0; \; f_H^\star(3.5, -1.8) = 0.$	

Problem	Definition / Minimum	Plot for $\mathbf{x} \in \mathbb{R}^2$
Easom $-100 \leq x_1, x_2 \leq 100$	$f_E(x_1, x_2) = -\cos(x_1)\cos(x_2)\exp\left(-\left((x_1-\pi)^2 + (x_2-\pi)^2\right)\right)$ $f_E^\star(\pi, \pi) = -1$	
Eggholder $-512 \leq x_1, x_2 \leq 512$	$f_G(x_1, x_2) = -(y+47)\sin\sqrt{\left\lvert \frac{x}{2} + (y+47)\right\rvert} - x\sin\sqrt{\lvert x-(y+47)\rvert}$ $f_G^\star(512, 404.2319) = -959.6407$	

Challenge 3.46

Write function definitions in Python and produce 3D surface plots like the above for each problem $f_S, f_R, f_A, f_K, f_B, f_H, f_E$ and f_G. Besides, produce 2D contour plots. Determine which problems have single, multiple or non local minima, and single or multiple global minima.

NB. *Try using* `plt.contour(...)`, `plt.contourf(...)`, `plt.colorbar()` *for the 2D plots.*

Challenge 3.47

Solve each problem $f_S, f_R, f_A, f_K, f_B, f_H, f_E$ and f_G using the following metaheuristics: *Exhaustive Search*, *Random Search*, *Random Walk*, *Hill Climbing* family, *Simulated Annealing* and *Genetic Algorithm*. Compare their performance regarding the following criteria:

- **Effectiveness.** For example, measuring the *success rate*, i.e. the proportion of runs that actually found the global minimum from a number of repetitions (that is, those runs that did not get stuck into a local minimum); alternatively, you may allow solutions close enough to the global, up to a small tolerance error.
- **Efficiency.** For example, measuring the runtime or number of evaluations needed to find the minimum or a solution close to the minimum within a small tolerance error.

Combining both criteria, which metaheuristic is more suitable for each problem? Which one is best overall the benchmark evaluation?

NB. *Support your answer using visually-friendly tools such as contour and scatter plots, bar and pie diagrams, heatmaps, etc.*

Challenge 3.48

Complete the benchmark of **Challenge 3.47** additionally including the previous results obtained for your own–defined multi–peak functions of **Challenge 3.28**.

Now let's switch to discrete domains and see how to benchmark meta-heuristics in boolean vector–domain functions ($x_i \in \{0,1\}$). The maximisation problems listed in the table below, are commonly used for comparison purposes.

Problem	Definition	Description
All ones (OneMax)	$f(\langle x_1,...,x_n\rangle) = \sum_{i=1}^{n} x_i$	It's the total number of ones in the vector. This is the classic example of a linear problem, where there is no linkage between any of the vector values at all.
Leading ones	$f(\langle x_1,...,x_n\rangle) = \sum_{i=1}^{n}\prod_{j=1}^{i} x_j$	It counts the number of ones in the vector, starting at the beginning, until a zero is encountered. In other words, it returns the position of the first zero found in the vector (minus one).
Leading ones blocks	$f(\langle x_1,...,x_n\rangle) = \left\lfloor \frac{1}{b}\sum_{i=1}^{n}\prod_{j=1}^{i} x_j \right\rfloor$	Given a value b, it counts the number of substrings of ones, each one b bits long, until it sees a zero. Examples: $f(1,1,1,0,0,0,1,0,1) = 1$ $f(1,1,1,1,1,1,0,1,0) = 2$ $f(1,1,1,1,1,1,1,1,1) = 3$

Challenge 3.49

Solve each problem f_{OM}, f_{LO} and f_{LOB} for $n \in \{10, 10^2, \ldots, 10^6\}$, using the following metaheuristics: *Exhaustive Search*, *Random Search*, *Random Walk*, *Hill Climbing* family, *Simulated Annealing* and *Genetic Algorithm*. Compare their performance regarding the following criteria:

- **Effectiveness.** For example, measuring the *success rate*, i.e. the proportion of runs that actually found the global maximum from a number of repetitions (that is, those runs that did not get stuck into a local maximum); alternatively, you may allow solutions close enough to the global, up to a small tolerance error.
- **Efficiency.** For example, measuring the runtime or number of evaluations needed to find the maximum or a solution close to the minimum within a small tolerance error.

Combining both criteria, which metaheuristic is more suitable for each problem? Which one is best overall the benchmark evaluation?

NB. *Again, support your answer using visually-friendly tools such as contour and scatter plots, bar and pie diagrams, heatmaps, etc.*

Challenge 3.50

Come up with two brand-new boolean vector-domain problems, like those aforementioned (for example, the **ZeroMax** problem). Replicate **Challenge 3.49** for these two new problems.

Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDA) are stochastic search techniques that evolve a probability distribution model from a population of solution candidates, rather than evolving the population itself. The distribution is estimated iteratively using the most promising (sub-optimal) solutions until convergence.

The generic estimation procedure is depicted in the algorithm below, which encompasses the following steps. Firstly the parameters of the probability distribution model, $\boldsymbol{\theta}$, are initialised. Then, the algorithm loops over three basic operators in order to update the parameters $\boldsymbol{\theta}$ until convergence. The first operator samples a pool $\mathcal{S}$ of n candidates from the model. The second operator ranks the pool according to the $\mathrm{fitness}(\cdot)$ function and select the most promising solutions into $\mathcal{B}$. The final operator re-estimates the parameters $\boldsymbol{\theta}$ from the $\mathcal{B}$ subset.

Algorithm 10: *EDA*

Input: n: pool size, ℓ: problem dimensionality
Input: $\mathrm{fitness}(\mathbf{x})$: a cost function to optimise, $\mathbf{x} \in \mathbb{R}^\ell$
Output: $\boldsymbol{\theta}$: set of probability model parameters
$\boldsymbol{\theta} \leftarrow \mathsf{initialize}(\ell)$
repeat until θ **converges**
  $\mathcal{S} \leftarrow \mathsf{sample}(P(X;\boldsymbol{\theta}), n)$
  $\mathcal{F} \leftarrow \mathrm{fitness}(\mathcal{S})$
  $\mathcal{B} \leftarrow \mathsf{select}(\mathcal{S}, \mathcal{F})$
  $\boldsymbol{\theta} \leftarrow \mathsf{estimate}(\boldsymbol{\theta}, \mathcal{B})$

The actual realisation of each of those steps determine different types of EDAs: discrete or continuous model parameters; binomial, multinomial or gaussian distributions; univariate, bivariate or multivariate dependencies. For example, the *Population-Based Incremental Algorithm* (*PBIL*) uses a binomial probability distribution to model a population of binary chromosomes. Hence, the parameters of the model are represented as the prototype probability vector $\boldsymbol{\theta} := [\theta_1, \ldots, \theta_\ell]$, where $P(c_i = 1) = \theta_i$ and $P(c_i = 0) = (1 - \theta_i)$. From this prototype vector a population of binary candidate solutions can be sampled by flipping a biased coin to set each gene with bias θ_i.

The algorithm initialises a parameter vector with random values in the unit interval; then a pool of candidates is sampled from a binomial distribution with the parameter vector, and their fitness are computed. After that, the fittest candidates are used to refine the parameters with an incremental learning rule that keeps a fraction of the old values and adds the average of the top–ranked candidates (with learning rate $0 \leq \eta \leq 1$). These steps are iterated until convergence of the parameters. The *PBIL* pseudo-code is summarised below.

Algorithm 11: *PBIL*

Input: n: pool size, ℓ: problem dimensionality, m: selection size
Input: $\mathrm{fitness}(\mathbf{x})$: a cost function to optimise, $\mathbf{x} \in \mathbb{R}^\ell$, η: learning rate
Output: $\boldsymbol{\theta} := \{\theta_i\}_{i=1}^{\ell}$: parameters of Binomial distribution, $0 \leq \theta_i \leq 1$
Output: $\mathbf{x}_{\text{best}}$: the best found minimum
$\boldsymbol{\theta} \leftarrow \mathsf{initialize}([0,1], \ell)$
$\mathbf{x}_{\text{best}} \leftarrow \mathrm{create}()$
repeat until θ **converges**
  $\mathcal{S} \leftarrow \mathsf{sample}(Binomial(\boldsymbol{\theta}), n)$
  $\mathcal{F} = \mathsf{fitness}(\mathcal{S})$
  $\mathcal{B} \leftarrow \mathsf{select}(\mathcal{S}, \mathcal{F}, m)$
  $\boldsymbol{\theta} = (1 - \eta)\boldsymbol{\theta} + \eta \frac{1}{m} \sum_i \mathbf{x}_i, \quad \mathbf{x}_i \in S$
  $\mathbf{x}_{\text{best}} \leftarrow \mathsf{update\_best}(\mathcal{B}, \mathcal{F}, \mathbf{x}_{\text{best}})$

In order to implement PBIL in Python, we shall firstly define the generic EDA class:

```
class GenericEDA():

    ## Setup EDA basic variables ##
    def setup(self, cand_size=20, max_evals=100, selection_rate=0.5, learning_rate=1.0,
           cost_func=None):
        self.cand_size = cand_size
        self.sample_size = cand_size
        self.max_evals = max_evals
        self.selection_rate = selection_rate
        self.learning_rate = learning_rate
        self.cost_func = cost_func
        self.distr = None

     def sample(self, sample_size, top_ranked, best):
        return self.distr.sample(sample_size)

     def get_top_ranked(self, candidates):
        fits = self.cost_func(candidates)
        index = np.argsort(fits)[:(self.cand_size*self.selection_rate/100):-1]
        return candidates[index],
         Solution(candidates[index[0]],fits[index[0]])

    ## Optimisation algorithm ##
     def optimise(self):
         best = Solution(None, float('-Inf'))
         top_ranked = None
         while not self.distr.has_converged():
             candidates=self.sample(self.sample_size, top_ranked, best)
             top_ranked, winner = self.get_top_ranked(candidates)
             self.estimate(top_ranked, best)
             if best.cost < winner.cost:
                 best = winner
             self.iters += 1
         return best

     ## Abstract method ##
     def estimate(self, candidates, best):
         raise NotImplementedError()
```

Now we can easily derive the `PBIL` class using the inheritance mechanism:

```
class PBIL(GenericEDA):

 def initialize(self):
   self.distr = Binomial(self.var_size)

 def estimate(self, top_ranked, best):
   self.distr.p = self.distr.p*(1-self.learning_rate)
      + self.learning_rate * np.average(top_ranked)
```

Challenge 3.51

Evaluate the performance of *PBIL* in solving the discrete domain problems of **Challenge 3.49**. Show how *PBIL* compares to *GA* with respect to effectiveness and efficiency.

Another variation of an EDA, the so-called *Compact GA (cGA)*, is designed to save memory costs in the estimation step of the algorithm, which can be an important consideration when dealing with large–scale problems (those with a high number of decision variables).

cGA resembles *PBIL*, the only difference is the estimation rule which is designed in a *compact* mode: instead of using a batch of promising candidates to refine the distribution parameters, this algorithm updates said parameters by sampling only two candidates at a time. This update is repeated m times to take into account the contributions of the top–ranked candidates in the population; the latter resembles the mechanism of **Tournament Selection** used in the canonical *GA*. Below a pseudocode of *cGA* is shown.

Algorithm 12: *cGA*

Input: n: population size, ℓ: problem dimensionality
Input: $\text{fitness}(\mathbf{x})$: a cost function to optimise, $\mathbf{x} \in \mathbb{R}^{\ell}$
Output: $\boldsymbol{\theta} := \{\theta_i\}_{i=1}^{\ell}$: parameters of Binomial distribution, $0 \leq \theta_i \leq 1$
Output: $\mathbf{x}_{\text{best}}$: the best found minimum
$\boldsymbol{\theta} \leftarrow \text{initialize}([0,1], \ell)$
$\mathbf{x}_{\text{best}} \leftarrow \text{create}()$
repeat until θ **converges**
  repeat n **times**
    $\{\mathbf{x}_1, \mathbf{x}_2\} \leftarrow \text{sample}(Binomial(\boldsymbol{\theta}), 2)$
    $\{\mathbf{x}_{\text{win}}, \mathbf{x}_{\text{fail}}\} \leftarrow \text{compete}(\text{fitness}(\mathbf{x}_1), \text{fitness}(\mathbf{x}_2))$
    $\boldsymbol{\theta} = \boldsymbol{\theta} + \frac{1}{n}(\mathbf{x}_{\text{win}} - \mathbf{x}_{\text{fail}})$
    $\mathbf{x}_{\text{best}} \leftarrow \text{update\_best}(\mathbf{x}_{\text{win}}, \mathbf{x}_{\text{best}})$

The following Python script implements the `cGA` class.

```
class cGA(GenericEDA):
   def initialize(self):
       self.distr=Binomial(self.var_size)
       self.learning_rate=1.0/float(self.pop_size)
       self.sample_size=2

   def estimate(self, (winner, loser), best):
       self.distr.p =
         np.minimum(np.ones((1, self.var_size)),
           np.maximum(np.zeros((1, self.var_size)),
           self.distr.p + (winner.params-loser.params)*self.learning_rate))

   def get_top_ranked(self, candidates):
       costs = self.cost_func(candidates)
       maxindx = np.argmax(costs)
       winner = Solution(candidates[maxindx], costs[maxindx])
       loser = Solution(candidates[not maxindx], costs[not maxindx])
       return  (winner, loser), winner
```

Challenge 3.52

Repeat **Challenge 3.51** this time evaluating the performance of *cGA*. Show how it compares to *PBIL* and *GA* regarding effectiveness and efficiency.

Suggested readings

↓ Contemporary books

- Sean Luke. *Essentials of Metaheuristics*, 2nd. Edition, lulu.com, 2015.
- Bozorg-Haddad, Solgi & Loiciga. *Metaheuristic and Evolutionary Algorithms for Engineering Optimization*, Wiley, 2017.
- Patrick Siarry. *Metaheuristics*, Springer, 2016.
- Ke-Lin Du & M. N. S. Swamy. *Search and Optimization by Metaheuristics*, Birkhäuser, 2016.

↓ Classic books

- John Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, 1975.
- David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
- Melanie Mitchell. *An Introduction to Genetic Algorithms*, MIT Press, 1998.

—— THE END ——