MINISTRY EDUCATION AND SCIENCES UKRAINE
NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
"IGOR SIKORSKY KYIV
POLYTECHNIC INSTITUTE"

Gordienko Yu.G., Kochura Yu.P.

BASICS OF EVOLUTIONARY COMPUTING

Labs

Tutorial
for master's degree holders
according to the educational program "Software engineering of computer systems»
specialties 121 "Software engineering"
according to the educational program "Computer systems and networks»
specialty 123 "Computer engineering"
according to the educational program "Information management systems and technologies»
specialties 126 "Information systems and technologies»

Electronic educational publication

APPROVED
at the meeting of Computer Engineering department,
protocol No. 10 on 05/25/2022

2022

Lab 1 - Introduction to Genetic Algorithms - OneMax problem

(short version with Hall of Fame) based on (C) Eyal Wirsansky work

In this lab we introduce **DEAP — a powerful and flexible evolutionary computation** framework capable of solving real-life problems using **genetic algorithms (GA)**.

Brief Content:

- introduction,
- installation,
- main modules: *creator* and *toolbox*,
- components needed for the GA workflow,
- the simplest example, *the OneMax problem*, so called the Hello World of genetic algorithms.

By the end of this lab you will know:

- the DEAP framework and its modules,
- the concepts of creator and toolbox in the DEAP framework,
- the simplest example of GA,
- how to create a GA solution using the DEAP framework,
- how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the OneMax problem using a GA coded with the DEAP framework,
- how to experiment with various settings of the GA and interpret the differences in the results.

Installation and import of libraries

In these and other labs, we will use various Python packages:

- [NumPy]
- [Matplotlib]
- [Seaborn]

They are already pre-installed in Colab. Let's import them by the following code.

```
# Import all necessary standard libraries
import random
import numpy
import matplotlib.pyplot as plt
import seaborn as sns
```

Install DEAP by *pip* with the following code:

```
# Install DEAP
!pip install deap
```

```
Requirement already satisfied: deap in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-package
```

```
# Import DEAP
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
```

Example: OneMax problem

Constants

```
# Let's declare constants that set the parameters for the problem and control the

# problem constants:
ONE_MAX_LENGTH = 100  # length of bit string to be optimized

# GA constants:
POPULATION_SIZE = 200
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.1   # probability for mutating an individual
MAX_GENERATIONS = 50
```

Reproducibility of Results

One important aspect of the GA is the use of probability, which introduces a random element to the behavior of the algorithm.

However, **for reproducibility of results**, when experimenting with the code, we may want to be able to run the same experiment several times and get repeatable results.

To accomplish this, we set the random function seed to a constant number of some value, as shown in the following code:

```
# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

Toolbox class

The **Toolbox** class is used as a container for functions (or operators), and enables us to create new operators by aliasing and customizing existing functions.

```
toolbox = base.Toolbox()
```

```
# For example, suppose we have a function, multiply() , defined as follows:
def multiply(a, b):
  return a*b

# Using toolbox, we can now create a new operator, incrementByFive(),
# which customizes the sumOfTwo() function as follows:
toolbox.register("MultiplyBy", multiply, b=5)

# examples:
A = toolbox.MultiplyBy(10)
print('toolbox.MultiplyBy(10) =', A)
B = multiply(10,5)
print('multiply(10,5) =', B)
```

```
toolbox.MultiplyBy(10) = 50
multiply(10,5) = 50
```

Let's create the *zeroOrOne* operator, which customizes the *random.randomint(a, b)* function.

This function normally returns a random integer *N* such that $a \leq N \leq b$.

By fixing the two arguments, *a* and *b*, to the values 0 and 1 the *zeroOrOne* operator will randomly return either the value 0 or the value 1 when called later in the code.

```
# create an operator that randomly returns 0 or 1:
toolbox.register("zeroOrOne", random.randint, 0, 1)

# examples:
A = toolbox.zeroOrOne()
print('zeroOrOne =', A)
B = toolbox.zeroOrOne()
print('zeroOrOne =', B)
C = toolbox.zeroOrOne()
print('zeroOrOne =', C)
D = toolbox.zeroOrOne()
print('zeroOrOne =', D)
```

```
zeroOrOne = 0
```

```
zeroOrOne = 0
zeroOrOne = 1
zeroOrOne = 0
```

Fitness class

Next, we need to create the *Fitness* class. Since we only have one objective here—the sum of digits—and our goal is to maximize it, we choose the FitnessMax strategy, using a weights tuple with a single positive weight, as shown in the following code.

```
# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

```
A = base.Fitness.weights
print(A)
```

```
None
```

In DEAP, the *Individual* class is used to represent each of the population's individuals. This class is created with the help of the creator tool. In our case, list serves as the base class, which is used as the individual's chromosome. The class is augmented with the fitness attribute, initialized to the *FitnessMax* class that we defined earlier

```
# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)
#creator.create("Individual", array.array, typecode='b', fitness=creator.FitnessMa
```

Next, register the *individualCreator* operator, which creates an instance of the *Individual* class, filled up with random values of either 0 or 1 . This is done by customizing the previously defined *zeroOrOne* operator.

Since the objects generated by the *zeroOrOne* operator are integers with random values of either 0 or 1 , the resulting *individualCreator* operator will fill an *Individual* instance with 100 randomly generated values of 0 or 1.

```
# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator",   # Register the individualCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 creator.Individual,    # The container type (Individual) in which
                 toolbox.zeroOrOne,     # The function used to generate objects (=
                 ONE_MAX_LENGTH)        # The number of objects we want to generat
```

Register the *populationCreator* operator that creates a list of individuals.

```
# create the population operator to generate a list of individuals:
toolbox.register("populationCreator",   # Register the populationCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 list,                  # The container type (list) in which the r
                 toolbox.individualCreator) # The function used to generate objects
```

Define the function *oneMaxFitness* that computes the number of 1s in the individual.

```
# fitness calculation:
# compute the number of '1's in the individual
def oneMaxFitness(individual):
    return sum(individual),  # return a tuple,
                             # fitness values in DEAP are represented as tuples,
                             # and therefore a comma needs to follow when a single
```

Define the *evaluate* operator as an alias to the *oneMaxfitness()* function we defined earlier.

```
# create the evaluate alias for calculating the fitness (by a DEAP convention)
toolbox.register("evaluate", oneMaxFitness)
```

Genetic operators

The genetic operators are typically created by aliasing existing functions from the tools module and setting the argument values as needed.

Note: The *mutFlipBit* function iterates over all the attributes of the individual, a list with values of 1s and 0s in our case, and for each attribute will use the argument value (*indpb* parameter) as the probability of flipping (applying the not operator to) the attribute value. This value is independent of the mutation probability, which is set by the *P_MUTATION* constant that we defined earlier and has not yet been used. The mutation probability serves to decide if the *mutFlipBit* function is called for a given individual in the population.

```
# genetic operators:

# Tournament selection with tournament size of 3:
toolbox.register("select", tools.selTournament, tournsize=3)

# Single-point crossover:
toolbox.register("mate", tools.cxOnePoint)

# Flip-bit mutation:
# indpb: Independent probability for each attribute to be flipped
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/ONE_MAX_LENGTH)
```

GA workflow

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)
```

Short version with 'Hall of Fame' (HoF)

Let's consider the additional feature of the built-in *algorithms.eaSimple* method - the hall of fame (HoF). It is implemented as *HallOfFame* class that can be used to retain the best individuals that ever existed in the population during the evolution, even if they have been lost at some point due to selection, crossover, or mutation. HoF is continuously sorted so that the first element is **the first individual** that had **the best fitness value** ever seen.

```
# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
HALL_OF_FAME_SIZE = 10
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

# perform the Genetic Algorithm flow with hof feature added:
population, logbook = algorithms.eaSimple(population, toolbox, cxpb=P_CROSSOVER, mu
                                          ngen=MAX_GENERATIONS, stats=stats, ha

# print Hall of Fame info:
print("Hall of Fame Individuals = ", *hof.items, sep="\n")
print("Best Ever Individual = ", hof.items[0])

# Genetic Algorithm is done - extract statistics:
maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")
```

```
gen	nevals	max	avg
0  	200   	60 	49.705
1  	190   	68 	53.56 
2  	175   	67 	56.87 
3  	179   	69 	60.21 
4  	175   	72 	62.825
5  	184   	71 	65.45 
6  	178   	76 	67.68 
7  	187   	80 	69.865
8  	189   	81 	72.055
9  	184   	84 	74.765
10 	185   	85 	77.515
11 	181   	86 	79.485
12 	190   	87 	81.49 
13 	181   	89 	83.27 
14 	184   	89 	84.94 
15 	189   	90 	86.22 
16 	176   	90 	87.725
17 	176   	91 	88.79 
```

```
18      182     92      89.485
19      185     93      90.065
20      182     94      90.765
21      170     94      91.535
22      179     94      92.28
23      178     95      92.985
24      181     95      93.545
25      189     95      93.855
26      174     96      94.125
27      179     96      94.36
28      186     96      94.78
29      185     96      95.055
30      185     97      95.43
31      186     97      95.775
32      187     97      96.075
33      179     97      96.435
34      176     98      96.745
35      187     98      96.885
36      186     98      96.93
37      190     98      97.015
38      175     98      97.245
39      171     98      97.515
40      179     98      97.78
41      188     98      97.845
42      188     98      97.87
43      178     99      97.925
44      174     99      97.95
45      176     99      97.87
46      185     99      98.04
47      184     99      98.14
48      184     99      98.37
49      187     99      98.79
50      185     99      98.885
Hall of Fame Individuals =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

You should get the following output:

gen nevals max avg

0 200 60 49.705

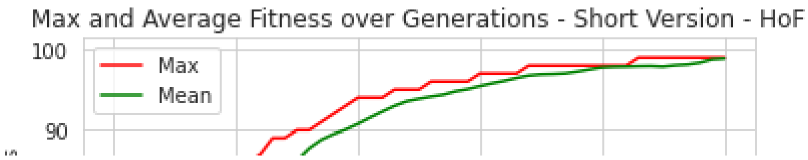
...

50 185 99 98.885

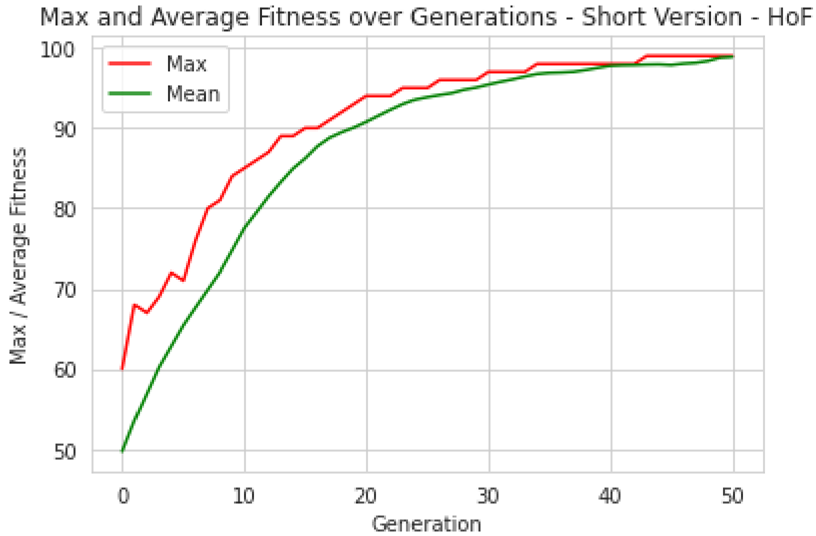
Hall of Fame Individuals = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Best Ever Individual = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

```
# Plot statistics:
sns.set_style("whitegrid")
plt.plot(maxFitnessValues, color='red', label='Max')
plt.plot(meanFitnessValues, color='green', label='Mean')
plt.xlabel('Generation')
plt.ylabel('Max / Average Fitness')
plt.title('Max and Average Fitness over Generations - Short Version - HoF')
plt.legend()
plt.show()
```



You should get the following output:



[Colab paid products](#) - [Cancel contracts here](#)

Evolutionary Algorithms (EA) Basics

Lab 2 - Applications of EA based on (C) Eyal Wirsansky work

Brief Content:

- DEAP installation (**every time after start of Colab VM!**),
- main modules: *creator* and *toolbox*,
- components needed for the GA workflow,
- *Knapsack problem*,
- *Traveling Saleperson Problem*.

By the end of this lab you will know:

- again, how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the Knapsack problem using a GA coded with the DEAP framework,
- how to experiment with various settings of the GA and interpret the differences in the results.

Installation and import of libraries

IMPORTANT: Mount your Google Drive!

At left sidebar -> click "Files" icon, the click "Mount Drive" icon with Google Drive logo, follow instructions.

```
# Copy all lab-related materials from Google Drive to your current location at Goo
! cp -r /content/drive/MyDrive/COLAB_EVO/EVO_Lab01/* .
```

```
# Check the folders/files copied
! ls
```

```
drive        EVO_Lab01              knapsack.py  tsp.py
elitism.py   EVO_Lab01_A_bag.ipynb  sample_data  vrp.py
```

In these and other lectures, we will use various Python packages:

- [NumPy]
- [Matplotlib]

- Seaborn

They are already pre-installed in Colab. Let's import them by the following code.

```
# Import all necessary standard libraries
import random
import numpy as np

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

Install DEAP by *pip* with the following code:

```
# Install DEAP
!pip install deap
```

```
Requirement already satisfied: deap in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-package
```

```
# Import DEAP
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
```

Here the difference is started for these new examples!

Example 1: Knapsack problem (0-1 type)

Rosetta Code's Description:

A tourist wants to make a good trip at the weekend with his friends.

They will go to the mountains to see the wonders of nature, so he needs to pack well for the trip.

He has a good knapsack for carrying things, but knows that he can carry a maximum of only 4kg in it, and it will have to last the whole day.

He creates a list of what he wants to bring for the trip but the total weight of all items is too much.

He then decides to add columns to his initial list detailing their weights and a numerical value representing how important the item is for the trip.

The tourist can choose to take any combination of items from the list, but only one of each item is available.

He may not cut or diminish the items, so he can only take whole units of any item.

Task

Show which items the tourist can carry in his knapsack so that their total weight does not exceed 400 dag [4 kg], and their total value is maximized. Note: [dag = decagram = 10 grams]

```
# knapsack example-specific library
import knapsack
```

Constants

```
# Let's declare constants that set the parameters for the problem and control the

# problem constants:
# create the knapsack problem instance to be used:
knapsack = knapsack.Knapsack01Problem()

# GA constants:
POPULATION_SIZE = 50
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.1   # probability for mutating an individual
MAX_GENERATIONS = 50
```

Reproducibility of Results

One important aspect of the GA is the use of probability, which introduces a random element to the behavior of the algorithm.

However, **for reproducibility of results**, when experimenting with the code, we may want to be able to run the same experiment several times and get repeatable results.

To accomplish this, we set the random function seed to a constant number of some value, as shown in the following code:

```
# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

Toolbox class

The **Toolbox** class is used as a container for functions (or operators), and enables us to create new operators by aliasing and customizing existing functions.

```
toolbox = base.Toolbox()
```

```
# For example, suppose we have a function, multiply() , defined as follows:
def multiply(a, b):
  return a*b

# Using toolbox, we can now create a new operator, incrementByFive(),
# which customizes the sumOfTwo() function as follows:
toolbox.register("MultiplyBy", multiply, b=5)

# examples:
A = toolbox.MultiplyBy(10)
print('toolbox.MultiplyBy(10) =', A)
B = multiply(10,5)
print('multiply(10,5) =', B)
```

```
toolbox.MultiplyBy(10) = 50
multiply(10,5) = 50
```

Let's create the *zeroOrOne* operator, which customizes the *random.randomint(a, b)* function.

This function normally returns a random integer *N* such that $a \leq N \leq b$.

By fixing the two arguments, *a* and *b*, to the values 0 and 1 the *zeroOrOne* operator will randomly return either the value 0 or the value 1 when called later in the code.

```
# create an operator that randomly returns 0 or 1:
toolbox.register("zeroOrOne", random.randint, 0, 1)

# examples:
A = toolbox.zeroOrOne()
print('zeroOrOne =', A)
B = toolbox.zeroOrOne()
print('zeroOrOne =', B)
C = toolbox.zeroOrOne()
print('zeroOrOne =', C)
D = toolbox.zeroOrOne()
print('zeroOrOne =', D)
```

```
zeroOrOne = 0
zeroOrOne = 0
zeroOrOne = 1
zeroOrOne = 0
```

Fitness class

Next, we need to create the *Fitness* class. Since we only have one objective here—the sum of digits—and our goal is to maximize it, we choose the FitnessMax strategy, using a weights tuple with a single positive weight, as shown in the following code.

```
# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

```
A = base.Fitness.weights
print(A)
```

```
None
```

In DEAP, the *Individual* class is used to represent each of the population's individuals. This class is created with the help of the creator tool. In our case, list serves as the base class, which is used as the individual's chromosome. The class is augmented with the fitness attribute, initialized to the *FitnessMax* class that we defined earlier

```
# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)
```

```
/usr/local/lib/python3.6/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
```

Next, register the *individualCreator* operator, which creates an instance of the *Individual* class, filled up with random values of either 0 or 1 . This is done by customizing the previously defined *zeroOrOne* operator.

Since the objects generated by the *zeroOrOne* operator are integers with random values of either 0 or 1 , the resulting *individualCreator* operator will fill an *Individual* instance with 100 randomly generated values of 0 or 1.

```
# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator",   # Register the individualCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 creator.Individual,    # The container type (Individual) in which
                 toolbox.zeroOrOne,     # The function used to generate objects (=
                 len(knapsack))        # The number of objects we want to generate
```

Register the *populationCreator* operator that creates a list of individuals.

```
# create the population operator to generate a list of individuals:
toolbox.register("populationCreator",   # Register the populationCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 list,                  # The container type (list) in which the r
                 toolbox.individualCreator) # The function used to generate object
```

Fitness function

Define the function *knapsackValue* that computes the fitness.

```
# fitness calculation:
# compute the number of '1's in the individual
```

```
def knapsackValue(individual):
    return knapsack.getValue(individual),  # return a tuple,
                                           # fitness values in DEAP are represente
                                            # and therefore a comma needs to follo
```

Define the *evaluate* operator as an alias to the *knapsackValue()* function we defined earlier.

```
# create the evaluate alias for calculating the fitness (by a DEAP convention)
toolbox.register("evaluate", knapsackValue)
```

Genetic operators

The genetic operators are typically created by aliasing existing functions from the tools module and setting the argument values as needed.

Note: The *mutFlipBit* function iterates over all the attributes of the individual, a list with values of 1s and 0s in our case, and for each attribute will use the argument value (*indpb* parameter) as the probability of flipping (applying the not operator to) the attribute value. This value is independent of the mutation probability, which is set by the *P_MUTATION* constant that we defined earlier and has not yet been used. The mutation probability serves to decide if the *mutFlipBit* function is called for a given individual in the population.

```
# genetic operators:

# Tournament selection with tournament size of 3:
toolbox.register("select", tools.selTournament, tournsize=3)

# Single-point crossover:
toolbox.register("mate", tools.cxOnePoint)

# or
# Two-point crossover:
#toolbox.register("mate", tools.cxTwoPoint)

# Flip-bit mutation:
# indpb: Independent probability for each attribute to be flipped
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/len(knapsack))
```

GA workflow

Create population of individual solutions

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)
```

```
# let's check the population created...
population
```

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

```
# let's check the items that we should put in the knapsack
# ... the items in the population created...
```

```
# the 1st individual solution is ...
knapsack.printItems(population[0])
```

```
- Adding map: weight = 9, value = 150, accumulated weight = 9, accumulated va
- Adding water: weight = 153, value = 200, accumulated weight = 162, accumula
- Adding banana: weight = 27, value = 60, accumulated weight = 189, accumulat
- Adding beer: weight = 52, value = 10, accumulated weight = 241, accumulated
- Adding camera: weight = 32, value = 30, accumulated weight = 273, accumulat
- Adding t-shirt: weight = 24, value = 15, accumulated weight = 297, accumula
- Adding trousers: weight = 48, value = 10, accumulated weight = 345, accumul
- Adding waterproof trousers: weight = 42, value = 70, accumulated weight = 3
- Adding socks: weight = 4, value = 50, accumulated weight = 391, accumulated
- Total weight = 391, Total value = 595
```

```
# the last individual solution is ...
knapsack.printItems(population[-1])
```

```
- Adding map: weight = 9, value = 150, accumulated weight = 9, accumulated va
- Adding sandwich: weight = 50, value = 160, accumulated weight = 59, accumul
- Adding glucose: weight = 15, value = 60, accumulated weight = 74, accumulat
- Adding banana: weight = 27, value = 60, accumulated weight = 101, accumulat
- Adding apple: weight = 39, value = 40, accumulated weight = 140, accumulate
- Adding cheese: weight = 23, value = 30, accumulated weight = 163, accumulat
- Adding suntan cream: weight = 11, value = 70, accumulated weight = 174, acc
- Adding camera: weight = 32, value = 30, accumulated weight = 206, accumulat
- Adding t-shirt: weight = 24, value = 15, accumulated weight = 230, accumula
- Adding trousers: weight = 48, value = 10, accumulated weight = 278, accumul
- Adding umbrella: weight = 73, value = 40, accumulated weight = 351, accumul
- Adding waterproof overclothes: weight = 43, value = 75, accumulated weight
- Total weight = 394, Total value = 740
```

Short version with 'Hall of Fame' (HoF)

Let's consider the additional feature of the built-in *algorithms.eaSimple* method - the hall of fame (HoF). It is implemented as *HallOfFame* class that can be used to retain the best individuals that ever existed in the population during the evolution, even if they have been lost at some point due to selection, crossover, or mutation. HoF is continuously sorted so that the first element is **the first individual** that had **the best fitness value** ever seen.

```
# define the hall-of-fame object:
HALL_OF_FAME_SIZE = 1
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
```

```
# let's chekc the initial state of HoF
hof.items
```

```
[]
```

Start workflow

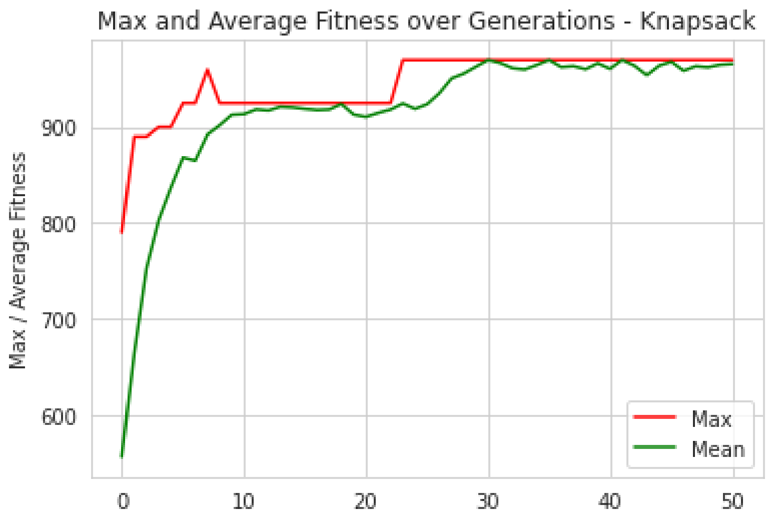
```
# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# perform the Genetic Algorithm flow with hof feature added:
population, logbook = algorithms.eaSimple(population, toolbox, cxpb=P_CROSSOVER, m
                                          ngen=MAX_GENERATIONS, stats=stats, h

# print Hall of Fame info:
print("Hall of Fame Individuals = ", *hof.items, sep="\n")
print("Best Ever Individual = ", hof.items[0])

print("-- Knapsack Items = ")
knapsack.printItems(hof.items[0])

# Genetic Algorithm is done - extract statistics:
maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")
```

```
gen	nevals	max	avg
0  	50    	790	555.56
1  	42    	890	663.56
2  	40    	890	752.36
3  	44    	900	802.64
4  	50    	900	836.84
5  	41    	925	868.4
6  	44    	925	865.1
7  	42    	960	892.2
8  	42    	925	901.74
9  	48    	925	912.7
10 	46    	925	913.34
11 	48    	925	918.3
12 	41    	925	917.2
13 	45    	925	921.1
14 	46    	925	920.4
15 	46    	925	918.9
16 	46    	925	917.7
17 	48    	925	918.2
18 	44    	925	924
19 	50    	925	912.9
20 	46    	925	910.5
21 	50    	925	914.5
22 	47    	925	918.2
23 	40    	970	924.5
24 	45    	970	918.9
25 	46    	970	923.8
26 	50    	970	935.3
27 	45    	970	950.8
28 	45    	970	955.8
29 	46    	970	963
30 	42    	970	970
31 	46    	970	966.8
32 	43    	970	961.5
33 	49    	970	960.1
34 	49    	970	964.8
```

```
35      38      970     970
36      42      970     962.6
37      46      970     963.6
38      44      970     960.4
39      43      970     966.5
40      48      970     960.8
41      46      970     970
42      46      970     964.1
43      46      970     954.3
44      48      970     964.1
45      42      970     968.1
46      50      970     958.8
47      41      970     963.5
48      50      970     962.4
49      50      970     965
50      44      970     965.8
Hall of Fame Individuals =
[1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1]
Best Ever Individual =  [1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1,
-- Knapsack Items =
- Adding map: weight = 9, value = 150, accumulated weight = 9, accumulated
```

Plot results

```
# Plot statistics:
sns.set_style("whitegrid")
plt.plot(maxFitnessValues, color='red', label='Max')
plt.plot(meanFitnessValues, color='green', label='Mean')
plt.xlabel('Generation')
plt.ylabel('Max / Average Fitness')
plt.title('Max and Average Fitness over Generations - Knapsack')
plt.legend()
plt.show()
```



Example 2: Traveling Salesman Problem

[Wikipedia Description]

"Given a list of cities and the distances between each pair of the cities, find the shortest possible path that goes through all the cities, and returns to the starting city."

Details

City coordinates are read from an online file and distance matrix is calculated. The data is serialized to disk.The total distance can be calculated for a path represented by a list of city indices. A plot can be created for a path represented by a list of city indices.

```
# tsp example-specific library
import tsp
```

Constants

```
# Let's declare constants that set the parameters for the problem and control the

# problem constants:
# create the desired traveling salesman problem instace:
TSP_NAME = "bayg29"  # name of problem
tsp = tsp.TravelingSalesmanProblem(TSP_NAME)

# GA constants:
POPULATION_SIZE = 300
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.1   # probability for mutating an individual
MAX_GENERATIONS = 200
```

Reproducibility of Results

One important aspect of the GA is the use of probability, which introduces a random element to the behavior of the algorithm.

However, **for reproducibility of results**, when experimenting with the code, we may want to be able to run the same experiment several times and get repeatable results.

To accomplish this, we set the random function seed to a constant number of some value, as shown in the following code:

```
# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

Toolbox class

The **Toolbox** class is used as a container for functions (or operators), and enables us to create new operators by aliasing and customizing existing functions.

```
toolbox = base.Toolbox()
```

Fitness class

Next, we need to create the *Fitness* class. Since we only have one objective here—the sum of digits—and our goal is to maximize it, we choose the FitnessMax strategy, using a weights tuple with a single positive weight, as shown in the following code.

```
# define a single objective, minimizing fitness strategy:
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

In DEAP, the *Individual* class is used to represent each of the population's individuals. This class is created with the help of the creator tool. In our case, list serves as the base class, which is used as the individual's chromosome. The class is augmented with the fitness attribute, initialized to the *FitnessMax* class that we defined earlier

```
import array
# create the Individual class based on list:
creator.create("Individual", array.array, typecode='i', fitness=creator.FitnessMin
```

```
# create an operator that generates randomly shuffled indices:
toolbox.register("randomOrder", random.sample, range(len(tsp)), len(tsp))
```

```
# create the individual creation operator to fill up an Individual instance with s
toolbox.register("individualCreator", tools.initIterate, creator.Individual, toolb
```

Register the *populationCreator* operator that creates a list of individuals.

```
# create the population operator to generate a list of individuals:
toolbox.register("populationCreator",   # Register the populationCreator operator,
                 tools.initRepeat,      # The initRepeat operator is customized he
                 list,                  # The container type (list) in which the r
                 toolbox.individualCreator) # The function used to generate object
```

Fitness function

Define the function *knapsackValue* that computes the fitness.

```
# fitness calculation:
# compute the total distance of the list of cities represented by indices:
def tpsDistance(individual):
    return tsp.getTotalDistance(individual),  # return a tuple,
                                              # fitness values in DEAP are represente
                                               # and therefore a comma needs to follo
```

Define the *evaluate* operator as an alias to the *knapsackValue()* function we defined earlier.

```
# create the evaluate alias for calculating the fitness (by a DEAP convention)
toolbox.register("evaluate", tpsDistance)
```

Genetic operators

The genetic operators are typically created by aliasing existing functions from the tools module and setting the argument values as needed.

Note: The *mutFlipBit* function iterates over all the attributes of the individual, a list with values of 1s and 0s in our case, and for each attribute will use the argument value (*indpb* parameter) as the probability of flipping (applying the not operator to) the attribute value. This value is independent of the mutation probability, which is set by the *P_MUTATION* constant that we defined earlier and has not yet been used. The mutation probability serves to decide if the *mutFlipBit* function is called for a given individual in the population.

```
# genetic operators:

# Tournament selection with tournament size of 3:
toolbox.register("select", tools.selTournament, tournsize=3)

# Ordered crossover
toolbox.register("mate", tools.cxOrdered)

# Shuffle mutation:
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=1.0/len(tsp))
```

GA workflow

Create population of individual solutions

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)
```

```
# let's check the population created...
#population
```

Short version with 'Hall of Fame' (HoF)

Let's consider the additional feature of the built-in *algorithms.eaSimple* method - the hall of fame (HoF). It is implemented as *HallOfFame* class that can be used to retain the best individuals that ever existed in the population during the evolution, even if they have been lost at some point due to selection, crossover, or mutation. HoF is continuously sorted so that the first element is **the first individual** that had **the best fitness value** ever seen.

```
# define the hall-of-fame object:
HALL_OF_FAME_SIZE = 1
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
```

```
# let's check the initial state of HoF
hof.items
```

```
[]
```

Start workflow

```
# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", np.min)
stats.register("avg", np.mean)

# perform the Genetic Algorithm flow with hof feature added:
population, logbook = algorithms.eaSimple(population, toolbox, cxpb=P_CROSSOVER, m
                                          ngen=MAX_GENERATIONS, stats=stats, h

# print Hall of Fame info:
best = hof.items[0]
print("Hall of Fame Individuals = ", best, sep="\n")
print("Best Ever Individual = ", best.fitness.values[0])

# Genetic Algorithm is done - extract statistics:
minFitnessValues, meanFitnessValues = logbook.select("min", "avg")
```

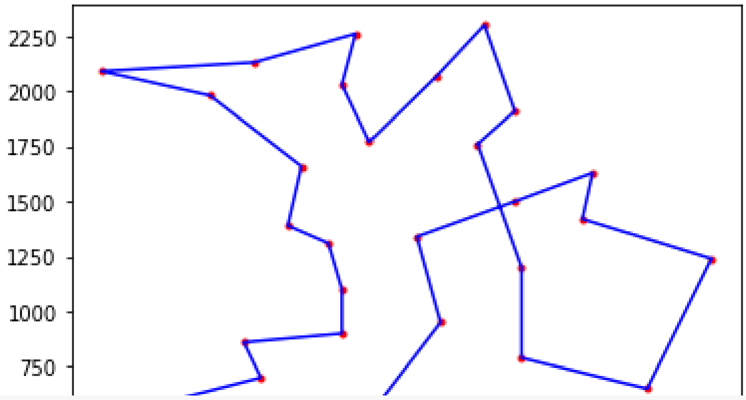
```
gen     nevals  min     avg
0       300     21103.3 26457
1       279     19562.4 25128.9
2       275     19456.5 24267.3
3       279     19760.9 23592
4       278     19406.4 22963.5
5       276     19105.2 22480.7
6       281     17802.4 22129.9
7       279     18160.2 21581.5
```

```
8       274     17691.3 21253.6
9       277     16011.9 20877.8
10      268     16011.9 20597.9
11      279     15878.8 20413.2
12      269     14589.1 20188.2
13      272     14589.1 19987.8
14      281     15182.9 19910.4
15      276     15836.2 19437
16      276     15687.4 19117.2
17      282     15426.5 19039.4
18      282     14905.2 18696.7
19      269     15020.4 18614.9
20      275     13346.3 18424
21      276     14711.4 18330.1
22      275     13139.1 18312
23      274     13139.1 18005
24      278     13002.5 17725.6
25      262     12203.5 17543.2
26      274     13157.3 17285.2
27      260     12918.6 16946.3
28      271     12918.6 16646
29      262     12918.6 16441
30      279     12652.9 16194.2
31      265     12652.9 15910.6
32      278     12565.6 15942.4
33      278     12520.5 15682
34      283     12436.1 15569.5
35      272     12214   15441.9
36      265     12278.3 15353.6
37      271     12302.9 15371.8
38      273     12037.5 15182
39      271     11969.4 14974.3
40      267     12081.2 14825
41      266     11785.8 14612.3
42      282     11477   14591.8
43      269     10869.8 14446
44      270     11475   14452.8
45      277     11796.8 14491.9
46      277     11262.8 14324.7
47      277     11333.1 14183.2
48      281     11311   14135.4
49      271     10921.8 14212.2
50      274     11172.8 14137.9
51      279     11295.1 13945
52      280     10870.6 14044.3
53      278     11225.2 14167.9
54      270     11336.3 13993.6
```

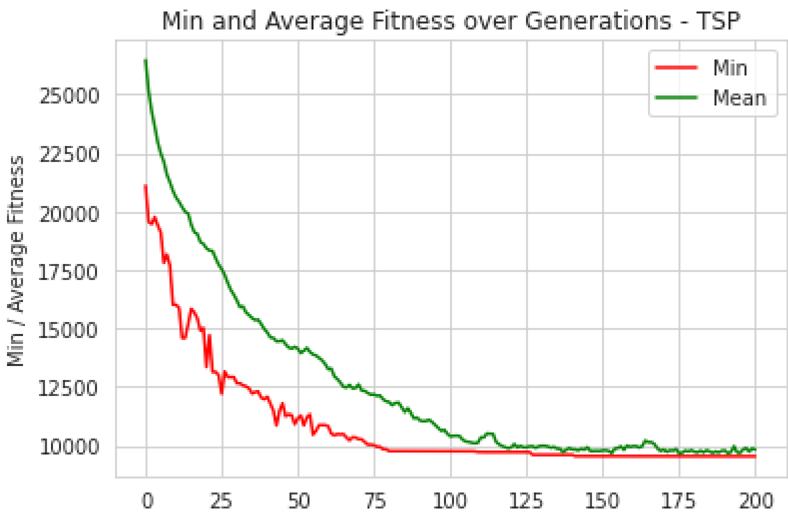
Plot results

```
# plot best solution:
plt.figure(1)
tsp.plotData(best)
```

```
<module 'matplotlib.pyplot' from '/usr/local/lib/python3.6/dist-packages/matplotlib/pyplot.py'>
```




```
# Plot statistics:
plt.figure(2)
sns.set_style("whitegrid")
plt.plot(minFitnessValues, color='red', label='Min')
plt.plot(meanFitnessValues, color='green', label='Mean')
plt.xlabel('Generation')
plt.ylabel('Min / Average Fitness')
plt.title('Min and Average Fitness over Generations - TSP')
plt.legend()
plt.show()
```



Lab 3 - Applications of EA for ML

based on (C) Eyal Wirsansky work

Brief Content:

- DEAP installation (**every time after start of Colab VM!**),
- main modules: *creator* and *toolbox*,
- components needed for the GA workflow,
- *regression problem*,
- *classification problem*.

By the end of this lab you will know:

- again, how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the *regression* and *classification* problems using a GA coded with the DEAP framework,
- how to experiment with various settings of the GA and interpret the differences in the results.

Installation and import of libraries

IMPORTANT: Mount your Google Drive!

At left sidebar -> click "Files" icon, the click "Mount Drive" icon with Google Drive logo, follow instructions.

```
# Copy all lab-related materials from Google Drive to your current location at Goo
! cp -r /content/drive/MyDrive/COLAB_EVO/EVO_Lecture03/* .
```

```
# Check the folders/files copied
! ls
```

```
01-solve-friedman.py  elitism.py                       __pycache__
02-solve-zoo.py       EVO_Lecture03_ML_examples.ipynb  sample_data
drive                 friedman.py                      zoo.py
```

In these and other lectures, we will use various Python packages:

- NumPy
- Matplotlib
- Seaborn

They are already pre-installed in Colab. Let's import them by the following code.

```
# Import all necessary standard libraries

import numpy as np
from pandas import read_csv
import random

# for Friedman1 problem data generation
from sklearn import datasets

# for ML training/testing
from sklearn import model_selection

# for Regression example
from sklearn.ensemble import GradientBoostingRegressor

# for Classification example
from sklearn.tree import DecisionTreeClassifier

# for MSE metrics
from sklearn.metrics import mean_squared_error

# for plotting
import matplotlib.pyplot as plt
import seaborn as sns
```

Example: Regression

Classic Solution

```
class Friedman1Test:
    """This class encapsulates the Friedman1 regression test for feature selection
    """

    VALIDATION_SIZE = 0.20
    NOISE = 1.0

    def __init__(self, numFeatures, numSamples, randomSeed):
        """
        :param numFeatures: total number of features to be used (at least 5)
        :param numSamples: number of samples in dataset
        :param randomSeed: random seed value used for reproducible results
        """

        self.numFeatures = numFeatures
        self.numSamples = numSamples
        self.randomSeed = randomSeed
```

```
        # generate test data:
        self.X, self.y = datasets.make_friedman1(n_samples=self.numSamples, n_feat
                                                  noise=self.NOISE, random_state=se

        # divide the data to a training set and a validation set:
        self.X_train, self.X_validation, self.y_train, self.y_validation = \
            model_selection.train_test_split(self.X, self.y, test_size=self.VALIDA

        self.regressor = GradientBoostingRegressor(random_state=self.randomSeed)

    def __len__(self):
        """
        :return: the total number of features
        """
        return self.numFeatures


    def getMSE(self, zeroOneList):
        """
        returns the mean squared error of the regressor, calculated for the valida
        using the features selected by the zeroOneList
        :param zeroOneList: a list of binary values corresponding the features in
        represents selecting the corresponding feature, while a value of '0' means
        :return: the mean squared error of the regressor when using the features s
        """

        # drop the columns of the training and validation sets that correspond to
        # unselected features:
        zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
        currentX_train = np.delete(self.X_train, zeroIndices, 1)
        currentX_validation = np.delete(self.X_validation, zeroIndices, 1)

        # train the regression model using th etraining set:
        self.regressor.fit(currentX_train, self.y_train)

        # calculate the regressor's output for the validation set:
        prediction = self.regressor.predict(currentX_validation)

        # return the mean square error of predicition vs actual data:
        return mean_squared_error(self.y_validation, prediction)
```

```
%time
# create a test instance of Friedman1Test class:
test = Friedman1Test(numFeatures=15, numSamples=60, randomSeed=42)
test
```

```
CPU times: user 4 µs, sys: 0 ns, total: 4 µs
Wall time: 22.2 µs
<__main__.Friedman1Test at 0x7f2f2bc47518>
```

```
test
```

```
<__main__.Friedman1Test at 0x7f2f2bc47518>
```

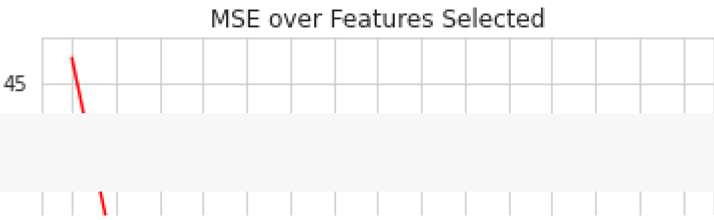
```
scores = []
# calculate MSE for 'n' first features:
for n in range(1, len(test) + 1):
  nFirstFeatures = [1] * n + [0] * (len(test) - n)
  score = test.getMSE(nFirstFeatures)
  print("%d first features: score (MSE) = %f" % (n, score))
  scores.append(score)
```

```
1 first features: score (MSE) = 47.553993
2 first features: score (MSE) = 26.121143
3 first features: score (MSE) = 18.509415
4 first features: score (MSE) = 7.322589
5 first features: score (MSE) = 6.702669
6 first features: score (MSE) = 7.677197
7 first features: score (MSE) = 11.614536
8 first features: score (MSE) = 11.294010
9 first features: score (MSE) = 10.858028
10 first features: score (MSE) = 11.602919
11 first features: score (MSE) = 15.017591
12 first features: score (MSE) = 14.258221
13 first features: score (MSE) = 15.274851
14 first features: score (MSE) = 15.726690
15 first features: score (MSE) = 17.187479
```

```
# Find the MINIMAL MSE and the correspondent number of features
index_min = np.argmin(scores)
print("MIN score (MSE) = %f, for %d first features" % (min(scores),index_min+1))
```

```
MIN score (MSE) = 6.702669, for 5 first features
```

```
# plot graph:
sns.set_style("whitegrid")
plt.plot([i + 1 for i in range(len(test))], scores, color='red')
plt.xticks(np.arange(1, len(test) + 1, 1.0))
plt.xlabel('n First Features')
plt.ylabel('MSE')
plt.title('MSE over Features Selected')
plt.show()
```



GA Solution

Install DEAP by *pip* with the following code:

```
# Install DEAP
!pip install deap
```

```
Requirement already satisfied: deap in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-package
```

```
from deap import base
from deap import creator
from deap import tools

import elitism
```

```
NUM_OF_FEATURES = 15
NUM_OF_SAMPLES = 60

# Genetic Algorithm constants:
POPULATION_SIZE = 30
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.2   # probability for mutating an individual
MAX_GENERATIONS = 30
HALL_OF_FAME_SIZE = 5

# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

```
# create the Friedman-1 test class:
friedman = Friedman1Test(NUM_OF_FEATURES, NUM_OF_SAMPLES, RANDOM_SEED)

toolbox = base.Toolbox()

# define a single objective, minimizing fitness strategy:
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))

# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMin)

# create an operator that randomly returns 0 or 1:
```

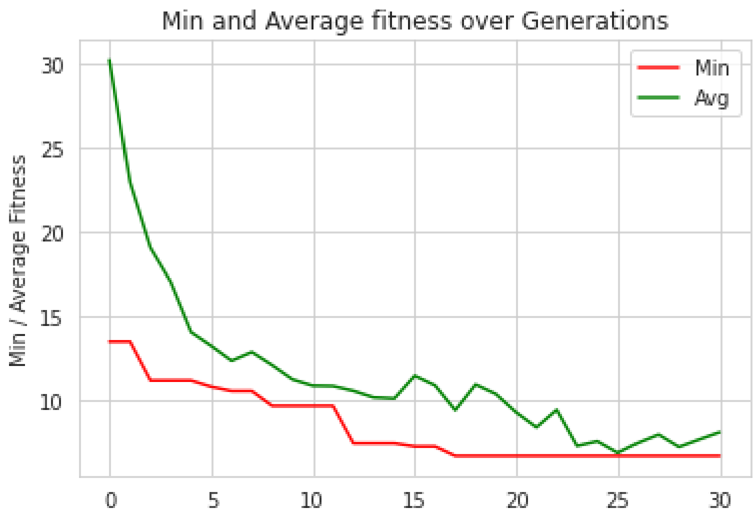
```
toolbox.register("zeroOrOne", random.randint, 0, 1)

# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator", tools.initRepeat, creator.Individual, toolbo

# create the population operator to generate a list of individuals:
toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCr


# fitness calculation
def friedmanTestScore(individual):
    return friedman.getMSE(individual),  # return a tuple


toolbox.register("evaluate", friedmanTestScore)

# genetic operators for binary list:
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/len(friedman))
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", np.min)
stats.register("avg", np.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

# perform the Genetic Algorithm flow with hof feature added:
population, logbook = elitism.eaSimpleWithElitism(population, toolbox, cxpb=P_CROS
                                                  ngen=MAX_GENERATIONS, stats=

# print best solution found:
best = hof.items[0]
print("Best Ever Individual = ", best)
print("Best Ever Fitness = ", best.fitness.values[0])
```

```
gen     nevals  min     avg
0       30      13.4946 30.2123
1       20      13.4946 22.9373
2       22      11.1869 19.0558
3       23      11.1869 17.0366
4       24      11.1869 14.0624
5       21      10.8123 13.2361
6       20      10.5603 12.3547
7       21      10.5603 12.8725
8       21      9.67682 12.0955
9       23      9.67682 11.2473
10      20      9.67682 10.8749
11      16      9.67682 10.8527
12      22      7.45319 10.5706
13      22      7.45319 10.1786
```

```
14      22      7.45319 10.1176
15      22      7.26529 11.4639
16      23      7.26529 10.8954
17      23      6.70267 9.41857
18      24      6.70267 10.9443
19      23      6.70267 10.3826
20      24      6.70267 9.28936
21      23      6.70267 8.39784
22      25      6.70267 9.43918
23      24      6.70267 7.29398
24      19      6.70267 7.57122
25      19      6.70267 6.88776
26      24      6.70267 7.48048
27      24      6.70267 7.96912
28      19      6.70267 7.23222
29      22      6.70267 7.67753
30      24      6.70267 8.10665
Best Ever Individual =  [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Best Ever Fitness =  6.702668910463276
```

```
# extract statistics:
minFitnessValues, meanFitnessValues = logbook.select("min", "avg")

# plot statistics:
sns.set_style("whitegrid")
plt.plot(minFitnessValues, label='Min', color='red')
plt.plot(meanFitnessValues, label='Avg', color='green')
plt.xlabel('Generation')
plt.ylabel('Min / Average Fitness')
plt.title('Min and Average fitness over Generations')
plt.legend()
plt.show()
```



Example: Classification

```
class Zoo:
    """This class encapsulates the Friedman1 test for a regressor
```

```
    """

    DATASET_URL = 'https://archive.ics.uci.edu/ml/machine-learning-databases/zoo/z
    NUM_FOLDS = 5

    def __init__(self, randomSeed):
        """
        :param randomSeed: random seed value used for reproducible results
        """
        self.randomSeed = randomSeed

        # read the dataset, skipping the first columns (animal name):
        self.data = read_csv(self.DATASET_URL, header=None, usecols=range(1, 18))

        # separate to input features and resulting category (last column):
        self.X = self.data.iloc[:, 0:16]
        self.y = self.data.iloc[:, 16]

        # split the data, creating a group of training/validation sets to be used
        self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS, random_state=s

        self.classifier = DecisionTreeClassifier(random_state=self.randomSeed)

    def __len__(self):
        """
        :return: the total number of features used in this classification problem
        """
        return self.X.shape[1]

    def getMeanAccuracy(self, zeroOneList):
        """
        returns the mean accuracy measure of the calssifier, calculated using k-fo
        using the features selected by the zeroOneList
        :param zeroOneList: a list of binary values corresponding the features in
        represents selecting the corresponding feature, while a value of '0' means
        :return: the mean accuracy measure of the calssifier when using the featur
        """

        # drop the dataset columns that correspond to the unselected features:
        zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
        currentX = self.X.drop(self.X.columns[zeroIndices], axis=1)

        # perform k-fold validation and determine the accuracy measure of the clas
        cv_results = model_selection.cross_val_score(self.classifier, currentX, se

        # return mean accuracy:
        return cv_results.mean()
```

Classic Solution

```
# create a zoo instance of Zoo class:
```

```
zoo = Zoo(randomSeed=42)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:296:
  FutureWarning
```

```
allOnes = [1] * len(zoo)
#print("For all features selected: ", allOnes, ", accuracy = ", round(zoo.getMeanA
print("For all features selected: ", allOnes, ", accuracy = ", zoo.getMeanAccuracy
```

```
For all features selected:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

GA Solution

```
from deap import base
from deap import creator
from deap import tools

import elitism
```

```
# Genetic Algorithm constants:
POPULATION_SIZE = 50
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.2   # probability for mutating an individual
MAX_GENERATIONS = 50
HALL_OF_FAME_SIZE = 5

FEATURE_PENALTY_FACTOR = 0.001
```

```
# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

```
# create the Zoo test class:
zoo = Zoo(RANDOM_SEED)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:296:
  FutureWarning
```

Genetic Algorithm - tools

```
toolbox = base.Toolbox()
```

```
# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

```
# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)

# create an operator that randomly returns 0 or 1:
toolbox.register("zeroOrOne", random.randint, 0, 1)

# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator", tools.initRepeat, creator.Individual, toolbo

# create the population operator to generate a list of individuals:
toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCr


# fitness calculation
def zooClassificationAccuracy(individual):
    numFeaturesUsed = sum(individual)
    if numFeaturesUsed == 0:
        return 0.0,
    else:
        accuracy = zoo.getMeanAccuracy(individual)
        return accuracy - FEATURE_PENALTY_FACTOR * numFeaturesUsed,  # return a tu


toolbox.register("evaluate", zooClassificationAccuracy)

# genetic operators:mutFlipBit

# Tournament selection with tournament size of 2:
toolbox.register("select", tools.selTournament, tournsize=2)

# Single-point crossover:
toolbox.register("mate", tools.cxTwoPoint)

# Flip-bit mutation:
# indpb: Independent probability for each attribute to be flipped
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/len(zoo))
```

```
/usr/local/lib/python3.6/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
/usr/local/lib/python3.6/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
```

Genetic Algorithm - workflow

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", np.max)
stats.register("avg", np.mean)
```

```
# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

# perform the Genetic Algorithm flow with hof feature added:
population, logbook = elitism.eaSimpleWithElitism(population, toolbox, cxpb=P_CROS
                                                  ngen=MAX_GENERATIONS, stats=
```

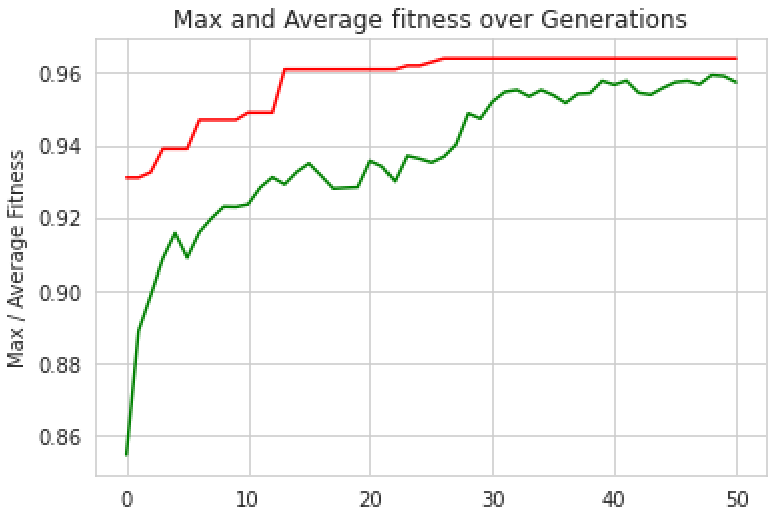
```
gen     nevals  max     avg
0       50      0.931   0.854475
1       41      0.931   0.888799
2       42      0.932476        0.898603
3       39      0.939           0.908676
4       38      0.939           0.915697
5       41      0.939           0.908824
6       43      0.947           0.915913
7       39      0.947           0.919742
8       36      0.947           0.922974
9       41      0.947           0.922894
10      44      0.949           0.923587
11      41      0.949           0.928286
12      42      0.949           0.931107
13      37      0.961           0.929126
14      44      0.961           0.932577
15      42      0.961           0.934937
16      38      0.961           0.931545
17      43      0.961           0.92797
18      44      0.961           0.928183
19      42      0.961           0.92835
20      40      0.961           0.935627
21      40      0.961           0.934047
22      38      0.961           0.929953
23      42      0.962           0.937037
24      39      0.962           0.936218
25      41      0.963           0.935186
26      40      0.964           0.936786
27      42      0.964           0.940153
28      34      0.964           0.948788
29      41      0.964           0.947346
30      44      0.964           0.952076
31      37      0.964           0.954729
32      43      0.964           0.955279
33      43      0.964           0.953498
34      42      0.964           0.955269
35      38      0.964           0.953825
36      42      0.964           0.951685
37      38      0.964           0.954197
38      39      0.964           0.954385
39      41      0.964           0.95778
40      41      0.964           0.95672
41      44      0.964           0.957848
42      43      0.964           0.954497
43      42      0.964           0.953987
44      39      0.964           0.955858
45      40      0.964           0.957407
46      43      0.964           0.957788
47      44      0.964           0.956839
48      41      0.964           0.959409
```

```
49      41      0.964           0.95912
50      44      0.964           0.957408
```

```
# print best solution found:
print("Best solutions are:")
for i in range(HALL_OF_FAME_SIZE):
  print(i, ": ", hof.items[i]," fitness =", round(hof.items[i].fitness.values[0],6
        " accuracy =", round(zoo.getMeanAccuracy(hof.items[i]),6)," features =", s

# extract statistics:
maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")
```

```
Best solutions are:
0 :  [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0]  fitness = 0.964  accur
1 :  [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1]  fitness = 0.963  accur
2 :  [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0]  fitness = 0.963  accur
3 :  [1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0]  fitness = 0.963  accur
4 :  [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0]  fitness = 0.963  accur
```

```
# plot statistics:
sns.set_style("whitegrid")
plt.plot(maxFitnessValues, color='red')
plt.plot(meanFitnessValues, color='green')
plt.xlabel('Generation')
plt.ylabel('Max / Average Fitness')
plt.title('Max and Average fitness over Generations')
plt.show()
```




```
population
```

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

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

Lab 4 - Applications of EA for ML hyperparameter tuning

based on (C) Eyal Wirsansky work

Brief Content:

- DEAP installation (**every time after start of Colab VM!**),
- components needed for the GA workflow,
- *classification problem*
  - classic solution
  - sklearn-based solution,
  - DEAP-based solution with finetuning.

By the end of this lab you will know:

- again, how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the *classification* problem using a classic and GA-based solutions with the SKLEARN and DEAP frameworks,
- how to experiment with various settings of the GA and interpret the differences in the results.

Installation and import of libraries

```
! pip install deap
```

```
Requirement already satisfied: deap in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-package
```

```
! pip install sklearn-deap
```

```
Requirement already satisfied: sklearn-deap in /usr/local/lib/python3.6/dist-
Requirement already satisfied: deap>=1.0.2 in /usr/local/lib/python3.6/dist-p
Requirement already satisfied: scipy>=0.16.0 in /usr/local/lib/python3.6/dist
Requirement already satisfied: scikit-learn>=0.18.0 in /usr/local/lib/python3
Requirement already satisfied: numpy>=1.9.3 in /usr/local/lib/python3.6/dist-
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/dist-
```

… e various Python packages:

- [Matplotlib]()

- [Seaborn]()

They are already pre-installed in Colab. Let's import them by the following code.

```
# Import all necessary standard libraries

import numpy as np
from pandas import read_csv
import random

# for Friedman1 problem data generation
from sklearn import datasets

# for ML training/testing
from sklearn import model_selection

# for Regression example
from sklearn.ensemble import GradientBoostingRegressor

# for Classification example
from sklearn.tree import DecisionTreeClassifier

# for MSE metrics
from sklearn.metrics import mean_squared_error

# for plotting
import matplotlib.pyplot as plt
import seaborn as sns
```

Example - Dataset Description

[UCI Wine Data Set]()

- These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from 3 different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines.
- The attributes are (dontated by Riccardo Leardi, [riclea@anchem.unige.it]()):
    1. Alcohol
    2. Malic acid
    3. Ash
    4. Alcalinity of ash
    5. Magnesium
    6. Total phenols
    7. Flavanoids

Saved successfully!

    9. Proanthocyanins

10. Color intensity
11. Hue
12. OD280/OD315 of diluted wines
13. Proline

- Number of Instances:
    - class 1: 59
    - class 2: 71
    - class 3: 48
- Number of Attributes: 13

```
import numpy as np
import time
import random

from sklearn import model_selection
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import GridSearchCV

from pandas import read_csv
from evolutionary_search import EvolutionaryAlgorithmSearchCV

accuracy_classic_solution = 0
accuracy_sklearn_deap_solution = 0
accuracy_DEAP_solution = 0

class HyperparameterTuningGrid:

    NUM_FOLDS = 5

    def __init__(self, randomSeed):

        self.randomSeed = randomSeed
        self.initWineDataset()
        self.initClassifier()
        self.initKfold()
        self.initGridParams()

    def initWineDataset(self):
        url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine

        self.data = read_csv(url, header=None, usecols=range(0, 14))
        self.X = self.data.iloc[:, 1:14]
        self.y = self.data.iloc[:, 0]

    def initClassifier(self):
        self.classifier = AdaBoostClassifier(random_state=self.randomSeed)

        self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS,
                                           random_state=self.randomSeed)
```

```
    def initGridParams(self):
        self.gridParams = {
            'n_estimators': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
            'learning_rate': np.logspace(-2, 0, num=10, base=10),
            'algorithm': ['SAMME', 'SAMME.R'],
        }

    def getDefaultAccuracy(self):
        cv_results = model_selection.cross_val_score(self.classifier,
                                                     self.X,
                                                     self.y,
                                                     cv=self.kfold,
                                                     scoring='accuracy')
        return cv_results.mean()

    def gridTest(self):
        print("Classic grid search is STARTED ...")

        gridSearch = GridSearchCV(estimator=self.classifier,
                                  param_grid=self.gridParams,
                                  cv=self.kfold,
                                  scoring='accuracy',
                                  iid='False',
                                  n_jobs=4)

        gridSearch.fit(self.X, self.y)
        print("Best parameters: ", gridSearch.best_params_)
        print("Score (after gridSearch): ", gridSearch.best_score_)
        accuracy_classic_solution = gridSearch.best_score_
        print("Classic grid search is FINISHED.")
        return accuracy_classic_solution

    def geneticGridTest(self):
        print("Genetic grid search was STARTED ...")

        gridSearch = EvolutionaryAlgorithmSearchCV(estimator=self.classifier,
                                                   params=self.gridParams,
                                                   cv=self.kfold,
                                                   scoring='accuracy',
                                                   #verbose=True,
                                                   iid='False',
                                                   n_jobs=4,
                                                   verbose=1,
                                                   population_size=20,
                                                   gene_mutation_prob=0.30,
                                                   #gene_crossover_prob=0.5,
                                                   tournament_size=2,
                                                   generations_number=5)

        gridSearch.fit(self.X, self.y)
        [illegible]ore_)
        [illegible]rams_)
        print(gridSearch.all_logbooks_)
```

```
        print("Genetic grid search is FINISHED.")
        return gridSearch.best_score_, gridSearch.best_params_, gridSearch.all_log
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:144: Futu
  warnings.warn(message, FutureWarning)
```

```
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# create a problem instance:
test = HyperparameterTuningGrid(RANDOM_SEED)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:296:
  FutureWarning
```

Classic Solutions

DEMO 1. Default Hyperparameter Values

```
print('*************************')
start = time.time()
print("Default Classifier Hyperparameter values:")
print(test.classifier.get_params())
print("Score (with default values) = ", test.getDefaultAccuracy())
end = time.time()
print("Time Elapsed = ", end - start)
```

```
*************************
Default Classifier Hyperparameter values:
{'algorithm': 'SAMME.R', 'base_estimator': None, 'learning_rate': 1.0, 'n_est
Score (with default values) =  0.6457142857142857
Time Elapsed =  0.4167492389678955
```

DEMO 2. Extensive Grid Search

```
print('*************************')
start = time.time()
accuracy_classic_solution = test.gridTest()
end = time.time()
print("Time Elapsed = ", end - start)
```



```
TED ...
Best parameters:  {'algorithm': 'SAMME.R', 'learning_rate': 0.35938136638046
Score (after gridSearch):  0.9325842696629213
```

```
Classic grid search is FINISHED.
Time Elapsed =  74.51628732681274
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_search.py:823
  "removed in 0.24.", FutureWarning
```

GA Solutions

DEMO 3. GA-driven Grid Search

based on sklearn-deap

```
print('***************************')
start = time.time()
best_score_, best_params_, logbook_GA_sklearn = test.geneticGridTest()
print(best_score_, best_params_)
end = time.time()
print("Time Elapsed = ", end - start)

# extract statistics:
maxFitnessValues_GA_sklearn, meanFitnessValues_GA_sklearn = logbook_GA_sklearn[0].
```

```
***************************
Genetic grid search was STARTED ...
Types [1, 2, 1] and maxint [9, 9, 1] detected
--- Evolve in 200 possible combinations ---
gen     nevals  avg             min             max             std
0       20      0.708427        0.117978        0.910112        0.265992
1       13      0.865169        0.662921        0.926966        0.0717915
2       15      0.887921        0.646067        0.926966        0.0571676
3       12      0.896348        0.679775        0.926966        0.0526256
4       16      0.918539        0.88764         0.926966        0.0110233
5       9       0.911517        0.730337        0.926966        0.0425958
Best individual is: {'n_estimators': 60, 'learning_rate': 0.5994842503189409,
with fitness: 0.9269662921348315
0.9269662921348315
{'n_estimators': 60, 'learning_rate': 0.5994842503189409, 'algorithm': 'SAMME
[[{'gen': 0, 'nevals': 20, 'avg': 0.7084269662921348, 'min': 0.11797752808988
Genetic grid search is FINISHED.
0.9269662921348315 {'n_estimators': 60, 'learning_rate': 0.5994842503189409,
Time Elapsed =  24.287983655929565
```

DEMO 4. Direct GA

based on DEAP

Saved successfully!

```
from deap import creator
from deap import tools
```

```
from deap import algorithms

import random
import numpy

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

Genetic Tools

```
from sklearn import model_selection
from sklearn.ensemble import AdaBoostClassifier

from pandas import read_csv


class HyperparameterTuningGenetic:

    NUM_FOLDS = 5

    def __init__(self, randomSeed):

        self.randomSeed = randomSeed
        self.initWineDataset()
        self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS, random_state=s

    def initWineDataset(self):
        url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine

        self.data = read_csv(url, header=None, usecols=range(0, 14))
        self.X = self.data.iloc[:, 1:14]
        self.y = self.data.iloc[:, 0]

    # ADABoost [n_estimators, learning_rate, algorithm]:
    # "n_estimators": integer
    # "learning_rate": float
    # "algorithm": {'SAMME', 'SAMME.R'}
    def convertParams(self, params):
        n_estimators = round(params[0])  # round to nearest integer
        learning_rate = params[1]        # no conversion needed
        algorithm = ['SAMME', 'SAMME.R'][round(params[2])]  # round to 0 or 1, the
        return n_estimators, learning_rate, algorithm

    def getAccuracy(self, params):
        n_estimators, learning_rate, algorithm = self.convertParams(params)
        self.classifier = AdaBoostClassifier(random_state=self.randomSeed,
                                             n_estimators=n_estimators,
                                             learning_rate=learning_rate,
                                             algorithm=algorithm
                                             )

        cv_results = model_selection.cross_val_score(self.classifier,
```

```
                                                             self.X,
                                                             self.y,
                                                             cv=self.kfold,
                                                             scoring='accuracy')
        return cv_results.mean()

    def formatParams(self, params):
        return "'n_estimators'=%3d, 'learning_rate'=%1.3f, 'algorithm'=%s" % (self
```

Elitism Tools

```
# boundaries for ADABOOST parameters:
# "n_estimators": 1..100
# "learning_rate": 0.01..100
# "algorithm": 0, 1
# [n_estimators, learning_rate, algorithm]:
BOUNDS_LOW =  [  1, 0.01, 0]
BOUNDS_HIGH = [100, 1.00, 1]

NUM_OF_PARAMS = len(BOUNDS_HIGH)

# Genetic Algorithm constants:
POPULATION_SIZE = 20
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.5   # probability for mutating an individual
MAX_GENERATIONS = 5
HALL_OF_FAME_SIZE = 5
CROWDING_FACTOR = 20.0  # crowding factor for crossover and mutation

# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

```
def eaSimpleWithElitism(population, toolbox, cxpb, mutpb, ngen, stats=None,
             halloffame=None, verbose=__debug__):
    """This algorithm is similar to DEAP eaSimple() algorithm, with the modificati
    halloffame is used to implement an elitism mechanism. The individuals containe
    halloffame are directly injected into the next generation and are not subject
    genetic operators of selection, crossover and mutation.
    """
    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in population if not ind.fitness.valid]
                                       box.evaluate, invalid_ind)
                                       ind, fitnesses):
```

Saved successfully!

```
    if halloffame is None:
        raise ValueError("halloffame parameter must not be empty!")

    halloffame.update(population)
    hof_size = len(halloffame.items) if halloffame.items else 0

    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=len(invalid_ind), **record)
    if verbose:
        print(logbook.stream)

    # Begin the generational process
    for gen in range(1, ngen + 1):

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population) - hof_size)

        # Vary the pool of individuals
        offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # add the best back to population:
        offspring.extend(halloffame.items)

        # Update the hall of fame with the generated individuals
        halloffame.update(offspring)

        # Replace the current population by the offspring
        population[:] = offspring

        # Append the current generation statistics to the logbook
        record = stats.compile(population) if stats else {}
        logbook.record(gen=gen, nevals=len(invalid_ind), **record)
        if verbose:
            print(logbook.stream)

    return population, logbook
```

```
# create the classifier accuracy test class:
test = HyperparameterTuningGenetic(RANDOM_SEED)

toolbox = base.Toolbox()

# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))

sed on list:
t, fitness=creator.FitnessMax)

# define the hyperparameter attributes individually:
```

```
for i in range(NUM_OF_PARAMS):
    # "hyperparameter_0", "hyperparameter_1", ...
    toolbox.register("hyperparameter_" + str(i),
                     random.uniform,
                     BOUNDS_LOW[i],
                     BOUNDS_HIGH[i])

# create a tuple containing an attribute generator for each param searched:
hyperparameters = ()
for i in range(NUM_OF_PARAMS):
    hyperparameters = hyperparameters + \
                      (toolbox.__getattribute__("hyperparameter_" + str(i)),)

# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator",
                 tools.initCycle,
                 creator.Individual,
                 hyperparameters,
                 n=1)

# create the population operator to generate a list of individuals:
toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCr

# fitness calculation
def classificationAccuracy(individual):
    return test.getAccuracy(individual),

toolbox.register("evaluate", classificationAccuracy)

# genetic operators:mutFlipBit

# genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0 / NUM_OF_PARAMS)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/model_selection/_split.py:296:
  FutureWarning
/usr/local/lib/python3.6/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
/usr/local/lib/python3.6/dist-packages/deap/creator.py:141: RuntimeWarning: A
```

Saved successfully!

```
def eaSimpleWithElitism(population, toolbox, cxpb, mutpb, ngen, stats=None,
```

```
             halloffame=None, verbose=__debug__):
    """This algorithm is similar to DEAP eaSimple() algorithm, with the modificati
    halloffame is used to implement an elitism mechanism. The individuals containe
    halloffame are directly injected into the next generation and are not subject
    genetic operators of selection, crossover and mutation.
    """
    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in population if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    if halloffame is None:
        raise ValueError("halloffame parameter must not be empty!")

    halloffame.update(population)
    hof_size = len(halloffame.items) if halloffame.items else 0

    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=len(invalid_ind), **record)
    if verbose:
        print(logbook.stream)

    # Begin the generational process
    for gen in range(1, ngen + 1):

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population) - hof_size)

        # Vary the pool of individuals
        offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # add the best back to population:
        offspring.extend(halloffame.items)

        # Update the hall of fame with the generated individuals
        halloffame.update(offspring)

        # Replace the current population by the offspring
        population[:] = offspring

        # Append the current generation statistics to the logbook
Saved successfully!                      opulation) if stats else {}
                                         nevals=len(invalid_ind), **record)
        if verbose:
            print(logbook.stream)
```

```
    return population, logbook
```

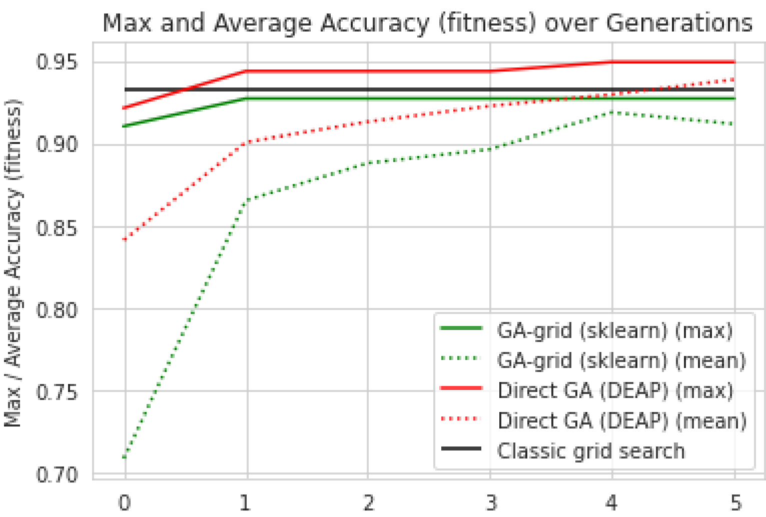
GA workflow

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
print("Time Elapsed = ", end - start)

# print best solution found:
print("- Best solution is: ")
print("params = ", test.formatParams(hof.items[0]))
print("Accuracy = %1.5f" % hof.items[0].fitness.values[0])

# extract statistics:
maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")
```

```
***************************
gen	nevals	max	avg
0	20	0.92127	0.841024
1	14	0.943651	0.900603
2	13	0.943651	0.912841
3	14	0.943651	0.922476
4	15	0.949206	0.929468
5	13	0.949206	0.938563
Time Elapsed =  46.62226867675781
- Best solution is:
params =  'n_estimators'= 69, 'learning_rate'=0.628, 'algorithm'=SAMME.R
Accuracy = 0.94921
```

Comparison Plot

```
sns.set_style("whitegrid")

# Classic grid search solution
plt.hlines(accuracy_classic_solution, 0, 5, linestyle = 'solid', label='Classic gr

# GA_deap_sklearn solution
plt.plot(maxFitnessValues_GA_sklearn, color='green', label='GA-grid (sklearn) (max
plt.plot(meanFitnessValues_GA_sklearn, color='green', linestyle = 'dotted', label=

# GA_DEAP solution
plt.plot(maxFitnessValues, color='red', label='Direct GA (DEAP) (max)')
plt.plot(meanFitnessValues, color='red', linestyle = 'dotted', label='Direct GA (D

plt.xlabel('Generation')
plt.ylabel('Max / Average Accuracy (fitness)')
plt.title('Max and Average Accuracy (fitness) over Generations')
plt.legend()
plt.show()
```



Saved successfully!

Lab 5 - Applications of EA (actually GA here) for DL tuning (architecture + hyperparameters)

based on (C) Varoquaux, Grobler, Wirsansky work

Brief Content:

- DEAP installation (**every time after start of Colab VM!**),
- components needed for the GA workflow,
- *classification problem*
  - Neural Network (NN) Architecture Tuning
  - NN Hyperparameter Tuning,
  - NN Architecture + NN Hyperparameter Tuning
- performance comparison (accuracy and run time).

By the end of this lab you will know:

- again, how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the *classification* problem using a GA-based solutions for NN architecture tuning, NN hyperparameter tuning, and their combination,
- how to experiment with various settings of the GA and interpret the differences in the results.

Installation and import of libraries

```
! pip install deap
```

```
Requirement already satisfied: deap in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-package
```

In these and other lectures, we will use various Python packages:

- NumPy
- Matplotlib
- Seaborn

They are already pre-installed in Colab. Let's import them by the following code.

```
# Import all necessary standard libraries
```

```
import numpy as np
from pandas import read_csv
import random

# for Friedman1 problem data generation
from sklearn import datasets

# for ML training/testing
from sklearn import model_selection

# for Regression example
from sklearn.ensemble import GradientBoostingRegressor

# for Classification example
from sklearn.tree import DecisionTreeClassifier

# for MSE metrics
from sklearn.metrics import mean_squared_error

# for plotting
import matplotlib.pyplot as plt
import seaborn as sns


from sklearn import model_selection
from sklearn import datasets
from sklearn.neural_network import MLPClassifier

from sklearn.exceptions import ConvergenceWarning
from sklearn.utils.testing import ignore_warnings

import numpy as np
import time
import random
from pandas import read_csv

import random
import numpy
```

Part 1. Neural Network Architecture Tuning

Wine Classification Example

Wine Dataset

Description

[UCI Wine Data Set]()

- These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from 3 different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines.
- The attributes are (dontated by Riccardo Leardi, riclea@anchem.unige.it):
    1. Alcohol
    2. Malic acid
    3. Ash
    4. Alcalinity of ash
    5. Magnesium
    6. Total phenols
    7. Flavanoids
    8. Nonflavanoid phenols
    9. Proanthocyanins
    10. Color intensity
    11. Hue
    12. OD280/OD315 of diluted wines
    13. Proline
- Number of Instances:
    - class 1: 59
    - class 2: 71
    - class 3: 48
- Number of Attributes: 13

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

# import Wine dataset
wine_dataset = datasets.load_wine()
X = wine_dataset.data[:, :2]  # we only take the first two features.
y = wine_dataset.target
```

Features and Targets

```
list(wine_dataset.target_names)
```

```
['class_0', 'class_1', 'class_2']
```

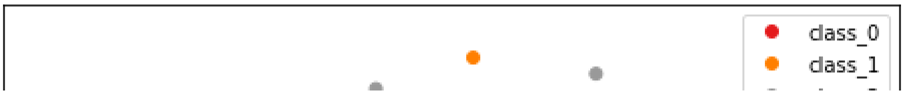
```
list(wine_dataset.feature_names)
```

```
['alcohol',
 'malic_acid',
```

```
     'ash',
     'alcalinity_of_ash',
     'magnesium',
     'total_phenols',
     'flavanoids',
     'nonflavanoid_phenols',
     'proanthocyanins',
     'color_intensity',
     'hue',
     'od280/od315_of_diluted_wines',
     'proline']
```

Plot Some Features

```
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

plt.figure(2, figsize=(8, 6))
plt.clf()

# Plot the training points
scatter = plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1)
            #edgecolor='k')
plt.xlabel(wine_dataset.feature_names[0])
plt.ylabel(wine_dataset.feature_names[1])

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
#plt.legend()
#plt.legend(*scatter.legend_elements())
classes = list(wine_dataset.target_names)
plt.legend(handles=scatter.legend_elements()[0], labels=classes)

plt.show()
```



Neural Network

```
from deap import base
from deap import creator
from deap import tools
from deap import algorithms

class MlpLayersTest:

    NUM_FOLDS = 5

    def __init__(self, randomSeed, network_name):

        self.randomSeed = randomSeed
        self.initDataset(network_name)
        self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS, random_state=s

    def initDataset(self, network_name):

        if network_name == 'iris':
          self.data = datasets.load_iris()
        elif network_name == 'wine':
          self.data = datasets.load_wine()
        elif network_name == 'breast_cancer':
          self.data = datasets.load_breast_cancer()
        else:
          self.data = []
          print('ERROR: Wrong dataset name was used!')

        self.X = self.data['data']
        self.y = self.data['target']


    # params contains: [layer_1_size, layer_2_size, layer_3_size, layer_4_size]
    def convertParams(self, params):

        # transform the layer sizes from float (possibly negative) values into hid
        if round(params[1]) <= 0:
            hiddenLayerSizes = round(params[0]),
        elif round(params[2]) <= 0:
            hiddenLayerSizes = (round(params[0]), round(params[1]))
        elif round(params[3]) <= 0:
            hiddenLayerSizes = (round(params[0]), round(params[1]), round(params[2
        else:
            hiddenLayerSizes = (round(params[0]), round(params[1]), round(params[2

        return hiddenLayerSizes

    @ignore_warnings(category=ConvergenceWarning)
```

```
    def getAccuracy(self, params):
        hiddenLayerSizes = self.convertParams(params)

        self.classifier = MLPClassifier(random_state=self.randomSeed,
                                        hidden_layer_sizes=hiddenLayerSizes)

        cv_results = model_selection.cross_val_score(self.classifier,
                                                     self.X,
                                                     self.y,
                                                     cv=self.kfold,
                                                     scoring='accuracy')

        return cv_results.mean()

    def formatParams(self, params):
        return "'hidden_layer_sizes'={}".format(self.convertParams(params))
```

GA Solution - max: 4 layers

```
# boundaries for layer size parameters:
# [layer_layer_1_size, hidden_layer_2_size, hidden_layer_3_size, hidden_layer_4_si
BOUNDS_LOW =  [ 5,  -5, -10, -20]
BOUNDS_HIGH = [15,  10,  10,  10]

NUM_OF_PARAMS = len(BOUNDS_HIGH)

# Genetic Algorithm constants:
POPULATION_SIZE = 20
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.5   # probability for mutating an individual
MAX_GENERATIONS = 10
HALL_OF_FAME_SIZE = 3
CROWDING_FACTOR = 10.0  # crowding factor for crossover and mutation
```

Genetic Tools

```
toolbox = base.Toolbox()

# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))

# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)

# define the layer_size_attributes individually:
for i in range(NUM_OF_PARAMS):
    # "layer_size_attribute_0", "layer_size_attribute_1", ...
    toolbox.register("layer_size_attribute_" + str(i),
                     random.uniform,
```

```
                    BOUNDS_LOW[i],
                    BOUNDS_HIGH[i])

# create a tuple containing an layer_size_attribute generator for each hidden laye
layer_size_attributes = ()
for i in range(NUM_OF_PARAMS):
    layer_size_attributes = layer_size_attributes + \
                            (toolbox.__getattribute__("layer_size_attribute_" + st

# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator",
                 tools.initCycle,
                 creator.Individual,
                 layer_size_attributes,
                 n=1)

# create the population operator to generate a list of individuals:
toolbox.register("populationCreator",
                 tools.initRepeat,
                 list,
                 toolbox.individualCreator)



# fitness calculation
def classificationAccuracy(individual):
    return test.getAccuracy(individual),



toolbox.register("evaluate", classificationAccuracy)

# genetic operators:mutFlipBit

# genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)
```

```
/usr/local/lib/python3.7/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
/usr/local/lib/python3.7/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
```

Elitism Tools

```
def eaSimpleWithElitism(population, toolbox, cxpb, mutpb, ngen, stats=None,
             halloffame=None, verbose=__debug__):
    """This algorithm is similar to DEAP eaSimple() algorithm, with the modificati
    halloffame is used to implement an elitism mechanism. The individuals containe
    halloffame are directly injected into the next generation and are not subject
    genetic operators of selection, crossover and mutation.
    """
    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in population if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    if halloffame is None:
        raise ValueError("halloffame parameter must not be empty!")

    halloffame.update(population)
    hof_size = len(halloffame.items) if halloffame.items else 0

    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=len(invalid_ind), **record)
    if verbose:
        print(logbook.stream)

    # Begin the generational process
    for gen in range(1, ngen + 1):

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population) - hof_size)

        # Vary the pool of individuals
        offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # add the best back to population:
        offspring.extend(halloffame.items)

        # Update the hall of fame with the generated individuals
        halloffame.update(offspring)

        # Replace the current population by the offspring
        population[:] = offspring

        # Append the current generation statistics to the logbook
        record = stats.compile(population) if stats else {}
```

```
        record = stats.compile(population) if stats else {}
        logbook.record(gen=gen, nevals=len(invalid_ind), **record)
        if verbose:
            print(logbook.stream)

    return population, logbook
```

GA Workflow

```
# create the classifier accuracy test class:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# dataset_name = 'iris'
# dataset_name = 'breast_cancer'
dataset_name = 'wine'

test = MlpLayersTest(RANDOM_SEED, dataset_name)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/model_selection/_split.py:296:
  FutureWarning
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                          toolbox,
                                          cxpb=P_CROSSOVER,
                                          mutpb=P_MUTATION,
                                          ngen=MAX_GENERATIONS,
                                          stats=stats,
                                          halloffame=hof,
                                          verbose=True)

end = time.time()
time_NNA = end - start
print("Time Elapsed = ", time_NNA)

# print best solution found:
print("Best solution is: ", test.formatParams(hof.items[0]))
```

```
print("Accuracy = %1.5f" % hof.items[0].fitness.values[0])

# extract statistics:
maxFitnessValues_NNA, meanFitnessValues_NNA = logbook.select("max", "avg")
```

```
***************************
gen     nevals  max             avg
0       20      0.769841        0.284063
1       17      0.769841        0.473413
2       15      0.769841        0.606905
3       16      0.769841        0.659238
4       17      0.769841        0.673444
5       14      0.769841        0.703746
6       17      0.769841        0.739619
7       14      0.769841        0.70954
8       16      0.769841        0.686921
9       17      0.769841        0.689833
10      15      0.769841        0.680286
Time Elapsed =  82.1906521320343
Best solution is:  'hidden_layer_sizes'=(13, 4, 7)
Accuracy = 0.76984
```

Problems?

... ... with various RANDOM_SEED ......

Results for various RANDOM_SEEDs

```
# dataset = 'wine'
# RANDOM_SEED = 42

gen nevals  max         avg
0   20      0.769841    0.284063
1   17      0.769841    0.473413
2   15      0.769841    0.606905
3   16      0.769841    0.659238
4   17      0.769841    0.673444
5   14      0.769841    0.703746
6   17      0.769841    0.739619
7   14      0.769841    0.70954
8   16      0.769841    0.686921
9   17      0.769841    0.689833
10  15      0.769841    0.680286
Best solution is:  'hidden_layer_sizes'=(13, 4, 7)
Accuracy =  0.76984
```

```
# dataset = 'wine'
# RANDOM_SEED = 666

***************************
```

```
gen nevals  max         avg
0   20      0.647937    0.31354
1   17      0.647937    0.41869
2   15      0.647937    0.478095
3   16      0.647937    0.418651
4   17      0.647937    0.503325
5   12      0.647937    0.492421
6   17      0.647937    0.435524
7   16      0.647937    0.503032
8   16      0.647937    0.466016
9   16      0.647937    0.51246
10  17      0.647937    0.572524
Time Elapsed =  93.69340062141418
Best solution is:  'hidden_layer_sizes'=(14, 3, 4, 4)
Accuracy = 0.64794
```

```
# dataset = 'wine'
# RANDOM_SEED = 1042
***************************
gen nevals  max         avg
0   20      0.520159    0.289151
1   15      0.520159    0.322095
2   12      0.520159    0.42246
3   17      0.541587    0.419079
4   14      0.541587    0.41527
5   17      0.541587    0.471929
6   15      0.541587    0.457198
7   16      0.541587    0.472865
8   17      0.541587    0.493143
9   16      0.541587    0.45669
10  13      0.541587    0.488968
Time Elapsed =  84.34443235397339
Best solution is:  'hidden_layer_sizes'=(9, 9, 5)
Accuracy = 0.54159
```

RESUME

For various RANDOM_SEED we can obtain NNs with the **very different**:

- **performance** (accuracy),
- **the number of nodes** in layers,
- **the number of layers**.

The reason is the stochastic (so-called **non-gradient**) manner of parameter change during evolution. There is a possibility that all models for different RANDOM_SEEDs can reach the different **local** (not **global**) the maximum value of fitness function (accuracy here).

... with various datasets ...

(let's try it as a self-guided learning!)

It takes a small change in *dataset_name* variable.

Iris dataset

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

# import dataset
data = datasets.load_iris()
X = data.data[:, :2]  # we only take the first two features.
y = data.target
```

Features and Targets

```
list(data.target_names)
```

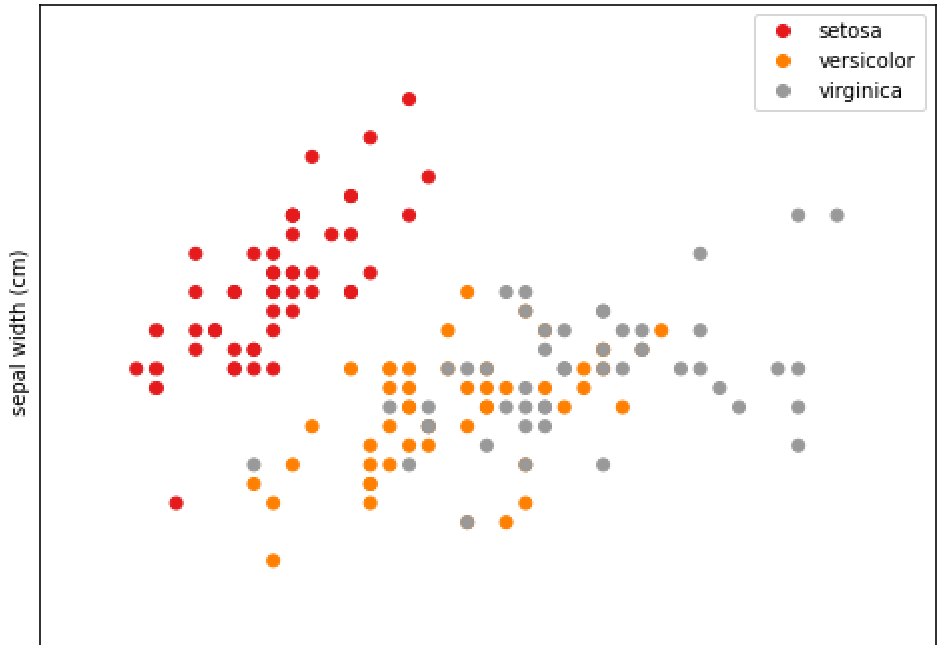
```
['setosa', 'versicolor', 'virginica']
```

```
list(data.feature_names)
```

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

Plot Some Features

```
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

plt.figure(2, figsize=(8, 6))
plt.clf()

# Plot the training points
scatter = plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1)
            #edgecolor='k')
plt.xlabel(data.feature_names[0])
plt.ylabel(data.feature_names[1])

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
#plt.legend()
#plt.legend(*scatter.legend_elements())
classes = list(data.target_names)
plt.legend(handles=scatter.legend_elements()[0], labels=classes)

plt.show()
```



Breast cancer dataset

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

# import dataset
data = datasets.load_breast_cancer()
X = data.data[:, :2]  # we only take the first two features.
y = data.target
```

Features and Targets

```
list(data.target_names)
```

```
['malignant', 'benign']
```

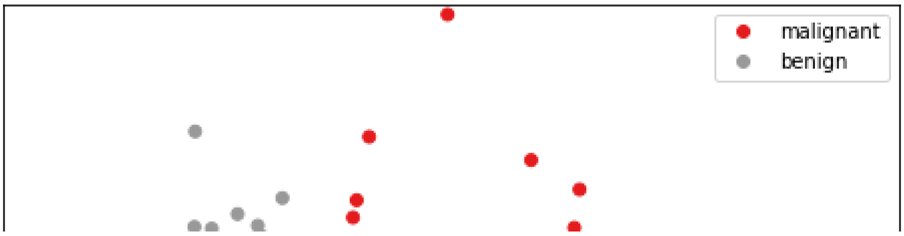
```
list(data.feature_names)
```

```
['mean radius',
 'mean texture',
 'mean perimeter',
 'mean area',
 'mean smoothness',
 'mean compactness',
 'mean concavity',
 'mean concave points',
 'mean symmetry',
 'mean fractal dimension',
 'radius error',
 'texture error',
 'perimeter error',
 'area error',
```

```
 'smoothness error',
 'compactness error',
 'concavity error',
 'concave points error',
 'symmetry error',
 'fractal dimension error',
 'worst radius',
 'worst texture',
 'worst perimeter',
 'worst area',
 'worst smoothness',
 'worst compactness',
 'worst concavity',
 'worst concave points',
 'worst symmetry',
 'worst fractal dimension']
```

Plot Some Features

```
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

plt.figure(2, figsize=(8, 6))
plt.clf()

# Plot the training points
scatter = plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1)
            #edgecolor='k')
plt.xlabel(data.feature_names[0])
plt.ylabel(data.feature_names[1])

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
#plt.legend()
#plt.legend(*scatter.legend_elements())
classes = list(data.target_names)
plt.legend(handles=scatter.legend_elements()[0], labels=classes)

plt.show()
```



Results for various datasets

```
### wine
# RANDOM_SEED = 42

gen nevals  max         avg
0   20      0.769841    0.284063
1   17      0.769841    0.473413
2   15      0.769841    0.606905
3   16      0.769841    0.659238
4   17      0.769841    0.673444
5   14      0.769841    0.703746
6   17      0.769841    0.739619
7   14      0.769841    0.70954
8   16      0.769841    0.686921
9   17      0.769841    0.689833
10  15      0.769841    0.680286
- Best solution is:  'hidden_layer_sizes'=(13, 4, 7) , accuracy =  0.7698412698412
```

```
### iris
# RANDOM_SEED = 42

gen nevals  max         avg
0   20      0.666667    0.416333
1   17      0.693333    0.487
2   15      0.76        0.537333
3   14      0.76        0.550667
4   17      0.76        0.568333
5   17      0.76        0.653667
6   14      0.76        0.589333
7   15      0.76        0.618
8   16      0.866667    0.616667
9   16      0.866667    0.666333
10  16      0.866667    0.722667
- Best solution is:  'hidden_layer_sizes'=(15, 5, 8) , accuracy =  0.8666666666666
```

```
### breast_cancer
# RANDOM_SEED = 42

gen nevals  max         avg
0   20      0.927946    0.808865
1   15      0.929669    0.889953
2   15      0.929669    0.893562
3   15      0.929669    0.893683
```

```
4   16      0.934932    0.839395
5   17      0.934932    0.912204
6   14      0.934932    0.895351
7   16      0.934932    0.908839
8   17      0.934932    0.900869
9   16      0.934932    0.845574
10  15      0.934932    0.900429
- Best solution is:  'hidden_layer_sizes'=(15, 8, 10, 4) , accuracy =  0.934932463
```

RESUME

Again ... for various DATASETS we can obtain NNs with the **very different**:

- **performance** (accuracy),
- **the number of nodes** in layers,
- **the number of layers**.

The reason is evident here: the different number of features and their different contribution to fitness function.

with various MAX number of layers ...

(let's try it as a self-guided learning!)

```

```

Part 2. NN Hyperparameter Tuning

```
from sklearn import model_selection
from sklearn import datasets
from sklearn.neural_network import MLPClassifier

from sklearn.exceptions import ConvergenceWarning
from sklearn.utils.testing import ignore_warnings

from math import floor

class MlpHyperparametersTest:

    NUM_FOLDS = 5

    # Only hyperparameters!
    HIDDEN_LAYER_SIZES = [13, 4, 7]

    def __init__(self, randomSeed):

        self.randomSeed = randomSeed
        self.initDataset()
```

```
        self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS, random_state=s

    def initDataset(self):
        self.data = datasets.load_iris()

        self.X = self.data['data']
        self.y = self.data['target']


    # params contains floats representing the following:
    # 'hidden_layer_sizes': up to 4 positive integers
    # 'activation': {'tanh', 'relu', 'logistic'},
    # 'solver': {'sgd', 'adam', 'lbfgs'},
    # 'alpha': float,
    # 'learning_rate': {'constant', 'invscaling', 'adaptive'}
    def convertParams(self, params):

        # transform the layer sizes from float (possibly negative) values into hid
#        if round(params[1]) <= 0:
#            hiddenLayerSizes = round(params[0]),
#        elif round(params[2]) <= 0:
#            hiddenLayerSizes = (round(params[0]), round(params[1]))
#        elif round(params[3]) <= 0:
#            hiddenLayerSizes = (round(params[0]), round(params[1]), round(params[
#        else:
#            hiddenLayerSizes = (round(params[0]), round(params[1]), round(params[

#        activation = ['tanh', 'relu', 'logistic'][floor(params[4])]
#        solver = ['sgd', 'adam', 'lbfgs'][floor(params[5])]
#        alpha = params[6]
#        learning_rate = ['constant', 'invscaling', 'adaptive'][floor(params[7])]

        activation = ['tanh', 'relu', 'logistic'][floor(params[0])]
        solver = ['sgd', 'adam', 'lbfgs'][floor(params[1])]
        alpha = params[2]
        learning_rate = ['constant', 'invscaling', 'adaptive'][floor(params[3])]

        #return hiddenLayerSizes, activation, solver, alpha, learning_rate
        return activation, solver, alpha, learning_rate

    @ignore_warnings(category=ConvergenceWarning)
    def getAccuracy(self, params):
        #hiddenLayerSizes, activation, solver, alpha, learning_rate = self.convert
        activation, solver, alpha, learning_rate = self.convertParams(params)

        self.classifier = MLPClassifier(random_state=self.randomSeed,
                                        #hidden_layer_sizes=hiddenLayerSizes,
                                        hidden_layer_sizes=self.HIDDEN_LAYER_SIZES
                                        activation=activation,
                                        solver=solver,
                                        alpha=alpha,
                                        learning_rate=learning_rate)

        cv_results = model_selection.cross_val_score(self.classifier,
                                                     self.X,
```

```
                                                         self.y,
                                                         cv=self.kfold,
                                                         scoring='accuracy')

        return cv_results.mean()

    def formatParams(self, params):
        #hiddenLayerSizes, activation, solver, alpha, learning_rate = self.convert
        activation, solver, alpha, learning_rate = self.convertParams(params)
#        return "'hidden_layer_sizes'={}\n " \
        return "'activation'='{}'\n " \
               "'solver'='{}'\n " \
               "'alpha'={}\n " \
               "'learning_rate'='{}'"\
               .format(activation, solver, alpha, learning_rate)
            #.format(hiddenLayerSizes, activation, solver, alpha, learning_rate)
```

```
from deap import base
from deap import creator
from deap import tools

import random
import numpy


# boundaries for all parameters:
# 'hidden_layer_sizes': first four values
# 'activation': ['tanh', 'relu', 'logistic'] -> 0, 1, 2
# 'solver': ['sgd', 'adam', 'lbfgs'] -> 0, 1, 2
# 'alpha': float in the range of [0.0001, 2.0],
# 'learning_rate': ['constant', 'invscaling', 'adaptive'] -> 0, 1, 2
#BOUNDS_LOW =  [ 5,  -5, -10, -20, 0,     0,     0.0001, 0    ]
#BOUNDS_HIGH = [15,  10,  10,  10, 2.999, 2.999, 2.0,    2.999]

# Only hyperparameters!
BOUNDS_LOW =  [0,     0,     0.0001, 0    ]
BOUNDS_HIGH = [2.999, 2.999, 2.0,    2.999]


NUM_OF_PARAMS = len(BOUNDS_HIGH)

# Genetic Algorithm constants:
POPULATION_SIZE = 20
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.5   # probability for mutating an individual
MAX_GENERATIONS = 10
HALL_OF_FAME_SIZE = 3
CROWDING_FACTOR = 10.0  # crowding factor for crossover and mutation

# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

```
# create the classifier accuracy test class:
test = MlpHyperparametersTest(RANDOM_SEED)

toolbox = base.Toolbox()

# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))

# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)

# define the layer size attributes individually:
for i in range(NUM_OF_PARAMS):
    # "attribute_0", "attribute_1", ...
    toolbox.register("attribute_" + str(i),
                     random.uniform,
                     BOUNDS_LOW[i],
                     BOUNDS_HIGH[i])

# create a tuple containing an attribute generator for each param searched:
attributes = ()
for i in range(NUM_OF_PARAMS):
    attributes = attributes + (toolbox.__getattribute__("attribute_" + str(i)),)

# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator",
                 tools.initCycle,
                 creator.Individual,
                 attributes,
                 n=1)

# create the population operator to generate a list of individuals:
toolbox.register("populationCreator",
                 tools.initRepeat,
                 list,
                 toolbox.individualCreator)



# fitness calculation
def classificationAccuracy(individual):
    return test.getAccuracy(individual),



toolbox.register("evaluate", classificationAccuracy)

# genetic operators:mutFlipBit

# genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)
```

```
toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/model_selection/_split.py:296:
  FutureWarning
/usr/local/lib/python3.7/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
/usr/local/lib/python3.7/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)

end = time.time()
time_HYP = end - start
print("Time Elapsed = ", time_HYP)

# print best solution found:
print("Best solution is: ", test.formatParams(hof.items[0]))
print("Accuracy = %1.5f" % hof.items[0].fitness.values[0])

# extract statistics:
maxFitnessValues_HYP, meanFitnessValues_HYP = logbook.select("max", "avg")
```

```
***************************
gen     nevals  max             avg
0       20      0.946667        0.362
1       15      0.946667        0.599667
2       16      0.946667        0.864333
```

```
3       16      0.946667        0.927333
4       17      0.946667        0.944667
5       14      0.946667        0.887
6       15      0.946667        0.944667
7       14      0.946667        0.946
8       16      0.946667        0.907667
9       15      0.946667        0.945
10      17      0.946667        0.946
Time Elapsed =  92.3740484714508
Best solution is:  'activation'='logistic'
 'solver'='lbfgs'
 'alpha'=0.17139833879055075
 'learning_rate'='invscaling'
Accuracy = 0.94667
```

Part 3. NN Architecture + Hyperparameter Tuning

```
from sklearn import model_selection
from sklearn import datasets
from sklearn.neural_network import MLPClassifier

from sklearn.exceptions import ConvergenceWarning
from sklearn.utils.testing import ignore_warnings

from math import floor

class MlpHyperparametersTest:

    NUM_FOLDS = 5

    def __init__(self, randomSeed):

        self.randomSeed = randomSeed
        self.initDataset()
        self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS, random_state=s

    def initDataset(self):
        self.data = datasets.load_iris()

        self.X = self.data['data']
        self.y = self.data['target']


    # params contains floats representing the following:
    # 'hidden_layer_sizes': up to 4 positive integers
    # 'activation': {'tanh', 'relu', 'logistic'},
    # 'solver': {'sgd', 'adam', 'lbfgs'},
    # 'alpha': float,
    # 'learning_rate': {'constant', 'invscaling', 'adaptive'}
    def convertParams(self, params):

        # transform the layer sizes from float (possibly negative) values into hid
        if round(params[1]) <= 0:
```

```
            hiddenLayerSizes = round(params[0]),
        elif round(params[2]) <= 0:
            hiddenLayerSizes = (round(params[0]), round(params[1]))
        elif round(params[3]) <= 0:
            hiddenLayerSizes = (round(params[0]), round(params[1]), round(params[2
        else:
            hiddenLayerSizes = (round(params[0]), round(params[1]), round(params[2

        activation = ['tanh', 'relu', 'logistic'][floor(params[4])]
        solver = ['sgd', 'adam', 'lbfgs'][floor(params[5])]
        alpha = params[6]
        learning_rate = ['constant', 'invscaling', 'adaptive'][floor(params[7])]

        return hiddenLayerSizes, activation, solver, alpha, learning_rate

    @ignore_warnings(category=ConvergenceWarning)
    def getAccuracy(self, params):
        hiddenLayerSizes, activation, solver, alpha, learning_rate = self.convertP

        self.classifier = MLPClassifier(random_state=self.randomSeed,
                                        hidden_layer_sizes=hiddenLayerSizes,
                                        activation=activation,
                                        solver=solver,
                                        alpha=alpha,
                                        learning_rate=learning_rate)

        cv_results = model_selection.cross_val_score(self.classifier,
                                                     self.X,
                                                     self.y,
                                                     cv=self.kfold,
                                                     scoring='accuracy')

        return cv_results.mean()

    def formatParams(self, params):
        hiddenLayerSizes, activation, solver, alpha, learning_rate = self.convertP
        return "'hidden_layer_sizes'={}\n " \
               "'activation'='{}'\n " \
               "'solver'='{}'\n " \
               "'alpha'={}\n " \
               "'learning_rate'='{}'"\
            .format(hiddenLayerSizes, activation, solver, alpha, learning_rate)
```

```
from deap import base
from deap import creator
from deap import tools

import random
import numpy



# boundaries for all parameters:
# 'hidden_layer_sizes': first four values
# 'activation': ['tanh', 'relu', 'logistic'] -> 0, 1, 2
```

```
# 'solver': ['sgd', 'adam', 'lbfgs'] -> 0, 1, 2
# 'alpha': float in the range of [0.0001, 2.0],
# 'learning_rate': ['constant', 'invscaling', 'adaptive'] -> 0, 1, 2
BOUNDS_LOW =  [ 5,  -5, -10, -20, 0,     0,     0.0001, 0    ]
BOUNDS_HIGH = [15,  10,  10,  10, 2.999, 2.999, 2.0,    2.999]

NUM_OF_PARAMS = len(BOUNDS_HIGH)

# Genetic Algorithm constants:
POPULATION_SIZE = 20
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.5   # probability for mutating an individual
MAX_GENERATIONS = 10
HALL_OF_FAME_SIZE = 3
CROWDING_FACTOR = 10.0  # crowding factor for crossover and mutation

# set the random seed:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# create the classifier accuracy test class:
test = MlpHyperparametersTest(RANDOM_SEED)

toolbox = base.Toolbox()

# define a single objective, maximizing fitness strategy:
creator.create("FitnessMax", base.Fitness, weights=(1.0,))

# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMax)

# define the layer size attributes individually:
for i in range(NUM_OF_PARAMS):
    # "attribute_0", "attribute_1", ...
    toolbox.register("attribute_" + str(i),
                     random.uniform,
                     BOUNDS_LOW[i],
                     BOUNDS_HIGH[i])

# create a tuple containing an attribute generator for each param searched:
attributes = ()
for i in range(NUM_OF_PARAMS):
    attributes = attributes + (toolbox.__getattribute__("attribute_" + str(i)),)

# create the individual operator to fill up an Individual instance:
toolbox.register("individualCreator",
                 tools.initCycle,
                 creator.Individual,
                 attributes,
                 n=1)

# create the population operator to generate a list of individuals:
toolbox.register("populationCreator",
                 tools.initRepeat,
                 list,
```

```
                 toolbox.individualCreator)


# fitness calculation
def classificationAccuracy(individual):
    return test.getAccuracy(individual),


toolbox.register("evaluate", classificationAccuracy)

# genetic operators:mutFlipBit

# genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/model_selection/_split.py:296:
  FutureWarning
/usr/local/lib/python3.7/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
/usr/local/lib/python3.7/dist-packages/deap/creator.py:141: RuntimeWarning: A
  RuntimeWarning)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('****************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
```

```
                                                                stats=stats,
                                                                halloffame=hof,
                                                                verbose=True)

end = time.time()
time_NNA_HYP = end - start
print("Time Elapsed = ", time_NNA_HYP)

# print best solution found:
print("Best solution is: ", test.formatParams(hof.items[0]))
print("Accuracy = %1.5f" % hof.items[0].fitness.values[0])

# extract statistics:
maxFitnessValues_NNA_HYP, meanFitnessValues_NNA_HYP = logbook.select("max", "avg")
```

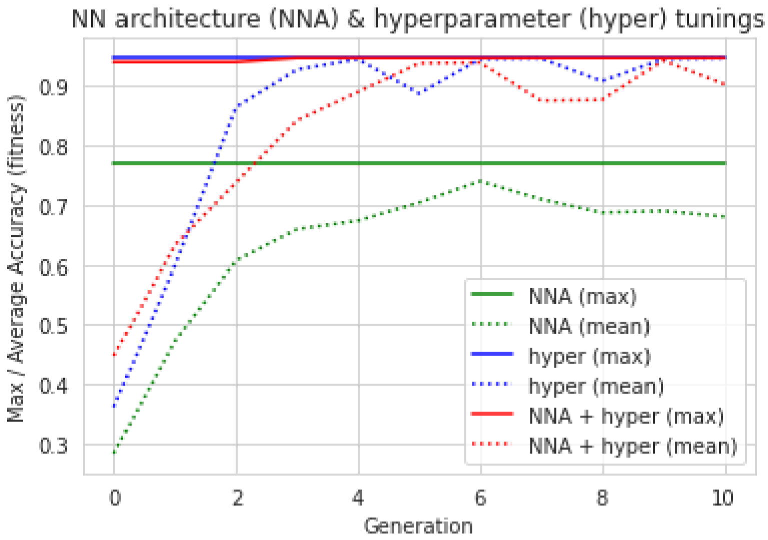
```
***************************
gen     nevals  max     avg
0       20      0.94    0.448
1       16      0.94    0.633
2       15      0.94    0.737667
3       16      0.946667        0.842
4       17      0.946667        0.889667
5       15      0.946667        0.937667
6       16      0.946667        0.939
7       16      0.946667        0.875
8       16      0.946667        0.876333
9       14      0.946667        0.942333
10      16      0.946667        0.902667
Time Elapsed =  83.84976434707642
Best solution is:  'hidden_layer_sizes'=(8, 7)
 'activation'='relu'
 'solver'='lbfgs'
 'alpha'=0.563775972907702
 'learning_rate'='adaptive'
Accuracy = 0.94667
```

Comparison Plots

Accuracy

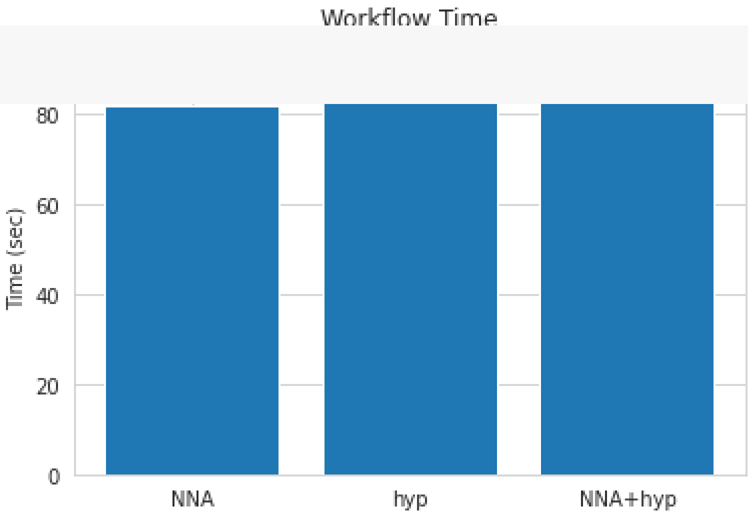
```
sns.set_style("whitegrid")

# Classic grid search solution
#plt.hlines(accuracy_classic_solution, 0, 5, linestyle = 'solid', label='Classic g

# NN architecture
plt.plot(maxFitnessValues_NNA, color='green', label='NNA (max)')
plt.plot(meanFitnessValues_NNA, color='green', linestyle = 'dotted', label='NNA (m

# NN hyperparameter
plt.plot(maxFitnessValues_HYP, color='blue', label='hyper (max)')
```

```
plt.plot(meanFitnessValues_HYP, color='blue', linestyle = 'dotted', label='hyper (

# NN architecture + hyperparameter
plt.plot(maxFitnessValues_NNA_HYP, color='red', label='NNA + hyper (max)')
plt.plot(meanFitnessValues_NNA_HYP, color='red', linestyle = 'dotted', label='NNA

plt.xlabel('Generation')
plt.ylabel('Max / Average Accuracy (fitness)')
plt.title('NN architecture (NNA) & hyperparameter (hyper) tunings')
plt.legend()
plt.show()
```



Time

```
import matplotlib.pyplot as plt

x = ['NNA','hyp','NNA+hyp']
y = [time_NNA,time_HYP,time_NNA_HYP]
plt.bar(x,y)
plt.ylabel('Time (sec)')
plt.title('Workflow Time')
plt.show()
```


Workflow Time
Time (sec)
80
60
40
20
0
NNA
hyp
NNA+hyp

Lab 6 - DEMO B - Applications of EA (actually GA here) for Reinforcement Learning

based on (C) OpenAI, Heaton, Moore, Varoquaux, Grobler, Wirsansky work

Brief Content:

- DEAP installation (**every time after start of Colab VM!**),
- components needed for the GA workflow,
- *Reinforcement Learning (RL) problems:*
    - MountainCar-v0,
    - MountainCarContinuous-v0,
    - CartPole-v1
    - ...
- performance comparison (fitness function and run time).

By the end of this lab you will know:

- again, how to use the DEAP framework's built-in algorithms to produce concise code
- how to solve the *Reinforcement Learning* problem using a GA-based solutions for search of solutions,
- how to experiment with various settings of the GA and interpret the differences in the results.

Part 1.Installation and import of libraries

```
! pip install deap
```

```
Collecting deap
  Downloading https://files.pythonhosted.org/packages/99/d1/803c7a387d8a7e686
     |████████████████████████████████| 163kB 5.6MB/s
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-package
Installing collected packages: deap
Successfully installed deap-1.3.1
```

Library to support RL algorithms

Gym is a toolkit for developing and comparing reinforcement learning algorithms.

It supports teaching agents everything from walking to playing games like Pong or Pinball.

```
! pip install gym
```

```
Requirement already satisfied: gym in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /usr/local/lib/python
Requirement already satisfied: numpy>=1.10.4 in /usr/local/lib/python3.7/dist
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /usr/local/lib/py
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packag
```

Libraries to Render OpenAI Gym Environments in Colab

It is possible to visualize the activities performed in Gym (game your agent is playing), even on Colab. This section provides information on how to generate a video in Colab that shows you an episode of the game your agent is playing.

```
%%time
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
```

```
CPU times: user 42.1 ms, sys: 11.1 ms, total: 53.2 ms
Wall time: 12.3 s
```

```
%%time
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
```

```
Collecting setuptools
  Downloading https://files.pythonhosted.org/packages/60/6a/dd9533a
     |████████████████████████████████| 788kB 4.4MB/s
ERROR: datascience 0.10.6 has requirement folium==0.2.1, but you'll
Installing collected packages: setuptools
  Found existing installation: setuptools 54.0.0
    Uninstalling setuptools-54.0.0:
      Successfully uninstalled setuptools-54.0.0
Successfully installed setuptools-54.1.1
CPU times: user 81.7 ms, sys: 32.5 ms, total: 114 ms
```

IMPORTANT: you should restart runtime!

Functions to visualize Gym-game-scenarios in Colab

Next we define functions used to show the video by adding it to the Colab notebook.

```
import gym
from gym.wrappers import Monitor
import glob
import io
import base64
from IPython.display import HTML
from pyvirtualdisplay import Display
from IPython import display as ipythondisplay

display = Display(visible=0, size=(1400, 900))
display.start()

"""
Utility functions to enable video recording of gym environment
and displaying it.
To enable video, just do "env = wrap_env(env)""
"""

def show_video():
  mp4list = glob.glob('video/*.mp4')
  if len(mp4list) > 0:
    mp4 = mp4list[0]
    video = io.open(mp4, 'r+b').read()
    encoded = base64.b64encode(video)
    ipythondisplay.display(HTML(data='''<video alt="test" autoplay
                loop controls style="height: 400px;">
                <source src="data:video/mp4;base64,{0}" type="video/mp4" />
             </video>'''.format(encoded.decode('ascii'))))
  else:
    print("Could not find video")


def wrap_env(env):
  env = Monitor(env, './video', force=True)
  return env
```

Part 2. GA Solution for RL problem - MountainCar-v0

MountainCar - Problem Description

A car is on a one-dimensional track, positioned between two "mountains".

The **goal** is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

This problem was first described by Andrew Moore in his PhD thesis:

A. Moore, Efficient Memory-Based Learning for Robot Control, PhD thesis, University of Cambridge, 1990. **Cited in 363 sources.**

Import Python libraries

In these and other lectures, we will use various Python packages:

- NumPy
- Matplotlib
- Seaborn

They are already pre-installed in Colab. Let's import them by the following code.

```
import gym
import time
import pickle
import random
import numpy

# for plotting
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# If you run this notebook again, please, clean the local directory.
! rm -r ./video
! rm ./*pickle
```

```
rm: cannot remove './video': No such file or directory
rm: cannot remove './*pickle': No such file or directory
```

Actors - Car

```
MAX_STEPS = 200
FLAG_LOCATION = 0.5

class MountainCar:
```

```
    def __init__(self, randomSeed):

        #self.env = gym.make('MountainCar-v0')
        self.env = wrap_env(gym.make("MountainCar-v0"))
        self.env.seed(randomSeed)

    def __len__(self):
        return MAX_STEPS

    def getScore(self, actions):
        """
        calculates the score of a given solution, represented by the list of actio
        by initiating an episode of the Mountain-Car environment and running it wi
        Lower score is better.
        :param actions: a list of actions (values 0, 1, or 2) to be fed into the m
        :return: the calculated score value
        """

        # start a new episode:
        self.env.reset()

        actionCounter = 0

        # feed the actions to the environment:
        for action in actions:
            actionCounter += 1

            # provide an action and get feedback:
            observation, reward, done, info = self.env.step(action)

            # episode over - either the car hit the flag, or 200 actions processed
            if done:
                break

        # evaluate the results to produce the score:
        if actionCounter < MAX_STEPS:
            # the car hit the flag:
            # start from a score of 0
            # reward further for a smaller amount of steps
            score = 0 - (MAX_STEPS - actionCounter)/MAX_STEPS
        else:
            # the car did not hit the flag:
            # reward according to distance from flag
            score = abs(observation[0] - FLAG_LOCATION)  # we want to minimize tha

        return score

    def saveActions(self, actions):
        """
        serializes and saves a list of actions using pickle
        :param actions: a list of actions (values 0, 1, or 2) to be fed into the m
        """
        savedActions = []
        for action in actions:
            savedActions.append(action)
```

```
        pickle.dump(savedActions, open("mountain-car-data.pickle", "wb"))

    def replaySavedActions(self):
        """
        deserializes a saved list of actions and replays it
        """
        savedActions = pickle.load(open("mountain-car-data.pickle", "rb"))
        self.replay(savedActions)

    def replay(self, actions):
        """
        renders the environment and replays list of actions into it, to visualize
        :param actions: a list of actions (values 0, 1, or 2) to be fed into the m
        """

        # start a new episode:
        observation = self.env.reset()

        # start rendering:
        self.env.render()

        actionCounter = 0

        # replay the given actions by feeding them into the environment:
        for action in actions:

            actionCounter += 1
            self.env.render()
            observation, reward, done, info = self.env.step(action)
            print(actionCounter, ": --------------------------")
            print("action = ", action)
            print("observation = ", observation)
            print("distance from flag = ", abs(observation[0] - 0.5))
            print()

            if done:
                break
            else:
                time.sleep(0.02)

        self.env.close()

    def replayVideo(self):
        #self.env.close()
        show_video()
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

```
# Create the instance of the MountainCar class:
```

GA Solution

```
from deap import base
from deap import creator
from deap import tools
from deap import algorithms
```

```
# Genetic Algorithm constants:
POPULATION_SIZE = 100
P_CROSSOVER = 0.9  # probability for crossover
P_MUTATION = 0.5   # probability for mutating an individual
MAX_GENERATIONS = 80
HALL_OF_FAME_SIZE = 20
```

Genetic Tools

```
toolbox = base.Toolbox()

# define a single objective, minimizing fitness strategy:
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))

# create the Individual class based on list:
creator.create("Individual", list, fitness=creator.FitnessMin)

# create an operator that randomly returns 0, 1 or 2:
toolbox.register("zeroOneOrTwo", random.randint, 0, 2)

# create an operator that generates a list of individuals:
toolbox.register("individualCreator",
                 tools.initRepeat,
                 creator.Individual,
                 toolbox.zeroOneOrTwo,
                 len(car))

# create the population operator to generate a list of individuals:
toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCr


# fitness calculation
def getCarScore(individual):
    return car.getScore(individual),  # return a tuple

toolbox.register("evaluate", getCarScore)

# genetic operators for binary list:
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=0, up=2, indpb=1.0/len(car))
```

Elitism Tools

```
def eaSimpleWithElitism(population, toolbox, cxpb, mutpb, ngen, stats=None,
             halloffame=None, verbose=__debug__):
    """This algorithm is similar to DEAP eaSimple() algorithm, with the modificati
    halloffame is used to implement an elitism mechanism. The individuals containe
    halloffame are directly injected into the next generation and are not subject
    genetic operators of selection, crossover and mutation.
    """
    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])

    # Evaluate the individuals with an invalid fitness
    invalid_ind = [ind for ind in population if not ind.fitness.valid]
    fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
    for ind, fit in zip(invalid_ind, fitnesses):
        ind.fitness.values = fit

    if halloffame is None:
        raise ValueError("halloffame parameter must not be empty!")

    halloffame.update(population)
    hof_size = len(halloffame.items) if halloffame.items else 0

    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=len(invalid_ind), **record)
    if verbose:
        print(logbook.stream)

    # Begin the generational process
    for gen in range(1, ngen + 1):

        # Select the next generation individuals
        offspring = toolbox.select(population, len(population) - hof_size)

        # Vary the pool of individuals
        offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = toolbox.map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit

        # add the best back to population:
        offspring.extend(halloffame.items)

        # Update the hall of fame with the generated individuals
        halloffame.update(offspring)

        # Replace the current population by the offspring
        population[:] = offspring
```

```
        # Append the current generation statistics to the logbook
        record = stats.compile(population) if stats else {}
        logbook.record(gen=gen, nevals=len(invalid_ind), **record)
        if verbose:
            print(logbook.stream)

    return population, logbook
```

GA Workflow

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", numpy.min)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_GA = end - start
print("Time Elapsed = ", time_GA)
```

```
***************************
gen     nevals  min             avg
0       100     0.659205        1.02616
1       78      0.659205        0.970209
2       77      0.659205        0.906442
3       76      0.659205        0.841666
4       74      0.659205        0.791741
5       78      0.581075        0.754467
6       76      0.581075        0.712045
7       77      0.551261        0.676387
8       74      0.455182        0.64108
9       76      0.455182        0.610402
10      79      0.455182        0.586111
11      77      0.441709        0.554078
12      75      0.415877        0.517882
13      78      0.415877        0.494984
14      77      0.36574         0.471906
```

```
15      79      0.36574         0.468517
16      80      0.362115        0.442253
17      74      0.34458         0.420221
18      77      0.325772        0.411453
19      75      0.325772        0.407308
20      77      0.325772        0.396775
21      76      0.312125        0.394258
22      73      0.312125        0.381892
23      74      0.276737        0.367306
24      79      0.276737        0.353065
25      75      0.263841        0.346514
26      73      0.262451        0.326677
27      75      0.241413        0.320918
28      79      0.241413        0.314168
29      73      0.241413        0.312155
30      76      0.224605        0.30864
31      76      0.224605        0.299474
32      77      0.224605        0.302423
33      78      0.20032         0.298204
34      76      0.20032         0.29605
35      75      0.20032         0.27704
36      78      0.196936        0.270632
37      76      0.196936        0.27497
38      73      0.196936        0.270335
39      75      0.194873        0.265736
40      76      0.192338        0.264394
41      70      0.116166        0.243454
42      74      0.0900748       0.256206
43      77      0.0694396       0.250983
44      75      0.0694396       0.244451
45      76      0.0694396       0.220137
46      78      0.0694396       0.208847
47      71      0.0400128       0.205876
48      74      0.0400128       0.186488
49      75      0.0400128       0.189645
50      77      0.0400128       0.199232
51      78      0.0400128       0.174549
52      75      0.0179342       0.152889
53      79      0.0179342       0.15147
54      79      0.00603528      0.139074
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])

# extract statistics:
minFitnessValues_GA, meanFitnessValues_GA = logbook.select("min", "avg")
print('History of minFitnessValues_GA =',minFitnessValues_GA)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA)

# save best solution for a replay:
car.saveActions(best)
```

```
Best solution:  [0, 1, 2, 0, 0, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, 0, 1, 2, 1,
Best FitnessMin = -0.02000
History of minFitnessValues_GA = [0.6592052516766607, 0.6592052516766607, 0.6
History of meanFitnessValues_GA = [1.0261591751881378, 0.9702094529954302, 0.
```

```
# Replay the best solution - TEXT version
car.replaySavedActions()
```

```
1 : -------------------------
action =  0
observation =  [-0.54270019 -0.00086328]
distance from flag =  1.0427001917084184

2 : -------------------------
action =  1
observation =  [-0.54342029 -0.0007201 ]
distance from flag =  1.0434202917150182

3 : -------------------------
action =  2
observation =  [-5.42991818e-01  4.28473768e-04]
distance from flag =  1.0429918179471604

4 : -------------------------
action =  0
observation =  [-5.43417978e-01 -4.26160453e-04]
distance from flag =  1.0434179784000257

5 : -------------------------
action =  0
observation =  [-0.54469558 -0.0012776 ]
distance from flag =  1.0446955823976336

6 : -------------------------
action =  1
observation =  [-0.54581507 -0.00111948]
distance from flag =  1.0458150659568424

7 : -------------------------
action =  2
observation =  [-5.45768051e-01  4.70152879e-05]
distance from flag =  1.0457680506689189

8 : -------------------------
action =  2
observation =  [-0.54455489  0.00121316]
distance from flag =  1.0445548883672975

9 : -------------------------
action =  2
observation =  [-0.54218466  0.00237023]
distance from flag =  1.0421846587338575

10 : -------------------------
action =  2
observation =  [-0.53867511  0.00350955]
distance from flag =  1.0386751071930986

11 : -------------------------
action =  2
observation =  [-0.53405252  0.00462259]
distance from flag =  1.0340525217127836
```

```
12 : -------------------------
action =  2
```

```
car.replayVideo()
```

Part 3.What about the solution dependence on GA conditions?

... with various RANDOM_SEED ...

Results for various RANDOM_SEEDs

RANDOM_SEED = 42

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 42
random.seed(RANDOM_SEED)

# Create the instance of the MountainCar class:
car = MountainCar(RANDOM_SEED)
```

```python
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", numpy.min)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                          toolbox,
                                          cxpb=P_CROSSOVER,
                                          mutpb=P_MUTATION,
                                          ngen=MAX_GENERATIONS,
                                          stats=stats,
                                          halloffame=hof,
                                          verbose=True)
end = time.time()
time_42 = end - start
print("Time Elapsed = ", time_42)
```

```
***************************
gen	nevals	min     	avg     
0  	100   	0.659205	1.02616 
1  	78    	0.659205	0.970209
2  	77    	0.659205	0.906442
3  	76    	0.659205	0.841666
4  	74    	0.659205	0.791741
5  	78    	0.581075	0.754467
6  	76    	0.581075	0.712045
7  	77    	0.551261	0.676387
8  	74    	0.455182	0.64108 
9  	76    	0.455182	0.610402
10 	79    	0.455182	0.586111
11 	77    	0.441709	0.554078
12 	75    	0.415877	0.517882
13 	78    	0.415877	0.494984
14 	77    	0.36574 	0.471906
15 	79    	0.36574 	0.468517
16 	80    	0.362115	0.442253
17 	74    	0.34458 	0.420221
18 	77    	0.325772	0.411453
19 	75    	0.325772	0.407308
20 	77    	0.325772	0.396775
21 	76    	0.312125	0.394258
22 	73    	0.312125	0.381892
23 	74    	0.276737	0.367306
24 	79    	0.276737	0.353065
25 	75    	0.263841	0.346514
26 	73    	0.262451	0.326677
27 	75    	0.241413	0.320918
28 	79    	0.241413	0.314168
```

```
29      73      0.241413        0.312155
30      76      0.224605        0.30864
31      76      0.224605        0.299474
32      77      0.224605        0.302423
33      78      0.20032         0.298204
34      76      0.20032         0.29605
35      75      0.20032         0.27704
36      78      0.196936        0.270632
37      76      0.196936        0.27497
38      73      0.196936        0.270335
39      75      0.194873        0.265736
40      76      0.192338        0.264394
41      70      0.116166        0.243454
42      74      0.0900748       0.256206
43      77      0.0694396       0.250983
44      75      0.0694396       0.244451
45      76      0.0694396       0.220137
46      78      0.0694396       0.208847
47      71      0.0400128       0.205876
48      74      0.0400128       0.186488
49      75      0.0400128       0.189645
50      77      0.0400128       0.199232
51      78      0.0400128       0.174549
52      75      0.0179342       0.152889
53      79      0.0179342       0.15147
54      79      0.00603528      0.139074
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
minFitnessValues_GA_42, meanFitnessValues_GA_42 = logbook.select("min", "avg")
print('History of minFitnessValues_GA =',minFitnessValues_GA_42)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_42)
```

```
Best solution:  [0, 1, 2, 0, 0, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, 0, 1, 2, 1,
Best FitnessMin = -0.02000
History of minFitnessValues_GA = [0.6592052516766607, 0.6592052516766607, 0.6
History of meanFitnessValues_GA = [1.0261591751881378, 0.9702094529954302, 0.
```

RANDOM_SEED = 666

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 666
random.seed(RANDOM_SEED)

# Create the instance of the MountainCar class:
car = MountainCar(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", numpy.min)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_666 = end - start
print("Time Elapsed = ", time_666)
```

```
***************************
gen     nevals  min             avg
0       100     0.824257        1.03458
1       80      0.768396        0.961691
2       76      0.695209        0.894844
3       75      0.631651        0.838345
4       75      0.601543        0.789886
5       74      0.572929        0.742351
6       72      0.577929        0.711419
7       77      0.577929        0.679999
8       72      0.534989        0.641107
9       72      0.534989        0.622413
10      75      0.512573        0.609433
11      78      0.439645        0.592019
12      74      0.439645        0.571789
13      72      0.437972        0.550431
14      79      0.418771        0.53019
15      77      0.418771        0.51943
16      76      0.41794         0.48955
17      76      0.410739        0.488176
18      77      0.372919        0.473319
19      79      0.372919        0.464156
20      77      0.252726        0.453411
21      72      0.252726        0.429531
22      71      0.252726        0.402247
23      77      0.252726        0.366217
24      78      0.23005         0.353316
25      74      0.211242        0.324498
26      76      0.211242        0.316425
27      78      0.20121         0.304086
28      79      0.20121         0.291859
```

```
29      76      0.197576        0.2995
30      76      0.197576        0.287731
31      73      0.173191        0.274362
32      77      0.173191        0.275585
33      79      0.173191        0.264559
34      75      0.173191        0.272344
35      77      0.168406        0.264175
36      77      0.162313        0.264218
37      74      0.162313        0.248879
38      72      0.162313        0.254829
39      72      0.162313        0.241994
40      79      0.1499          0.243321
41      78      0.148408        0.236391
42      77      0.148408        0.225184
43      75      0.131007        0.216867
44      75      0.123164        0.220414
45      77      0.123164        0.218808
46      74      0.108418        0.211137
47      72      0.106098        0.203032
48      77      0.106098        0.208214
49      71      0.0875458       0.202567
50      73      0.0875458       0.194274
51      77      0.0875458       0.197749
52      76      0.0875458       0.19476
53      77      0.0743577       0.196832
54      77      0.0743577       0.174699
55      79      0.0743577       0.180216
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
minFitnessValues_GA_666, meanFitnessValues_GA_666 = logbook.select("min", "avg")
print('History of minFitnessValues_GA =',minFitnessValues_GA_666)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_666)
```

```
Best solution:  [2, 2, 1, 1, 0, 1, 0, 0, 1, 2, 2, 1, 1, 1, 1, 0, 2, 1, 2, 1,
Best FitnessMin = -0.01500
History of minFitnessValues_GA = [0.8242573388334045, 0.7683963861683756, 0.6
History of meanFitnessValues_GA = [1.0345837515118468, 0.9616914914820174, 0.
```

RANDOM_SEED = 1042

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCar class:
car = MountainCar(RANDOM_SEED)
```

```python
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", numpy.min)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_1042 = end - start
print("Time Elapsed = ", time_1042)
```

```
***************************
gen     nevals  min             avg
0       100     0.758144        1.03179
1       75      0.713363        0.966288
2       75      0.713363        0.906746
3       75      0.639515        0.850049
4       78      0.577392        0.789484
5       77      0.577392        0.737554
6       78      0.513466        0.692913
7       75      0.513466        0.644129
8       79      0.482627        0.617077
9       75      0.405722        0.593555
10      75      0.405722        0.575717
11      75      0.383543        0.55207
12      74      0.378358        0.523467
13      78      0.359471        0.508689
14      79      0.349516        0.477198
15      75      0.349516        0.452328
16      74      0.349516        0.435158
17      79      0.349516        0.41954
18      76      0.34445         0.407666
19      76      0.296326        0.397487
20      78      0.296326        0.382383
21      77      0.271068        0.388545
22      72      0.260843        0.373072
23      79      0.260843        0.374041
24      73      0.260843        0.361229
25      75      0.260843        0.356421
26      74      0.218994        0.343781
27      77      0.218994        0.338026
28      76      0.218994        0.337831
```

```
29      77      0.197813        0.321685
30      76      0.160213        0.314113
31      80      0.160213        0.3191
32      80      0.160213        0.299479
33      76      0.160213        0.29697
34      73      0.160213        0.284896
35      78      0.155248        0.286583
36      72      0.160213        0.272549
37      78      0.14066         0.273315
38      77      0.14066         0.254534
39      79      0.134304        0.259926
40      73      0.134304        0.253324
41      77      0.108358        0.245759
42      79      0.108358        0.239136
43      75      0.108358        0.233271
44      74      0.108358        0.218148
45      78      0.108358        0.212218
46      75      0.102538        0.202415
47      76      0.102538        0.199431
48      77      0.102538        0.204366
49      76      0.0873598       0.20761
50      78      0.0711741       0.198609
51      77      0.0711741       0.186059
52      74      0.0584212       0.177664
53      75      0.0532709       0.166839
54      80      0.0532709       0.162786
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
minFitnessValues_GA_1042, meanFitnessValues_GA_1042 = logbook.select("min", "avg")
print('History of minFitnessValues_GA =',minFitnessValues_GA_1042)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042)
```

```
Best solution:  [1, 0, 0, 2, 1, 2, 1, 2, 0, 2, 1, 2, 1, 2, 2, 2, 1, 2, 2, 1,
Best FitnessMin = -0.01500
History of minFitnessValues_GA = [0.7581441108997677, 0.713363151121015, 0.71
History of meanFitnessValues_GA = [1.031790151391004, 0.9662875012494746, 0.9
```

RESUME

For various RANDOM_SEED we can obtain **different**:

- **solutions** :) ... of course,
- **performance** (fitness function value),
- **history**.

The reason is the stochastic manner of parameter change during evolution.

... with various GA parameters ... like Crossover Probability

It takes a small change in *P_CROSSOVER* variable.

P_CROSSOVER = 0.1

```
P_CROSSOVER = 0.1  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCar class:
car = MountainCar(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", numpy.min)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_1042_CR0p1 = end - start
print("Time Elapsed = ", time_1042_CR0p1)
```

```
***************************
gen	nevals	min     	avg     
0  	100   	0.758144	1.03179 
1  	41    	0.743802	0.964259
2  	39    	0.743802	0.915064
3  	41    	0.739214	0.865538
4  	52    	0.583381	0.830942
5  	37    	0.583381	0.78981 
6  	42    	0.575982	0.754293
7  	48    	0.549486	0.728786
8  	39    	0.547788	0.703206
```

```
9       40      0.547788        0.658415
10      41      0.538602        0.629071
11      43      0.520597        0.600203
12      45      0.520597        0.581616
13      44      0.519325        0.566761
14      42      0.519325        0.556239
15      46      0.520321        0.558152
16      44      0.496319        0.551587
17      46      0.496319        0.552638
18      57      0.48621         0.555001
19      42      0.48621         0.5451
20      35      0.48621         0.541675
21      32      0.48621         0.533755
22      44      0.48621         0.536453
23      52      0.48621         0.538906
24      38      0.48621         0.527768
25      34      0.481001        0.521111
26      42      0.481001        0.521335
27      49      0.46186         0.520217
28      51      0.46186         0.522274
29      47      0.46186         0.520333
30      48      0.46186         0.513685
31      50      0.46186         0.515025
32      52      0.46186         0.512728
33      50      0.450984        0.502648
34      36      0.449244        0.492566
35      45      0.425699        0.490416
36      51      0.425699        0.485783
37      44      0.425699        0.484683
38      45      0.415825        0.477998
39      47      0.415751        0.472314
40      45      0.414466        0.473612
41      42      0.394263        0.460849
42      44      0.38147         0.452556
43      44      0.38147         0.453463
44      38      0.359843        0.445091
45      49      0.38147         0.442556
46      47      0.356461        0.434419
47      47      0.356461        0.438531
48      41      0.351673        0.429193
49      44      0.333198        0.420095
50      47      0.310826        0.418048
51      41      0.310826        0.404214
52      37      0.310826        0.398673
53      38      0.307904        0.392828
54      42      0.307904        0.385709
55      50      0.307904        0.378934
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
minFitnessValues_GA_1042_CR0p1, meanFitnessValues_GA_1042_CR0p1 = logbook.select("
print('History of minFitnessValues_GA =',minFitnessValues_GA_1042_CR0p1)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p1)
```

```
Best solution:  [1, 1, 2, 2, 1, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 1, 2, 0, 1, 1,
Best FitnessMin = 0.25583
History of minFitnessValues_GA = [0.7581441108997677, 0.743802202262433, 0.74
History of meanFitnessValues_GA = [1.031790151391004, 0.9642592656875997, 0.9
```

P_CROSSOVER = 0.2

```
P_CROSSOVER = 0.2  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCar class:
car = MountainCar(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", numpy.min)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('***************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_1042_CR0p2 = end - start
print("Time Elapsed = ", time_1042_CR0p2)
```

```
***************************
gen     nevals  min             avg
0       100     0.758144        1.03179
1       51      0.758144        0.963915
2       47      0.743435        0.90916
3       51      0.695507        0.859285
```

```
4       39      0.687698        0.819404
5       46      0.673627        0.787391
6       50      0.673627        0.752036
7       43      0.673627        0.730605
8       51      0.633845        0.717065
9       54      0.617011        0.699343
10      47      0.60558         0.682446
11      39      0.52083         0.665377
12      44      0.52083         0.642318
13      49      0.52083         0.631635
14      47      0.52083         0.619007
15      48      0.52083         0.605986
16      42      0.52083         0.594333
17      56      0.519022        0.586025
18      43      0.517601        0.582297
19      44      0.517601        0.572918
20      47      0.517601        0.566298
21      37      0.514145        0.561395
22      48      0.486861        0.555451
23      45      0.495736        0.552859
24      51      0.488352        0.553029
25      49      0.488352        0.547999
26      54      0.488352        0.549498
27      40      0.477017        0.538995
28      53      0.456922        0.535805
29      48      0.456922        0.53067
30      49      0.456922        0.52461
31      51      0.456922        0.519212
32      56      0.456922        0.515279
33      43      0.429074        0.509413
34      45      0.429074        0.502732
35      44      0.433134        0.500906
36      51      0.381015        0.492789
37      55      0.400434        0.494057
38      41      0.393918        0.484146
39      43      0.380597        0.463111
40      50      0.380597        0.457285
41      45      0.380597        0.446375
42      45      0.37735         0.436169
43      53      0.37735         0.4369
44      42      0.37735         0.433312
45      51      0.373786        0.433094
46      55      0.354377        0.425752
47      48      0.354377        0.42211
48      50      0.354377        0.42557
49      51      0.354377        0.429721
50      47      0.338672        0.427246
51      46      0.338672        0.417907
52      50      0.338672        0.414365
53      43      0.338672        0.412554
54      51      0.338672        0.408941
55      46      0.334169        0.405105
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])
```

```
# extract statistics:
minFitnessValues_GA_1042_CR0p2, meanFitnessValues_GA_1042_CR0p2 = logbook.select("
print('History of minFitnessValues_GA =',minFitnessValues_GA_1042_CR0p2)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p2)
```

```
Best solution:  [1, 1, 2, 2, 1, 0, 0, 1, 1, 0, 1, 2, 2, 2, 2, 1, 1, 2, 2, 1,
Best FitnessMin = 0.27027
History of minFitnessValues_GA = [0.7581441108997677, 0.7581441108997677, 0.7
History of meanFitnessValues_GA = [1.031790151391004, 0.9639150575310811, 0.9
```

P_CROSSOVER = 0.4

```
P_CROSSOVER = 0.4  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCar class:
car = MountainCar(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", numpy.min)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('****************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                                      toolbox,
                                                      cxpb=P_CROSSOVER,
                                                      mutpb=P_MUTATION,
                                                      ngen=MAX_GENERATIONS,
                                                      stats=stats,
                                                      halloffame=hof,
                                                      verbose=True)
end = time.time()
time_1042_CR0p4 = end - start
print("Time Elapsed = ", time_1042_CR0p4)
```

```
****************************
gen     nevals  min             avg
0       100     0.758144        1.03179
1       47      0.758144        0.96371
2       54      0.727386        0.912661
3       60      0.705355        0.857932
4       61      0.657006        0.811883
5       58      0.657006        0.769599
6       47      0.595538        0.736513
7       53      0.595538        0.718777
8       66      0.595538        0.696454
9       49      0.561895        0.670669
10      53      0.514064        0.650457
11      59      0.505097        0.616604
12      55      0.505097        0.592464
13      60      0.497067        0.572719
14      48      0.467229        0.557426
15      58      0.44677         0.542682
16      57      0.44677         0.538783
17      53      0.44441         0.519375
18      59      0.44441         0.512814
19      47      0.431716        0.496614
20      51      0.431798        0.492885
21      54      0.431798        0.484241
22      55      0.431798        0.481492
23      52      0.402477        0.473635
24      58      0.395013        0.464906
25      58      0.395979        0.460759
26      62      0.395979        0.455918
27      56      0.373773        0.441825
28      51      0.377606        0.442353
29      64      0.373032        0.440682
30      59      0.373032        0.436511
31      65      0.368091        0.428604
32      61      0.348563        0.427526
33      52      0.348563        0.403681
34      61      0.348563        0.403101
35      65      0.33754         0.403952
36      58      0.341745        0.399188
37      67      0.304036        0.398323
38      64      0.304036        0.394692
39      58      0.294635        0.381294
40      58      0.293038        0.379477
41      55      0.285197        0.375154
42      64      0.285197        0.374484
43      58      0.280692        0.360436
44      63      0.276096        0.356769
45      63      0.276096        0.344396
46      57      0.276096        0.343845
47      58      0.270799        0.328424
48      54      0.240293        0.323965
49      55      0.240293        0.331106
50      61      0.234587        0.324565
51      53      0.234587        0.314767
52      52      0.234587        0.319126
53      44      0.234587        0.313707
54      59      0.234587        0.314518
55      47      0.234587        0.298974
```

```
# print best solution found:
```

```
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
minFitnessValues_GA_1042_CR0p4, meanFitnessValues_GA_1042_CR0p4 = logbook.select("
print('History of minFitnessValues_GA =',minFitnessValues_GA_1042_CR0p4)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p4)
```

```
Best solution:  [2, 2, 1, 0, 2, 2, 0, 2, 0, 1, 2, 0, 1, 2, 1, 2, 2, 1, 2, 0,
Best FitnessMin = 0.03568
History of minFitnessValues_GA = [0.7581441108997677, 0.7581441108997677, 0.7
History of meanFitnessValues_GA = [1.031790151391004, 0.963710243402782, 0.91
```

P_CROSSOVER = 0.8

```
P_CROSSOVER = 0.8  # probability for crossover
```

```
# Set the random seed
# for reprodicibility of results:
RANDOM_SEED = 1042
random.seed(RANDOM_SEED)

# Create the instance of the MountainCar class:
car = MountainCar(RANDOM_SEED)
```

```
# create initial population (generation 0):
population = toolbox.populationCreator(n=POPULATION_SIZE)

# prepare the statistics object:
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", numpy.min)
stats.register("avg", numpy.mean)

# define the hall-of-fame object:
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)

print('**************************')
start = time.time()
# perform the Genetic Algorithm flow with hof feature added:
population, logbook = eaSimpleWithElitism(population,
                                           toolbox,
                                           cxpb=P_CROSSOVER,
                                           mutpb=P_MUTATION,
                                           ngen=MAX_GENERATIONS,
                                           stats=stats,
                                           halloffame=hof,
```

```
                                                                  verbose=True)
end = time.time()
time_1042_CR0p8 = end - start
print("Time Elapsed = ", time_1042_CR0p8)
```

```
***************************
gen     nevals  min             avg
0       100     0.758144        1.03179
1       68      0.758144        0.966299
2       70      0.752055        0.910033
3       69      0.691596        0.8756
4       70      0.691596        0.831427
5       78      0.608389        0.796644
6       73      0.608389        0.744301
7       72      0.537531        0.699397
8       74      0.513783        0.662102
9       74      0.480739        0.639031
10      71      0.480739        0.603637
11      77      0.480739        0.591893
12      68      0.480739        0.571104
13      75      0.464238        0.55785
14      72      0.399844        0.541363
15      72      0.399844        0.525015
16      75      0.399844        0.502956
17      77      0.393905        0.496615
18      74      0.377284        0.47752
19      72      0.377284        0.46305
20      69      0.370618        0.443473
21      69      0.339829        0.429576
22      76      0.319831        0.424963
23      75      0.319831        0.412669
24      69      0.290862        0.404008
25      74      0.312944        0.388373
26      76      0.254349        0.37347
27      71      0.254349        0.367373
28      67      0.254349        0.358663
29      71      0.254349        0.349505
30      72      0.248498        0.337288
31      74      0.246287        0.328025
32      71      0.243479        0.321194
33      79      0.191121        0.311822
34      70      0.167603        0.298414
35      76      0.167603        0.287305
36      74      0.161337        0.271923
37      76      0.137685        0.249659
38      77      0.116808        0.242038
39      68      0.116808        0.237632
40      69      0.0931239       0.222499
41      72      0.0931239       0.221872
42      76      0.0931239       0.222749
43      73      0.0774572       0.209579
44      69      0.0774572       0.203161
45      72      0.0774572       0.205266
46      70      0.0397077       0.195229
47      72      0.0397077       0.195742
48      70      0.0397077       0.18765
49      69      0.0397077       0.18707
50      73      0.0397077       0.183028
51      71      0.0397077       0.179607
```

```
52      75      0.0397077       0.17581
53      70      0.033631        0.160689
54      72      0.033631        0.164106
```

```
# print best solution found:
best = hof.items[0]
print("Best solution: ", best)
print("Best FitnessMin = %1.5f" % best.fitness.values[0])
#print("Best Fitness = ", best.fitness.values[0])

# extract statistics:
minFitnessValues_GA_1042_CR0p8, meanFitnessValues_GA_1042_CR0p8 = logbook.select("
print('History of minFitnessValues_GA =',minFitnessValues_GA_1042_CR0p8)
print('History of meanFitnessValues_GA =',meanFitnessValues_GA_1042_CR0p8)
```

```
Best solution:  [1, 1, 0, 1, 2, 2, 1, 2, 1, 1, 1, 0, 2, 1, 2, 2, 1, 1, 2, 2,
Best FitnessMin = -0.00500
History of minFitnessValues_GA = [0.7581441108997677, 0.7581441108997677, 0.7
History of meanFitnessValues_GA = [1.031790151391004, 0.9662991740375603, 0.9
```

RESUME

Again ... for various P_CROSSOVER we can obtain **different**:

- **solutions** :) ... of course,
- **performance** (fitness function value),
- **history**.

The reasons are

- the stochastic manner of parameter change during evolution,
- BUT ... more important ... different levels of gene exchange.

... with various GA parameters ... like Mutation Probability

(let's try it as a self-guided learning!)

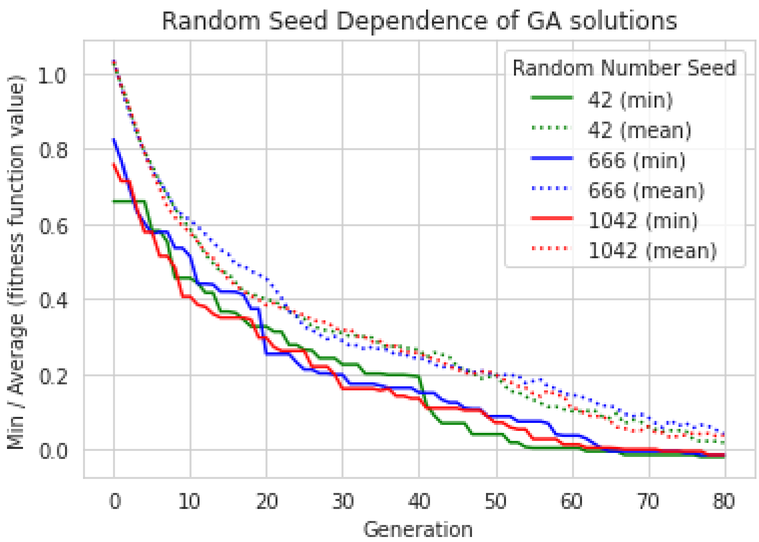
It takes a small change in *P_MUTATION* variable.

Comparison Plots

Random Seed Dependence

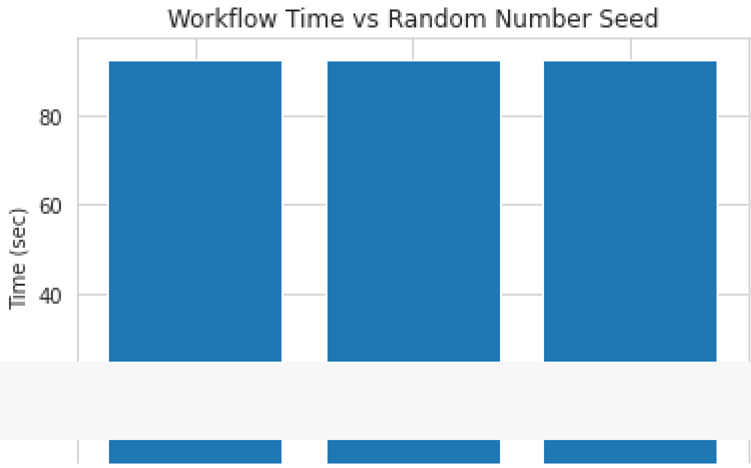
Fitness Function

```
sns.set_style("whitegrid")

# RS=42
plt.plot(minFitnessValues_GA_42, color='green', label='42 (min)')
plt.plot(meanFitnessValues_GA_42, color='green', linestyle = 'dotted', label='42 (

# RS=666
plt.plot(minFitnessValues_GA_666, color='blue', label='666 (min)')
plt.plot(meanFitnessValues_GA_666, color='blue', linestyle = 'dotted', label='666

# RS=1042
plt.plot(minFitnessValues_GA_1042, color='red', label='1042 (min)')
plt.plot(meanFitnessValues_GA_1042, color='red', linestyle = 'dotted', label='1042

plt.xlabel('Generation')
plt.ylabel('Min / Average (fitness function value)')
plt.title('Random Seed Dependence of GA solutions')
plt.legend(title='Random Number Seed')
plt.show()
```



Time

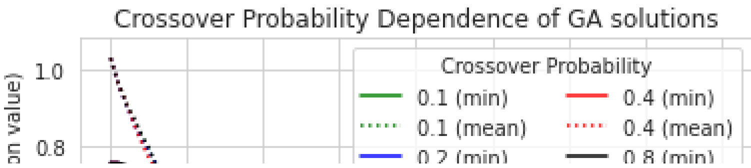
```
import matplotlib.pyplot as plt

x = ['42','666','1042']
y = [time_42,time_666,time_1042]
plt.bar(x,y)
plt.ylabel('Time (sec)')
plt.title('Workflow Time vs Random Number Seed')
plt.show()
```



Crossover Probability Dependence

Fitness Function

```
sns.set_style("whitegrid")

# CR=0.1
plt.plot(minFitnessValues_GA_1042_CR0p1, color='green', label='0.1 (min)')
plt.plot(meanFitnessValues_GA_1042_CR0p1, color='green', linestyle = 'dotted', lab

# CR=0.2
plt.plot(minFitnessValues_GA_1042_CR0p2, color='blue', label='0.2 (min)')
plt.plot(meanFitnessValues_GA_1042_CR0p2, color='blue', linestyle = 'dotted', labe

# CR=0.4
plt.plot(minFitnessValues_GA_1042_CR0p4, color='red', label='0.4 (min)')
plt.plot(meanFitnessValues_GA_1042_CR0p4, color='red', linestyle = 'dotted', label

# CR=0.8
plt.plot(minFitnessValues_GA_1042_CR0p8, color='black', label='0.8 (min)')
plt.plot(meanFitnessValues_GA_1042_CR0p8, color='black', linestyle = 'dotted', lab

plt.xlabel('Generation')
plt.ylabel('Min / Average (fitness function value)')
plt.title('Crossover Probability Dependence of GA solutions')
plt.legend(title='Crossover Probability', ncol=2)
plt.show()
```



Time

```
import matplotlib.pyplot as plt

x = ['0.1','0.2','0.4','0.8']
y = [time_1042_CR0p1,time_1042_CR0p2,time_1042_CR0p4,time_1042_CR0p8]
plt.bar(x,y)
plt.ylabel('Time (sec)')
plt.title('Workflow Time vs Crossover Probability')
plt.show()
```