# Requirement Analysis

ATSE - Assignment 3

Vlady Veselinov - University of Strathclyde

## 1. Introduction

There are many possibilities in the situations where significance of data can be quantified. Using fitness functions that transform data from multiple sources into meaningful parameters opens a pathway to using algorithms that can find solutions, optimising parameters to fit the needs of users. When there is more than one parameter, the process of optimisation becomes less trivial to implement because there are different trends to keep track of. A simple example can be minimising one number and maximising another. This report comes with a complementary GitHub repository that solves a multi-objective optimisation problem.

## 2. The Problem

This report covers the well-known Next Release problem, where there are requirements and customers. Each requirement has a cost and each customer is valuable to a different extent. If a customer cares about a particular set of requirements, a score can be assigned to said requirements for each customer, reflecting their overall value. In this case a dataset is provided, containing requirement costs, customer's requirements and weight. If each requirement is ordered by importance according to the customer, that data can be used to provide another metric that will factor into the final score formula.

## 3. DEAP: A Python Evolutionary Computation Framework

DEAP stands for Distributed Evolutionary Algorithms in Python. It's provides a useful framework for solving a wide variety of evolutionary problems by not limiting the user with predefined types of data. While this is beneficial in the long run, the library can initially seem like a black box. There is overhead in adoption for less-experienced users. Example code becomes harder to understand without having read the documentation. That's not necessarily a big problem as it is very thorough and provides clear explanation coupled with examples of every feature it has to offer, although it takes some time to get used to.

### a. Types

As with any evolutionary problem, it's important to initially select appropriate types to represent a solution. This can be thought of as choosing how a chromosome will look (binary string, list of floating point numbers, etc.) and how its fitness will be represented. Will it have more than one value for fitness, what will the library be doing to each fitness value, will it be maximising or minimising? These questions can be answered in just a couple of lines. The ***creator*** module is used for making types like this:

```
from deap import base, creator

creator.create('FitnessMin', base.Fitness, weights = (-1.0,))
creator.create('Individual', list, fitness = creator.FitnessMin)
```

These 3 lines are creating a ***FitnessMin*** class that will have one value in it. will aim to minimise it (hence the minus sign in the *weights* param). In the third line, an ***Individual*** is created that will be a ***list*** of things and will have a fitness type of ***FitnessMin***. More information on types can be found here.

### b. Initialisation

The created types need to be populated with something. The library provides a **Toolbox** class that contains many useful utilities, some of which allow initialisation. The following snippet demonstrates the most important and commonly used method in DEAP: **register**.

```
import random
from deap import tools

toolbox = base.Toolbox()
toolbox.register('attribute', random.random)
```

It lives in an initialised toolbox creates a method that can be called from it. In the above example, register makes a method with the name ***attribute.*** The first argument is the name, the second is the function that will be called whenever ***toolbox.attribute*** is called.

```
toolbox.register(
    'individual',
    tools.initRepeat,
    creator.Individual,
    toolbox.attribute,

    # Length of Individual
    n = 10
)

toolbox.register(
    'population',
    tools.initRepeat,
    list,
    toolbox.individual
)
```

The upper snippet first creates method that will make an ***individual*** that will be made up from ***attribute***s (in this case: random floating point numbers) and will have a length of ***n***. That ***individual*** is then used inside a method that can create a ***population*** which will be a list of ***individual***s. These snippets' purpose is to create the actual methods, which can later be called to initialise populations and individuals, for example:

```
population = toolbox.population()
```

### c. Operators

DEAP provides a series of useful tools that are used to do common tasks such as selection, crossover and mutation. They are already defined, they just need to be registered in the toolbox in order to be available for use.

```
toolbox.register('mate', tools.cxTwoPoint)
toolbox.register(
    'mutate',
    tools.mutGaussian,
    mu = 0,
    sigma = 1,
    indpb = 0.1
)
toolbox.register('select', tools.selTournament, tournsize = 3)
```

The purpose of register becomes easier to grasp in this example because of how obvious the arguments are:

1. Name of method being registered
2. Function that will be called when *toolbox.nameOfMethod* is called
3. Arguments that will be passed to said function

In other words, the definition from the documentation:

**register**(*alias*, *method*[, *argument*[, ...]])

There is one thing left to be registered. The fitness function. DEAP allows the user to implement their own:

```
def evaluate(individual):
    return sum(individual),

toolbox.register('evaluate', evaluate)
```

Note the comma after *return*. DEAP will expect any fitness function to return a collection of values, even if the algorithm has one fitness value. (docs: "fitness values must be iterable"). Collections are used in Python when returning or assigning multiple things.

### d. Algorithms

The library also relieves effort by providing multiple evolutionary algorithms to choose from. The simplest example is, unsurprisingly, **eaSimple**. It will return the last generation and a logbook that can contain useful information such as statistics.

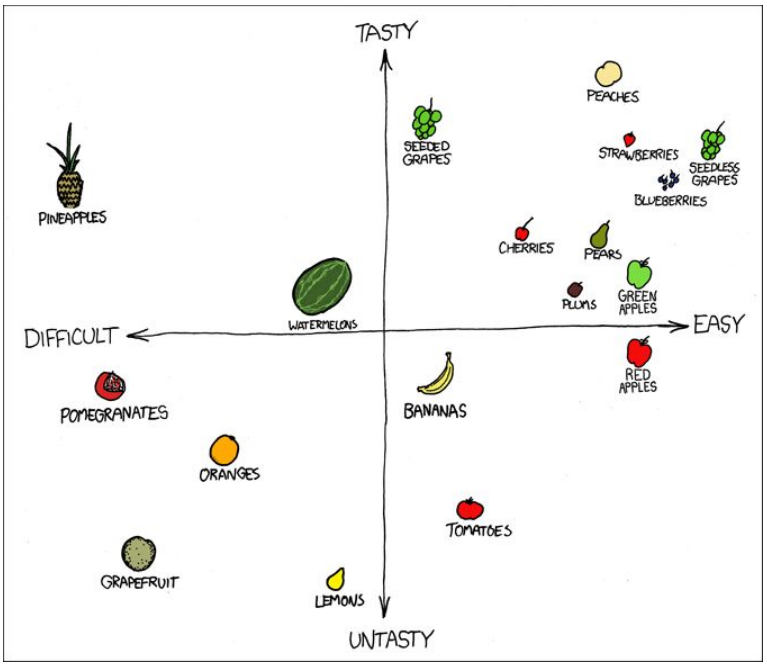
```
from deap import algorithms

population = toolbox.population(n = 50)
population = toolbox.select(population, len(population))

lastGeneration, logbook = algorithms.eaSimple(
    population,
    toolbox,

    # Crossover Probability
    cxpb = 0.9,

    # Mutation Probability
    mutpb = 0.05,

    # Number of Generations
    ngen = 10
)
```

In roughly 20 lines of code, DEAP allows for a generic evolutionary algorithm to be ran without much hassle. To summarise, all it takes for a simple example is to:

- Create types
- Register initialisation, selection, crossover, mutation and fitness functions in a toolbox
- Import an algorithm, pass it an initial population, the toolbox, some params and run it

## 4. Multi-Objective Optimisation

In reality, there are problems that require the consideration of more than one value. Areas such as economics, engineering and finance can all benefit from a solution that can solve these problems. For example, a bank might care about a problem connected with unemployment and inflation. Finance firms might want to minimise risk and maximise returns. Engineers might look for low cost and high efficiency. It's possible to combine these values into one mathematically, but that will bias the results. It will also be more complicated to introduce more than two parameters. Hence, a method for multi-objective optimisation is needed, this report focuses on exploring the Non Sorting Genetic Algorithm (NSGA-2). In order to implement it, a few concepts must be introduced:




a. Pareto Optimality - A set where no element is worse-off than the others

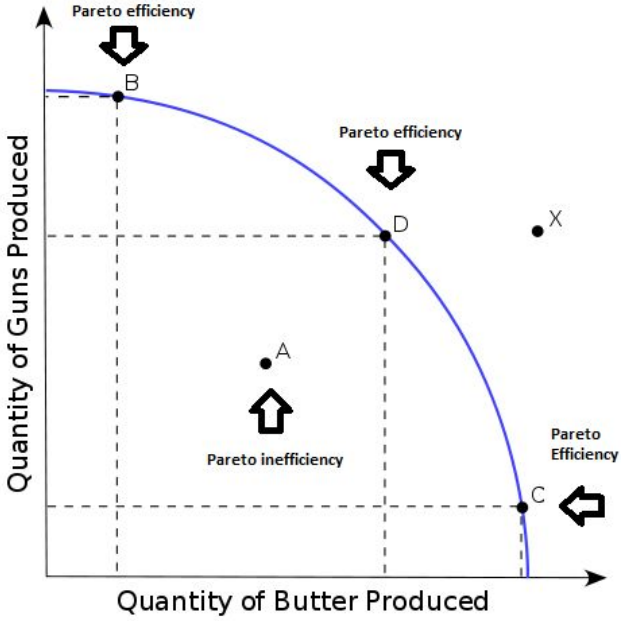



Image from [Economics Wiki]

b. Non-domination Sorting - algorithm that produces the Pareto Optimality (Deb et al., 2002)

c. [Crowding Distance] - A method for finding more varied (further apart) individuals (Deb et al., 2002)

# 5. Practical Example Using DEAP

This example covers the scenario of a software company deciding what requirements to build for what customers. If each requirement has a cost and each customer has a list of desired requirements, ordered by importance. The goal of the company would be to minimise cost and maximise the value of each requirement for, each customer, based on how important the customer is.

A score can be introduced that will combine the value of the requirement with the importance of the customer.

score(*feature*) = weight(*customer*) * value(*feature*)

The results of that formula for each customer summed would give the total score of a feature. If there is a vector of costs and a vector of scores for all features, the NSGA-2 algorithm can be utilised to find sets of optimal solutions, using the two vectors to calculate cost and score for each solution.

## a. Parsing Data, Score and Cost

Using the example files, two nrp datasets are parsed (one classic and one realistic), ignoring levels and dependencies, only focusing on requirement costs, customers and their requirements. The data is shaped into a dictionary like this:

```
'''
data = {
    textFileName: {
        requirementCosts: number[],
        customers: [
            { weight: float, requirements: number[] },
            ...
        ]
    }
}
'''
```

Where customer weight is calculated by getting profit divided by total sum of profits for all customers. After the data is parsed, the score and cost vectors are calculated taking into account the order of requirements each customer has. If score = weight * value, the value looks like this:

```
def value(requirement = -1, customer = { 'requirements': [] }):
    '''
    How valuable a requirement is to a customer.
    Returns a value between 0 and 1.

    1 for the first requirement,
    proportionally smaller to customer requirements length thereafter.
    '''
    requirements = customer['requirements']

    if requirement in requirements:
        length = len(requirements)

        return (length - requirements.index(requirement)) / length

    return 0.0
```

The above function can be used to get the score of a requirement for each customer like this:

```
def score(
    requirement = -1,
    customers = [{ 'weight': 0.0, 'requirements': [] }]
):
    '''
    The sum of scores of a requirement for each customer.
    '''
    score = 0

    for customer in customers:
        score += customer['weight'] * value(requirement, customer)

    return score
```

This function can be called for every requirement, for the sake of brevity, its definition is here.

### b. Creating the NSGA-2 toolbox

Much like the simple GA in point 3, DEAP requires us to create a toolbox that will contain all the methods necessary to run the evolutionary algorithm. The score and cost vectors need to be calculated first.

```
dataSet = data['nrp-e3']

scoreVector = getScoreVector(dataSet)
```

```
costVector = dataSet['requirementCosts']
```

The creation of the toolbox:

```
# NSGA2 toolbox

# Create a fitness Type,
# maximise first (score), minimise second (cost) element
creator.create('FitnessMaxMin', base.Fitness, weights = (1.0, -1.0))

# Create an Individual type that will have the fitness declared above
creator.create('Individual', list, fitness = creator.FitnessMaxMin)

nsgaToolbox = base.Toolbox()

# Make functions that will be called later,
# first argument is name of the function,
# second is the actual function being called,
# all arguments after are passed to function being called

# Each solution will be made up of booleans
nsgaToolbox.register('attr_bool', randint, 0, 1)

# and will be 'n' elements long
nsgaToolbox.register(
    'individual',
    tools.initRepeat,
    creator.Individual,
    nsgaToolbox.attr_bool,
    n = len(costVector)
)

nsgaToolbox.register(
    'population',
    tools.initRepeat,
    list,
    nsgaToolbox.individual
)
```

Fitness, crossover, mutation and selection follow after:

```
def getFitness(requirementVec = [], scoreVec = scoreVector, costVec =
costVector):
    score = numpy.dot(scoreVec, requirementVec)
    cost = numpy.dot(costVec, requirementVec)

    return score, cost
```

Using standard tools, NSGA-2 selection is handed to the user in one line.

```
nsgaToolbox.register('select', tools.selNSGA2)
```

### c. Creating a Single Objective Toolbox

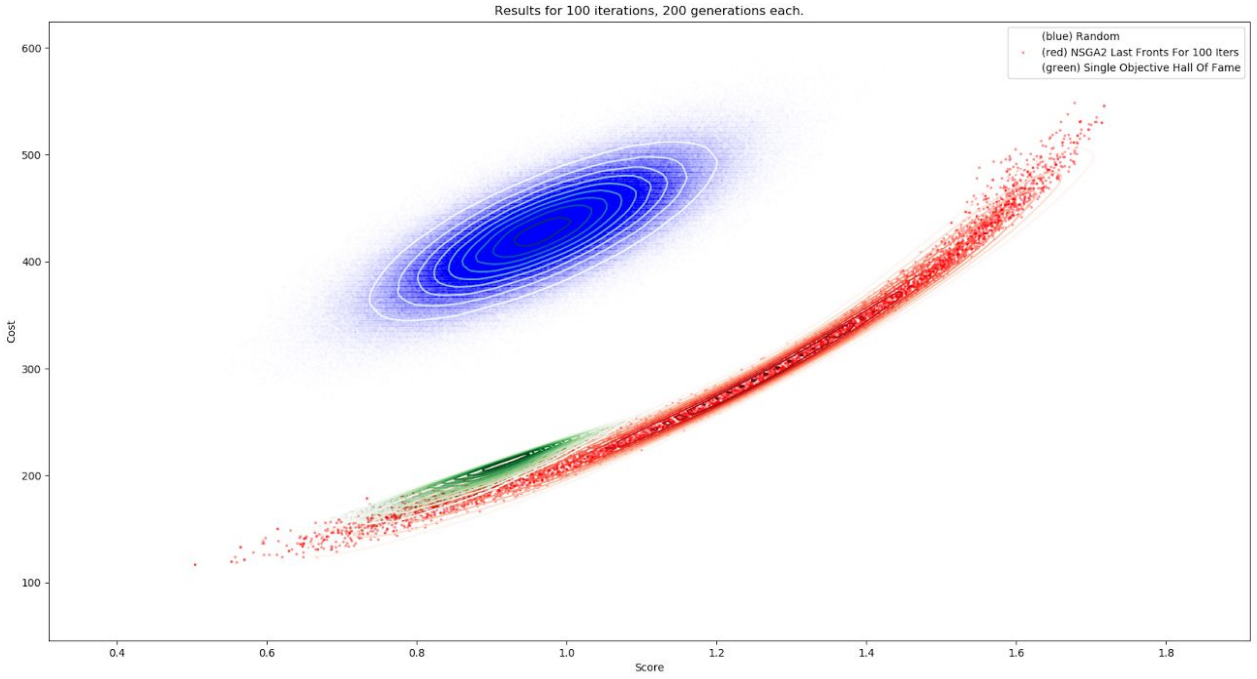
The only different aspect of this toolbox is the fitness function and fitness type. Instead of using a collection of score and cost. It uses a collection of a single value. Score divided by cost.

## 6. Analysis and Plots

Two datasets are selected: “nrp1” and “nrp-e3”. The following results show an animation of NSGA-2 pareto fronts for each generation

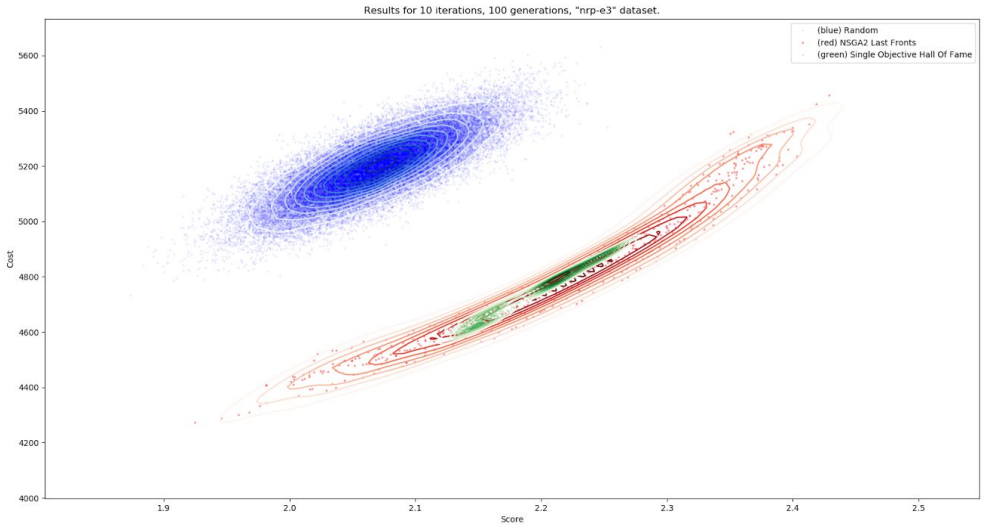
- Animation nrp1
- Animation nrp-e3

The algorithm is trying to provide a set of solutions minimising cost and maximising score. The animations clearly show that this is indeed the trend. To have a better idea of how NSGA-2 performs, it’s compared (nrp1) to random search and single-objective optimisation.




Blue is random, green are the best individuals single objective, red are the last fronts of multi-objective. The random individuals are clearly behind both multi and single objective when it comes to cost vs. score. It’s interesting to see how the single objective results are focused in a small area. Where the cost isn’t too high, but score is also quite low. Since the

fitness function returns score / cost, single objective would be useful in a scenario where the software company was only interested in solutions where they get the most value for money. However, not all companies care about value for money, sometimes a company can win more long-term benefits if requirements with higher score are developed. In other words, some people might be interested in more expensive solutions, where the score is higher than a score threshold that can't be surpassed by spending less. This practically means that NSGA-2 is good at identifying solutions on distant spectrums and can provide a wider variety of solutions. Here's how the same type of plot looks for the "nrp-e3" dataset:




It follows similar tendencies and confirms that NSGA-2 results in a better variety. In conclusion, single objective is good when the goal is very clear, NSGA-2 is superior when the problem is more complex and has more unfamiliar factors to keep track of. The GitHub repo for this report contains instructions on how to run the examples.

# References


Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2), pp.182-197.

Fortin, F. and De Rainville, F. (2017). *DEAP*. [online] GitHub. Available at: https://github.com/DEAP/deap [Accessed 5 Nov. 2017].

Martí, L. (2017). *Evolutionary Computation Course*. [online] GitHub. Available at: https://github.com/lmarti/evolutionary-computation-course [Accessed 5 Nov. 2017].