# 10
# Genetic Algorithms

This chapter introduces the concept of **evolutionary computing**. Algorithms derived from the theory of evolution are particularly efficient in solving large combinatorial or **NP problems**. Evolutionary computing has been pioneered by **John Holland** [10:1] and **David Goldberg** [10:2]. Their findings should be of interest to anyone eager to learn about the foundation of **genetic algorithms** (**GA**) and **artificial life**.

This chapter covers the following topics:

- The origin of evolutionary computing
- The theoretical foundation of genetic algorithms
- Advantages and limitations of genetic algorithms

From a practical perspective, you will learn how to:

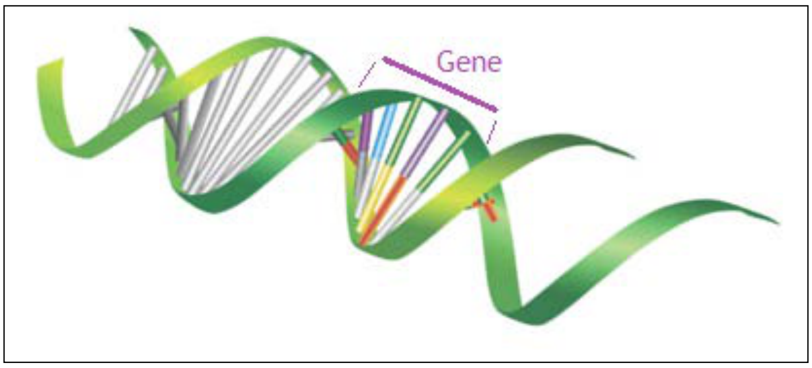
- Apply genetic algorithms to leverage technical analysis of market price and volume movement to predict future returns
- Evaluate or estimate the search space
- Encode solutions in the binary format using either hierarchical or flat addressing
- Tune some of the genetic operators
- Create and evaluate fitness functions

## Evolution

The **theory of evolution**, enunciated by Charles Darwin, describes the morphological adaptation of living organisms [10:3].

## The origin

The **Darwinian** process consists of optimizing the morphology of organisms to adapt to the harshest environments—hydrodynamic optimization for fishes, aerodynamic for birds, or stealth skills for predators. The following diagram shows a gene:



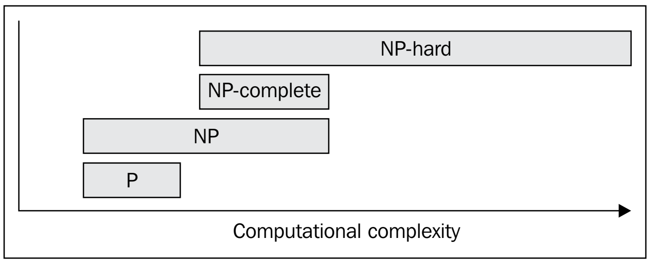
The **population** of organisms varies over time. The number of individuals within a population changes, sometimes dramatically. These variations are usually associated with the abundance or lack of predators and prey as well as the changing environment. Only the fittest organisms within the population can survive over time by adapting quickly to sudden changes in living environments and new constraints.

## NP problems

NP stands for nondeterministic polynomial time. The NP problems concept relates to the theory of computation and more precisely, time and space complexity. The categories of NP problems are as follows:

- **P-problems** (or P decision problems): For these problems, the resolution on a deterministic Turing machine (computer) takes a deterministic polynomial time.
- **NP problems**: These problems can be resolved in a polynomial time on nondeterministic machines.
- **NP-complete problems**: These are NP-hard problems that are reduced to NP problems for which the solution takes a deterministic polynomial time. These types of problems may be difficult to solve but their solution can be validated.

- **NP-hard problems**: These problems have solutions that may not be found in polynomial time.



Problems such as the traveling salesman, floor shop scheduling, the computation of a graph K-minimum spanning tree, map coloring, or cyclic ordering have a search execution time that is a nondeterministic polynomial, ranging from *n!* to $2^n$ for a population of *n* elements [10:4].

NP problems cannot always be solved using analytical methods because of the computation overhead—even in the case of a model, it relies on differentiable functions. Genetic algorithms were invented by John Holland in the 1970s, and they derived their properties from the Theory of Evolution of Darwin to tackle NP and NP-complete problems.

## Evolutionary computing

A living organism consists of cells that contain identical chromosomes. **Chromosomes** are strands of **DNA** and serve as a model for the whole organism. A chromosome consists of **genes** that are blocks of DNA and encode a specific protein.

**Recombination** (or crossover) is the first stage of reproduction. Genes from parents generate the whole new chromosome (**offspring**) that can be mutated. During mutation, one or more elements, also known as individual bases of the DNA strand or chromosomes, are changed. These changes are mainly caused by errors that occur when the genes from parents are being passed on to their offspring. The success of an organism in its life measures its fitness [10:5].

Genetic algorithms use reproduction to evolve a solution for a problem that is similar to unsupervised learning, for which a class or clusters are identified through an iterative or optimization methodology.

## Genetic algorithms and machine learning

The practical purpose of a genetic algorithm as an optimization technique is to solve problems by finding the most relevant or fittest solution among a set or group of solutions. Genetic algorithms have many applications in machine learning, as follows:

- **Discrete model parameters**: Genetic algorithms are particularly effective in finding the set of discrete parameters that maximizes the log likelihood. For example, the colorization of a black and white movie relies on a large but finite set of transformations from shades of grey to the RGB color scheme. The search space is composed of the different transformations and the objective function is the quality of the colorized version of the movie.
- **Reinforcement learning**: Systems that select the most appropriate rules or policies to match a given data set rely on genetic algorithms to evolve the set of rules over time. The search space or population is the set of candidate rules, and the objective function is the credit or reward for an action triggered by these rules (refer to the *Introduction* section of *Chapter 11, Reinforcement Learning*).
- **Neural network architecture**: A genetic algorithm drives the evaluation of different configurations of networks. The search space consists of different combinations of hidden layers and the size of those layers. The fitness or objective function is the sum of the squared errors.
- **Ensemble learning** [10:6]: A genetic algorithm can weed out the weak learners among a set of classifiers in order to improve the quality of the prediction.

## Genetic algorithm components

Genetic algorithms have the following three components:

- **Genetic encoding** (**and decoding**): This is the conversion of a solution candidate and its components into the binary format (an array of bits or a string of `0` and `1` characters)
- **Genetic operations**: This is the application of a set of operators to extract the best (most genetically fit) candidates (chromosomes)
- **Genetic fitness function**: This is the evaluation of the fittest candidate using an objective function

Encodings and the fitness function are problem dependent. Genetic operators are not.

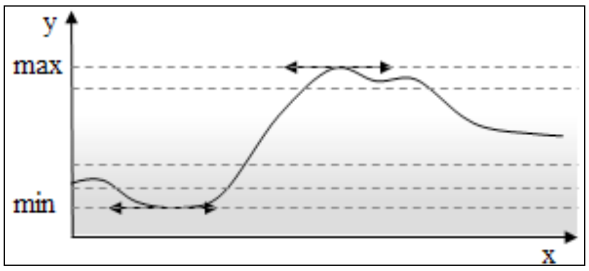
## Encodings

Let's consider the optimization problem in machine learning that consists of maximizing the log likelihood or minimizing the loss function. The goal is to compute the parameters or weights, $w=\{w_i\}$, that minimize or maximize a function $f(w)$. In the case of a nonlinear model, variables may depend on other variables, which make the optimization problem particularly challenging.

### Value encoding

The genetic algorithm manipulates variables as bits or bit strings. The conversion of a variable into a bit string is known as encoding. In the case where the variable is continuous, the conversion is known as **discretization**. Each type of variable has a unique encoding scheme, as follows:

- Boolean values are easily encoded with 1 bit: 0 for false and 1 for true.
- Continuous variables are discretized in a fashion similar to the conversion of an analog to a digital signal. Let's consider the function with a maximum **max** (similarly **min** for minimum) over a range of values, encoded with n=16 bits:

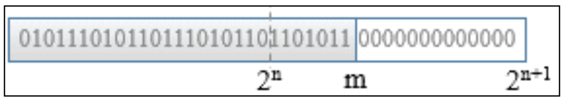


The step size of the discretization is computed as:

$$step = \frac{max\text{-}min}{2^n}$$

The step size of the discretization of the sine $y = sin(x)$ in 16 bits is 1.524e-5.

- Discrete or categorical variables are a bit more challenging to encode to bits. At a minimum, all the discrete values have to be accounted for. However, there is no guarantee that the number of variables will coincide with the bits boundary:



  In this case, the next exponent, *n+1*, defined the minimum number of bits required to represent the set of values: *n = log2(m).toInt + 1*. A discrete variable with 19 values requires 5 bits. The remaining bits are set to an arbitrary value (0, NaN,…) depending on the problem. This procedure is known as **padding**.

Encoding is as much art as it is science. For each encoding function, you need a decoding function to convert the bits representation back to actual values.

## Predicate encoding

A predicate for a variable *x* is a relation defined as *x operator [target]*, for instance, *unit cost < [9$]*, *temperature = [82F]*, or *Movie rating is [3 stars]*.

The simplest encoding scheme for predicates is as follows:

- Variables are encoded as category or type (for example, temperature, barometric pressure, and so on) because there is a finite number of variables in any model
- Operators are encoded as discrete type
- Values are encoded as either discrete or continuous values

**Encoding format for predicates**

There are many approaches for encoding a predicate in a bits string. For instance, the format *{operator, left-operand, right-operand}* is useful because it allows you to encode a binary tree. The entire rule, *IF predicate THEN action*, can be encoded with the action being represented as a discrete or categorical value.

## Solution encoding

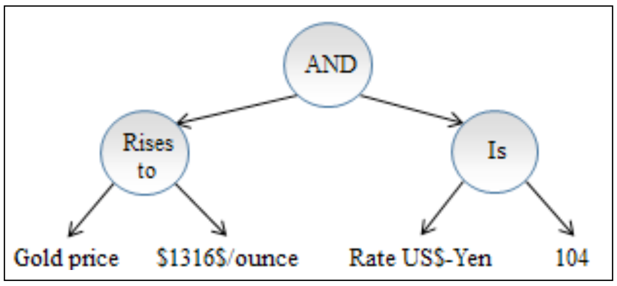
The solution encoding approach describes the solution to a problem as an unordered sequence of predicates. Let's consider the following rule:

```
IF {Gold price rises to [1316$/ounce]} AND
   {US$/Yen rate is [104]}).
THEN {S&P 500 index is [UP]}
```

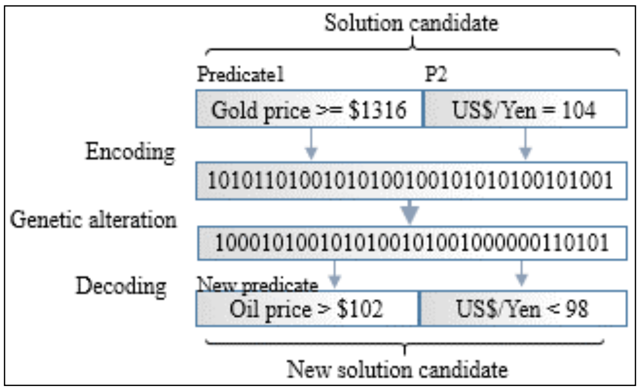
In this example, the search space is defined by two levels:

- Boolean operators (for example, AND) and predicates
- Each predicate is defined as a tuple *{variable, operator, target value}*

The tree representation for the search space is shown in the following diagram:



The bits string representation is decoded back to its original format for further computation:



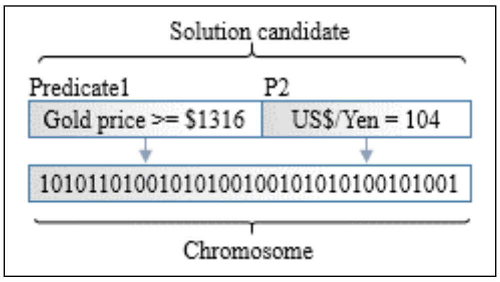
## The encoding scheme

There are two approaches to encode such a candidate solution or chain of predicates:

- Flat coding of a chromosome
- Hierarchical coding of a chromosome as a composition of genes

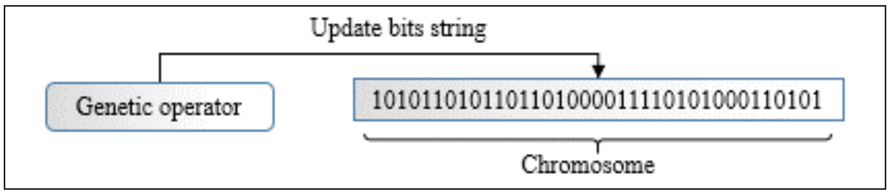
### Flat encoding

The flat encoding approach consists of encoding the set of predicates into a single chromosome (bits string) representing a specific solution candidate to the optimization problem. The identity of the predicates is not preserved:



An overview of flat addressing

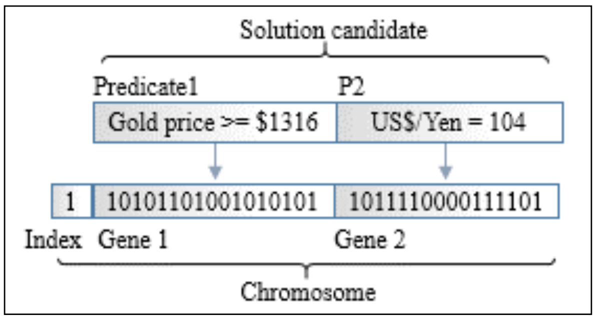
A genetic operator manipulates the bits of the chromosome regardless of whether the bits refer to a particular predicate:



Chromosome encoding with flat addressing

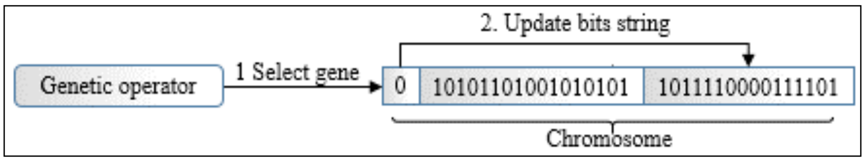
### Hierarchical encoding

In this configuration, the characteristic of each predicate is preserved during the encoding process. Each predicate is converted into a gene represented by a bit string. The genes are aggregated to form the chromosome. An extra field is added to the bits string or chromosome for the selection of the gene. This extra field consists of the index or the address of the gene:



An overview of hierarchical addressing

A generic operator selects the predicate it needs to manipulate first. Once the target gene is selected, the operator updates the bits string associated to the gene, as follows:



A chromosome with hierarchical addressing

The next step is to define the genetic operators that manipulate or update the bits string representing either a chromosome or individual genes.

## Genetic operators

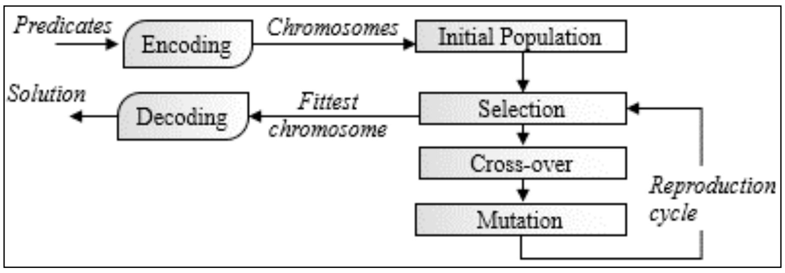
The implementation of the reproduction cycle attempts to replicate the natural reproduction process [10:7]. The reproduction cycle that controls the population of chromosomes consists of three genetic operators:

- **Selection**: This operator ranks chromosomes according to a fitness function or criteria. It eliminates the weakest or less-fit chromosomes and controls the population growth.
- **Crossover**: This operator pairs chromosomes to generate offspring chromosomes. These offspring chromosomes are added to the population along with their parent chromosomes.
- **Mutation**: This operator introduces minor alteration in the genetic code (bits string representation) to prevent the successive reproduction cycles from electing the same fittest chromosome. In optimization terms, this operator reduces the risk of the genetic algorithm converging quickly towards a local maximum or minimum.

**Transposition operator**

Some implementations of genetic algorithms use a fourth operator, genetic transposition, in case the fitness function cannot be very well defined and the initial population is very large. Although additional genetic operators could potentially reduce the odds of finding a local maximum or minimum, the inability to describe the fitness criteria or the search space is a sure sign that a genetic algorithm may not be the most suitable tool.

The following diagram gives an overview of the genetic algorithm workflow:



**Initialization**

The initialization of the search space (a set of potential solutions to a problem) in any optimization procedure is challenging, and genetic algorithms are no exception. In the absence of bias or heuristics, the reproduction initializes the population with randomly generated chromosomes. However, it is worth the effort to extract the characteristics of a population. Any well-founded bias introduced during initialization facilitates the convergence of the reproduction process.

Each of these genetic operators has at least one configurable parameter that has to be estimated and/or tuned. Moreover, you will likely need to experiment with different fitness functions and encoding schemes in order to increase your odds of finding a fittest solution (or chromosome).

## Selection

The purpose of the genetic selection phase is to evaluate, rank, and weed out the chromosomes (that is, the solution candidates) that are not a good fit for the problem. The selection procedure relies on a fitness function to score and rank candidate solutions through their chromosomal representation. It is a common practice to constrain the growth of the population of chromosomes by setting a limit to the size of the population.

There are several methodologies to implement the selection process from scaled relative fitness, Holland roulette wheel, and tournament selection to rank-based selection [10:8].

**Relative fitness degradation**

As the initial population of chromosomes evolves, the chromosomes tend to get more and more similar to each other. This phenomenon is a healthy sign that the population is actually converging. However, for some problems, you may need to scale or magnify the relative fitness to preserve a meaningful difference in the fitness score between the chromosomes [10:9].

The following implementation relies on rank-based selection using either a fitness or unfitness function to score chromosomes.

The selection process consists of the following steps:

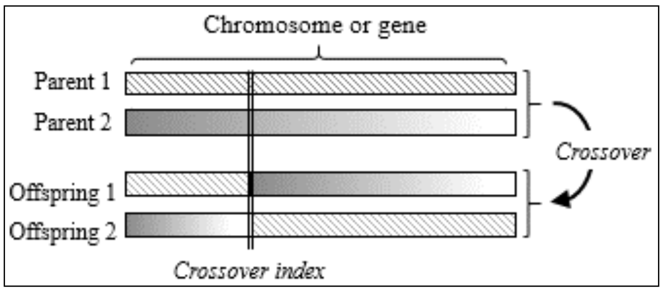
1. Apply the fitness/unfitness function to each chromosome *j* in the population, $f_j$
2. Compute the total fitness/unfitness score for the entire population, $\sum f_j$
3. Normalize the fitness/unfitness score of each chromosome by the sum of the fitness/unfitness scores of all the chromosomes, $f_j = f_j/\Sigma f_j$
4. Sort the chromosomes by their descending fitness score or ascending unfitness score
5. Compute the cumulative fitness/unfitness score for each chromosome, $j\, f_j = f_j + \sum f_k$
6. Generate the selection probability (for the rank-based formula) as a random value, $p\ \varepsilon\ [0,1]$
7. Eliminate the chromosome, *k*, having a low fitness score $f_k < p$ or high unfitness cost, $f_k > p$
8. Reduce the size of the population further if it exceeds the maximum allowed number of chromosomes.

**Natural selection**

You should not be surprised by the need to control the size of population of chromosomes. After all, nature does not allow any species to grow beyond a certain point in order to avoid depleting natural resources. The predator-prey process modeled by the **Lotka-Volterra equation** [10:10] keeps the population of each species in check.

## Crossover

The purpose of the genetic crossover is to expand the current population of chromosomes in order to intensify the competition among the solution candidates. The crossover phase consists of reprogramming chromosomes from one generation to the next. There are many different variations of crossover techniques. The algorithm for the evolution of the population of chromosomes is independent of the crossover technique. Therefore, the case study uses the simpler one-point crossover. The crossover swaps sections of the two-parent chromosomes to produce two offspring chromosomes, as illustrated in the following diagram:



A chromosome's crossover

An important element in the crossover phase is the selection and pairing of parent chromosomes. There are different approaches for selecting and pairing the parent chromosomes that are the most suitable for reproduction:

- Selecting only the *n* fittest chromosomes for reproduction
- Pairing chromosomes ordered by their fitness (or unfitness) value
- Pairing the fittest chromosome with the least-fit chromosome, the second fittest chromosome with the second least-fit chromosome, and so on

It is a common practice to rely on a specific optimization problem to select the most appropriate selection method as it is highly domain dependent.

The crossover phase that uses hierarchical addressing as the encoding scheme consists of the following steps:

1. Extract pairs of chromosomes from the population.
2. Generate a random probability $p \in [0,1]$.
3. Compute the index *ri* of the gene for which the crossover is applied as *ri* = *p.num_genes*, where *num_genes* are the number of genes in a chromosome.

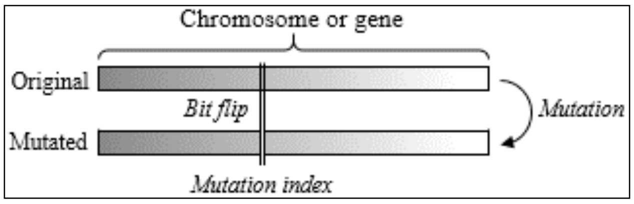
4. Compute the index of the bit in the selected gene for which the crossover is applied as *xi=p.gene_length*, where *gene_length* is the number of bits in the gene.
5. Generate two offspring chromosomes by interchanging strands between parents.
6. Add the two offspring chromosomes to the population.

**Preserving parent chromosomes**

You may wonder why the parents are not removed from the population once the offspring chromosomes are created. This is because there is no guarantee that any of the offspring chromosomes are a better fit.

## Mutation

The objective of genetic mutation is preventing the reproduction cycle from converging towards a local optimum by introducing a pseudo-random alteration to the genetic material. The mutation procedure inserts a small variation in a chromosome to maintain some level of diversity between generations. The methodology consists of flipping one bit in the bits string representation of the chromosome, as illustrated in the following diagram:



The chromosome mutation

The mutation is the simplest of the three phases in the reproduction process. In the case of hierarchical addressing, the steps are as follows:

1. Select the chromosome to be mutated.
2. Generate a random probability *p ϵ[0,1]*.
3. Compute the index *mi* of the gene to be mutated using the formula *mi = p.num_genes*.
4. Compute the index of the bit in the gene to be mutated *xi=p.genes_length*.
5. Perform a flip XOR operation on the selected bit.

**The tuning issue**

The tuning of a genetic algorithm can be a daunting task. A plan including a systematic design experiment for measuring the impact of the encoding, fitness function, crossover, and mutation ratio is necessary to avoid lengthy evaluation and self-doubt.

## Fitness score

The fitness function is the centerpiece of the selection process. There are three categories of fitness functions:

- **The fixed fitness function**: In this function, the computation of the fitness value does not vary during the reproduction process
- **The evolutionary fitness function**: In this function, the computation of the fitness value morphs between each selection according to predefined criteria
- **An approximate fitness function**: In this function, the fitness value cannot be computed directly using an analytical formula [10:11]

Our implementation of the genetic algorithm uses a fixed fitness function.

## Implementation

As mentioned earlier, the genetic operators are independent of the problem to be solved. Let's implement all the components of the reproduction cycle. The fitness function and the encoding scheme are highly domain specific.

In accordance with the principles of object-oriented programming, the software architecture defines the genetic operators using a top-down approach: starting with the population, then each chromosome, down to each gene.

## Software design

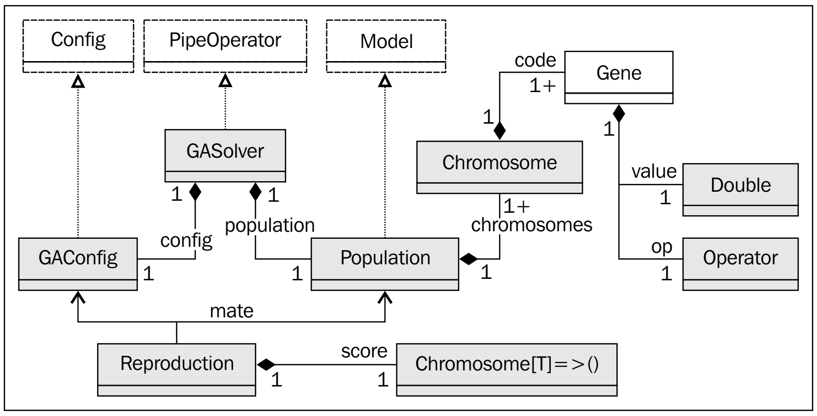
The implementation of the genetic algorithm uses a design that is similar to the template for classifiers (refer to the *Design template for classifier* section in *Appendix A*, *Basic Concepts*).

The key components of the implementation of the genetic algorithm are as follows:

- The `Population` class defines the current set of solution candidates or chromosomes.

- The `GASolver` class implements the GA solver and has two components: a configuration object of the type `GAConfig` and the initial population. This class defines a data transformation by implementing the `PipeOperator` trait.
- The configuration class `GAConfig` consists of the GA execution and reproduction configuration parameters.
- The reproduction (of the type `Reproduction`) controls the reproduction cycle between consecutive generations of chromosomes through the `mate` method.

The following UML class diagram describes the relation between the different components of the genetic algorithm:



UML class diagram of genetic algorithm components

Let's start by defining the key classes that control the genetic algorithm.

## Key components

The parameterized class `Population` (with the subtype `Gene`) contains the set or pool of chromosomes. A population contains chromosomes that are a sequence or list of element of the type inherited from `Gene`. A `Pool` is a mutable array in order to avoid excessive duplication of the `Chromosome` instances associated with immutable collections.

**A case for mutability**

It is a good Scala programming practice to stay away from mutable collections. However, in this case, the number of chromosomes can be very large. Most implementations of genetic algorithms update the population potentially three times per reproduction cycle, generating a large number of objects and taxing the Java garbage collector.

The `Population` class takes two parameters:

- `limit`: This is the maximum size of the population
- `chromosomes`: This is the pool of chromosomes defining the current population

A reproduction cycle executes the following sequence of three genetic operators on a population: `select` for selection across all the chromosomes of the population, `+-` for crossover of all the chromosomes, and `^` for the mutation of each chromosome. Consider the following code:

```
type Pool[T <: Gene] = ArrayBuffer[Chromosome[T]]
class Population[T <: Gene](limit: Int, val chromosomes: Pool[T]) {
   def select(score: Chromosome[T] => Unit, cutOff: Double)
   def +- (xOver: Double)
   def ^ (mu: Double)
   …
```

The `limit` value specifies the maximum size of the population during optimization. It defines the hard limit or constraints on the population growth.

The chromosome is the second level of containment in the genotype hierarchy. The `Chromosome` class takes a list of genes as parameter (`code`). The signature of the crossover and mutation methods, `+-` and `^`, are similar to their implementation in the `Population` class except for the fact that the crossover and mutable parameters are passed as indices relative to the list of genes and each gene. The section dedicated to the genetic crossover describes the `GeneticIndices` class:

```
class Chromosome[T <: Gene](val code: List[T]) {
  var unfitness: Double = 1e+5*(1 + Random.nextDouble)
  def +- (that: Chromosome[T], idx: GeneticIndices):
(Chromosome[T],Chromosome[T])
  def ^ (idx: GeneticIndices): Chromosome[T]
  …
```

The algorithm assigns the fitting score an `unfitness` value in this implementation to enable the ranking of the population and ultimately the selection of the fittest chromosomes.

**Fitness vs. unfitness**

The machine learning algorithms used the loss function or its variant as an objective function to be minimized. This implementation of the GA uses unfitness scores to be consistent with the concept of minimization of cost, loss, or penalty function.

Finally, the reproduction process executes the genetic operators on each gene:

```
class Gene(val id: String, val target: Double, op: Operator)(implicit
discr: Discretization) {
   val bits: BitSet
   …
   def +- (index: Int, that: Gene): Gene
   def ^ (index: Int): Unit
}
```

The `Gene` class takes four parameters:

- `id`: This is the identifier of the gene. It is usually the name of the variable represented by the gene.
- `target`: This is the target value or threshold to be converted or discretized into a bit string.
- `op`: This is the operator that is applied to the target value.
- `discr`: This is the discretization class that converts a double value to an integer to be converted into bits and vice versa.

The discretization is implemented as a case class:

```
case class Discretization(toInt: Double => Int,toDouble: Int =>
Double) {
  def this(R: Int) =
         this((x: Double) => (x*R).floor.toInt, (n: Int) => n/R)
}
```

The first function, `toInt`, converts a real value to an integer and `toDouble` converts the integer back to a real value. The discretization and inverse functions are encapsulated into a class to reduce the risk of inconsistency between the two opposite conversion functions.

The instantiation of a gene converts the predicate representation into a bit string (bits of the type `java.util.BitSet`) using the discretization function `Discretization.toInt`. The bit string is decoded by the `decode` method of the `Gene` companion object.

The `Operator` trait defines the signature of any operator. Each domain-specific problem requires a unique set of operations: Boolean, numeric, or string manipulation:

```
trait Operator {
  def id: Int
  def apply(id: Int): Operator
}
```

The preceding operator has two methods: an identifier `id` and an `apply` method that converts an index to an operator.

## Selection

The first genetic operator of the reproduction cycle is the selection process. The `select` method of the `Population` class implements the steps of the selection phase to the population of chromosomes in the most efficient manner, as follows:

```
def select(score: Chromosome[T] => Unit, cutOff: Double) = {
  val cumul = chromosomes.foldLeft(0.0)((s,x) =>{
            score(xy); s + xy.unfitness} ) //1
  chromosomes foreach( _ /= cumul) //2
  val newChromosomes = chromosomes.sortWith(_.unfitness < _.unfitness)
//3

  val cutOffSize = (cutOff*newChromosomes.size).floor.toInt //4
  val newPopSize = if(limit<cutOffSize) limit else cutOffSize //5
  chromosomes.clear //6
  chromosomes ++= newChromosomes.take(newPopSize) //7
}
```

The `select` method computes the cumulative sum of an unfitness value, `cumul`, for the entire population (line `1`). It normalizes the unfitness of each chromosome (line `2`), orders the population by decreasing value (line `3`), and applies a soft limit function on population growth, `cutOff` (line `4`). The next step reduces the size of the population to the lowest of the two limits: the hard limit, `limit`, or the soft limit, `cutOffSize` (line `5`). Finally, the current population is cleared (line `6`) and updated with the next generation (line `7`).

**Even population size**

The next phase in the reproduction cycle is the crossover, which requires the pairing of parent chromosomes. It makes sense to pad the population so that its size is an even integer.

The scoring function `score` takes a chromosome as parameter and updates its `unfitness` value for this chromosome.

## Controlling population growth

The natural selection process controls or manages the growth of the population of species. The genetic algorithm uses two mechanisms:

- The absolute maximum size of the population (hard limit).
- The incentive to reduce the population as the optimization progresses (soft limit). This incentive (or penalty) on the population growth is defined by the `cutOff` value used during selection (the `select` method).

The `cutoff` value is computed through a user-defined function, `softLimit`, of the type `Int => Double`, provided as a configuration parameter (`softLimit(cycle: Int) => a.cycle +b`).

## GA configuration

The four configurations and tuning parameters required by the genetic algorithm are:

- `xover`: This is the crossover ratio (or probability) and has a value in the interval [0, 1].
- `mu`: This is the mutation ratio with a value in the interval [0, 1].
- `maxCycles`: This is the maximum number of reproduction cycles.
- `softLimit`: This is the soft constraint on the population growth. The constraint function takes the number of iterations as argument and returns the maximum number of chromosomes allowed in the population.

Consider the following code:

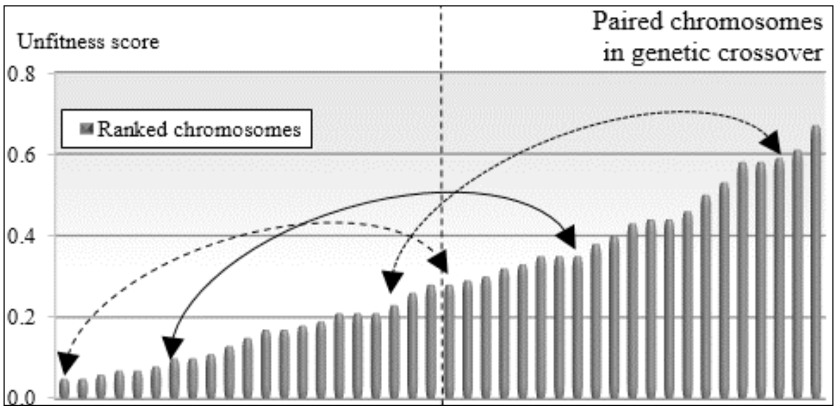
```
class GAConfig(val xover: Double,val mu: Double,val maxCycles: Int,val
softLimit: Int => Double) extends Config
```

## Crossover

As mentioned earlier, the genetic crossover operator couples two chromosomes to generate two offspring chromosomes that compete with all the other chromosomes in the population, including their own parents, in the selection phase of the next reproduction cycle.

## Population

We use the notation `+-` as the implementation of the crossover operator in Scala. There are several options to select pairs of chromosomes for crossover. This implementation ranks the chromosomes by their fitness value and then divides the population into two halves. Finally, it pairs the chromosomes of identical rank from each half as illustrated in the following diagram:



Pairing of chromosomes within a population prior to crossover

The crossover implementation, `+-`, selects the parent chromosome candidates for crossover using the pairing scheme described earlier. Consider the following code:

```
def +- (xOver: Double): Unit = {
  if( size > 1) {
    val mid = size>>1
    val bottom = chromosomes.slice(mid, size)  //1
    val gIdx = geneticIndices(xOver)  //5
    val offSprings = chromosomes.take(mid)
                        .zip(bottom) //2
                        .map(p => p._1 +-(p._2, gIdx))
                        .unzip //3
    chromosomes ++= offSprings._1 ++ offSprings._2 //4
  }
}
```

This method splits the population into two subpopulations of equal size (line `1`) and applies the Scala `zip` method (line `2`) to generate the set of pairs of offspring chromosomes (line `3`). The crossover operator, `+-`, is applied to each chromosome pair to produce an array of pairs of `offspring`. Finally, the crossover method adds offspring chromosomes to the existing population (line `4`). The crossover value, `xOver`, is a probability randomly generated over the interval `[config.xOver, 1]`.

The `geneticIndices` method (line `5`) computes the relative indices of the crossover bit in the chromosomes and genes:

```
case class GeneticIndices(val chOpIdx: Int, val geneOpIdx: Int)
def geneticIndices(prob: Double): GeneticIndices = {
   var idx = (prob*chromosomeSize).floor.toInt
   val chIdx = if(idx==0) 1
      else if(idx == chromosomeSize) chromosomeSize-1 else idx

   idx = (prob*geneSize).floor.toInt
   val gIdx = if(idx == 0) 1
     else if(idx == geneSize) geneSize-1 else idx
   GeneticIndices(chIdx, gIdx)
}
```

The `GeneticIndices` case class defines two indices of the bit whenever a crossover or a mutation occurs. The first index, `chOpIdx`, is the absolute index of the bit affected by the genetic operation in the chromosome. The second index, `geneOpIdx`, is the index of the bit within the gene subjected to crossover or mutation. The `geneticIndices` method of the `Population` class computes the two indices from a randomly generated value, `prob`, selected over the interval `[config.xover, 1]` for crossover and `[config.mu, 1]` for mutation.

## Chromosomes

First, we need to define the `Chromosome` class, which takes a list of genes, `code` (for genetic code), as the parameter:

```
class Chromosome[T <: Gene](val code: List[T])
```

The implementation of the crossover for a pair of chromosomes using hierarchical encoding follows two steps:

- Find the gene on each chromosome that corresponds to the crossover index, `gIdx.chOpIdx`, and then swap the remaining genes
- Split and splice the gene crossover at `xoverIdx`

Consider the following code:

```
def +-(that: Chromosome[T], gIdx: GeneticIndices): (Chromosome[T],
Chromosome[T]) = {
  val xoverIdx = gIdx.chOpIdx  //6
  val xGenes = spliceGene(gIdx, that.code(xoverIdx) ) //7

  val offSprng1 = code.slice(0, xoverIdx) ::: xGenes._1 :: that.code.
drop(xoverIdx+1) //8
```

```
    val offSprng2 = that.code.slice(0, xoverIdx) ::: xGenes._2 :: code.
  drop(xoverIdx+1)
    (Chromosome[T](offSprng1), Chromosome[T](offSprng2)//9
  }
```

The crossover method computes the index of the bit that defines the crossover (`xoverIdx`) in each parent chromosome (line `6`). The genes `this.code(xoverIdx)` and `that.code(xoverIdx)` are swapped and spliced by the `spliceGene` method to generate a spliced gene (line `7`).

```
  def spliceGene(gIdx: GeneticIndices, thatCode: T): (T, T) = {
    ((this.code(gIdx.chOpIdx) +- (thatCode, gIdx)),
     (thatCode +- (code(gIdx.chOpIdx), gIdx)) )
  }
```

The offspring chromosomes are gathered by collating the first `xOverIdx` genes of the parent chromosome, the crossover gene, and the remaining genes of the other parent (line `8`). The method returns the pair of offspring chromosomes (line `9`).

## Genes

The crossover is applied to a gene through the `+-`method of the `Gene` class. The exchange of bits between the two genes `this` and `that` uses the `BitSet` Java class to rearrange the bits after the permutation:

```
  def +- (that: Gene, idx: GeneticIndices): Gene = {
    val clonedBits = cloneBits(bits)  //10

    Range(gIdx.geneOpIdx, bits.size).foreach(n =>
      if( that.bits.get(n) ) clonedBits.set(n)
      else clonedBits.clear(n)
  ) //11

    val valOp = decode(clonedBits) //12
    Gene(id, valOp._1, valOp._2)
  }
```

The bits of the gene are cloned (line `10`) and then spliced by exchanging their bits along the crossover point `xOverIdx` (line `11`). The `cloneBits` function duplicates a bit string, which is then converted into a (target value, operator) tuple using the `decode` method (line `12`). We omit these two methods because they are not critical to the understanding of the algorithm.

## Mutation

The mutation of the population uses the same algorithmic approach as the crossover operation.

### Population

The mutation operator ^ invokes the same operator for all the chromosomes in the population and then adds the mutated chromosomes to the existing population, so that they can compete with the original chromosomes. We use the notation ^ to define the mutation operator to remind the reader that the mutation is implemented by flipping one bit:

```
def ^ (mu: Double): Unit =
  chromosomes ++= chromosomes.map(_ ^ geneticIndices(mu))
```

The mutation parameter `mu` is used to compute the absolute index of the mutating gene, `geneticIndices(mu)`.

### Chromosomes

The implementation of the mutation operator ^ on a chromosome consists of mutating the gene of the index `gIdx.chOpIdx` (line `1`) and then updating the list of genes in the chromosome (line `2`). The method returns a new chromosome (line `3`) that will compete with the original chromosome:

```
def ^ (gIdx: GeneticIndices): Chromosome[T] = { //1
  val mutated = code(gIdx.chOpIdx) ^ gIdx
  val xs = Range(0, code.size).map(i =>
    if(i==gIdx.chOpIdx) mutated  else code(i)).toList //2
  Chromosome[T](xs) //3
}
```

### Genes

Finally, the mutation operator flips (XOR) the bit at the index `gIdx.geneOpIdx`:

```
def ^ (gIdx: GeneticIndices): Gene = {
  val clonedBits = cloneBits(bits) //4
  clonedBits.flip(idx.geneOpIdx)  //5

  val valOp = decode(clonedBits)  //6
  Gene(id, valOp._1, valOp._2) //7
}
```

The ^ method mutates the cloned bit string, `clonedBits` (line `4`) by flipping the bit at the index `gIdx.geneOpIdx` (line `5`). It decodes and converts the mutated bit string by converting it into a (target value, operator) tuple (line `6`). The last step creates a new gene from the target-operator tuple (line `7`).

## The reproduction cycle

Let's wrap the reproduction cycle into a `Reproduction` class that uses the scoring function `score`:

```
class Reproduction[T <: Gene](score: Chromosome[T] => Unit)
```

The reproduction function, `mate`, implements the sequence or workflow of the three genetic operators: `select` for the selection, `+-` (`xover`) for the crossover, and ^ (`mu`) for the mutation:

```
def mate(population: Population[T], config: GAConfig, cycle: Int):
Boolean = population.size match {
  case 0 | 1 | 2=> false
  case _ => {
    population.select(score, config.softLimit(cycle))
    population +- (1.0 - Random.nextDouble*config.xover)
    population ^ (1.0 - Random.nextDouble*config.mu)
    true
  }
}
```

This method returns `true` if the size of the population is larger than 2. The last element of the puzzle is the exit condition. There are two options for estimating that the reproducing cycle is converging:

- **Greedy**: In this approach, the objective is to evaluate whether the *n* fittest chromosomes have not changed in the last *m* reproduction cycles
- **Loss function**: This approach is similar to the convergence criteria for the training of supervised learning

A simple exit condition describes the state, of the type `GAState`, of the genetic algorithm at each reproduction cycle:

```
def converge(population: Population[T], cycle: Int): GAState = {
  if(population == null) GA_FAILED
  else if(iters >= config.cycles)
          GA_NO_CONVERGENCE(s"failed after $cycle cycles")
  …
```

Let's define the state of the genetic algorithm as a case class of the super type GAState:

```
sealed abstract class GAState(val description: String)
case class GA_FAILED(val _description: String) extends GAState(_
description)
object GA_RUNNING extends GAState("Running")
case class GA_NO_CONVERGENCE(val _desc: String) extends GAState(_desc)
…
```

The last class GASolver manages the reproduction cycle and evaluates the exit condition or the convergence criteria:

```
class GASolver[T <: Gene](config: GAConfig, score: Chromosome[T]
=>Unit) extends PipeOperator[Population[T], Population[T]] {
   var state: GAState = GA_NOT_RUNNING
```

This class implements the data transformation |>, which transforms a population to another one, given a configuration, config and a scoring method, score, as follows:

```
def |> : PartialFunction[Population[T], Population[T]] = {
  case population: Population[T] if(population.size > 1) => {
    val reproduction = Reproduction[T](score)
    state = GA_RUNNING

    Range(0, config.maxCycles).find(n => {  //1
      reproduction.mate(population, config, n) match { //2
         case true => converge(population, n) != GA_RUNNING  //3
         case false => { …. }
      }
    }) match {
       case Some(n) => population
     …
```

The reproduction cycle is controlled by the find function (line 1) that tests whether an error occurs during the reproduction, mate (line 2), before the convergence criteria (line 3) are applied.

## GA for trading strategies

Let's apply our fresh expertise in genetic algorithms to evaluate different strategies to trade securities using trading signals. Knowledge in trading strategies is not required to understand the implementation of a GA. However, you may want to get familiar with the foundation and terminology of technical analysis of securities and financial markets, described briefly in the *Technical analysis* section in *Appendix A*, *Basic Concepts*.

*The problem is to find the best trading strategy to predict the increase or decrease of the price of a security given a set of trading signals.* A trading strategy is defined as a set of trading signals $ts_j$ that are triggered or fired when a variable $x = \{x_j\}$, derived from financial metrics such as the price of the security or the daily or weekly trading volume, either exceeds or equals or is below a predefined target value, $a_j$ (refer to the *Trading signals and strategy* section in *Appendix A, Basic Concepts*).

The number of variables that can be derived from price and volume can be very large. Even the most seasoned financial professionals face two challenges:

- Selecting a minimal set of trading signals that are relevant to a given data set (minimize a cost or unfitness function)
- Tuning those trading signals with heuristics derived from personal experience and expertise

**Alternative to GA**

The problem described earlier can certainly be solved using one of the machine learning algorithms introduced in the previous chapters. It is just a matter of defining a training set and formulating the problem as minimizing the loss function between the predictor and the training score.

The following table lists the trading classes with their counter part in the 'genetic world':

| **Generic classes** | **Corresponding securities trading classes** |
|---|---|
| `Operator` | `SOperator` |
| `Gene` | `Signal` |
| `Chromosome` | `Strategy` |
| `Population` | `StrategiesFactory` |

## Definition of trading strategies

A chromosome is the genetic encoding of a trading strategy. A factory class, `StrategyFactory`, assembles the components of a trading strategy: operators, unfitness function and signals.

## Trading operators

Let's extend the `Operator` trait with the `SOperator` class to define the operations we need to trigger the signals. The `SOperator` instance has a single parameter: its identifier, `_id`. The class overrides the `id` () method to retrieve the ID (similarly, the class overrides the `apply` method to convert an ID into an `SOperator` instance):

```
class SOperator(val _id: Int) extends Operator {
  override def id: Int = _id
  override def apply(idx: Int): SOperator = new SOperator(idx)
}
```

The operators used by trading signals are the logical operators: <, >, and =, as follows:

```
object LESS_THAN extends SOperator(1)
object GREATER_THAN extends SOperator(2)
…
```

Each operator is associated with a scoring function by the map `operatorFuncMap`. The function computes the unfitness of the signal against a real value or a time series:

```
val operatorFuncMap = Map[Operator, (Double, Double) =>Double](
  LESS_THAN -> ((x: Double, target: Double) => target - x),
  GREATER_THAN -> ((x: Double, target: Double) => x -target),
  … )
```

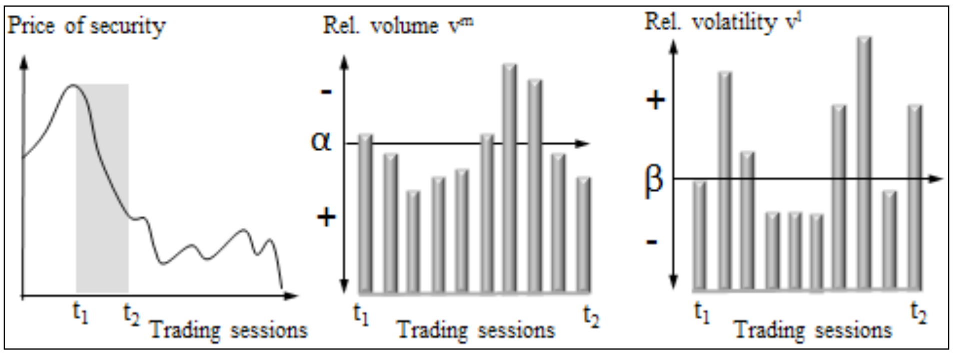
The `select` method of `Population` computes the unfitness value of a signal by quantifying the truthfulness of the predicate. For instance, the unfitness value for a trading signal, $x > 10$, is penalized as $5 - 10 = -5$ for $x = 5$ and credited as $14 - 10 = 4$ if $x = 14$. In this regard, the unfitness value is similar to the cost or loss in a discriminative machine learning algorithm.

## The cost/unfitness function

Let's consider the following trading strategy defined as a set of two signals to predict the sudden relative decrease $\Delta p$ of the price of a security:

- Relative volume $v^m$ with a condition $v^m < \alpha$
- Relative volatility $v^l$ with the condition $v^l > \beta$

Have a look at the following graphs:



As the goal is to model a sudden crash in stock price, we should reward the trading strategies that predict the steep decrease in the stock price and penalize the strategies that work well only with a small decrease or increase in stock price. For the case of the trading strategy with two signals, relative volume $v^m$ and relative volatility $v^l$, $n$ trading sessions, the cost or unfitness function `C`, and given a relative variation of stock price and a penalization $w = -\Delta p$:

$$w_t = -\Delta p_t$$
$$C\left(p, v^m, v^l \mid \alpha, \beta\right) = \sum_{t=0}^{n-1}\left(\alpha - v_t^m\right)w_t + \left(v_t^l - \beta\right)w_t$$

## Trading signals

Let's subclass the `Gene` class to define the trading signal:

```
class Signal(_id: String, _target: Double, _op: Operator,xt:
DblVector, weights: DblVector)(implicit discr: Discretization) extends
Gene(_id, _target, _op)
```

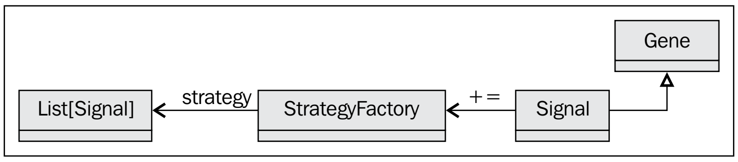
The `Signal` class takes the identifier for the feature, the `target` value, an operator `op`, the time series `xt` of the type `DblVector`, and the weights associated to each data point of the time series `xt`. The main purpose of the `Signal` class is to compute its score. The chromosome updates its unfitness by summing the score or weighted score of the signals it contains.

The score of the trading signal is simply the summation of the penalty or truthfulness of the signal for each entry of the time series, `ts`:

```
def score: Double = sumScore(operatorFuncMap.get(op).get)
def sumScore(f: (Double, Double) => Double): Double = xt.foldLeft(0.0)
((s, x) => s + f(x, target))
```

## Trading strategies

A trading strategy is an unordered list of trading signals. It makes sense to create a factory class to generate the trading strategies. The `StrategyFactory` class creates strategies of the type `List[Signal]` from an existing pool of signals of the subtype `Gene`:



The `StrategyFactory` class has two arguments: the number of signals, `nSignals`, in a trading strategy and the implicit `discretization` instance:

```
class StrategyFactory(nSignals: Int)(implicit discr: Discretization){
  val signals  = new ListBuffer[Signal]
  lazy val strategies: Pool[Signal]
   …
```

The `+=` method adds the trading signals to the factor. The `StrategyFactory` class generates all possible sequences of signals as trading strategies. The `+=` method takes five arguments: the identifier (`id`), `target`, operation (`op`) to qualify the class as a `Gene`, the times series `xt` for scoring the signals, and the `weights` associated to the overall cost function:

```
def += (id: String, target: Double, op: Operator, xt:
XTSeries[Double], weights: DblVector): Unit =
     signals.append(Signal(id, target, op, xt.toArray, weights) )
```

The `StrategyFactory` class defines `strategies` as `lazy` values to avoid unnecessary regeneration of the pool on demand:

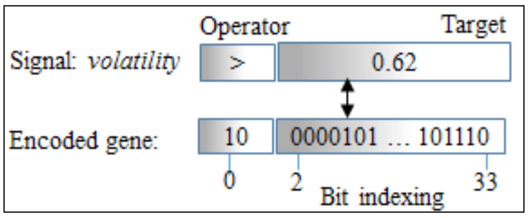
```
lazy val strategies: Pool[Signal] = {
  implicit val ordered = Signal.orderedSignals //7

  val xss = new Pool[Signal] //1
  val treeSet = new TreeSet[Signal] ++= signals.toList //2
  val subsetsIterator = treeSet.subsets(nSignals) //3
  while( subsetsIterator.hasNext) { //4
     val subset = subsetsIterator.next
     val signalList: List[Signal] = subset.toList //5
     xss.append(Chromosome[Signal](signalList)) //6
  } xss
}
```

The implementation of the `strategies` value creates `Pool` (line `1`) by converting the list of signals to a `treeset` (line `2`). It breaks down the tree set into unique subtrees of `nSignals` nodes each (line `3`). It instantiates a `subsetsIterator` iterator (line `3`) to traverse the sequence of subtrees (line `4`) and converts them into a list (line `5`) as arguments of the new chromosome (trading strategy) (line `6`). The procedure to order the signals, `orderedSignals`, in the tree set has to be implicitly defined (line `7`):

```
val orderedSignals = Ordering.by((signal: Signal) => signal.id)
```

## Signal encoding

The encoding of trading predicates is the most critical element of the genetic algorithm. In our example, we encode a predicate as a tuple (target value, operator). Let's consider the simple predicate *volatility > 0.62*. The discretization converts the value 0.62 into 32 bits for the instance and a 2-bit representation for the operator:



Encoding price volatility as a gene

**IEEE-732 encoding**

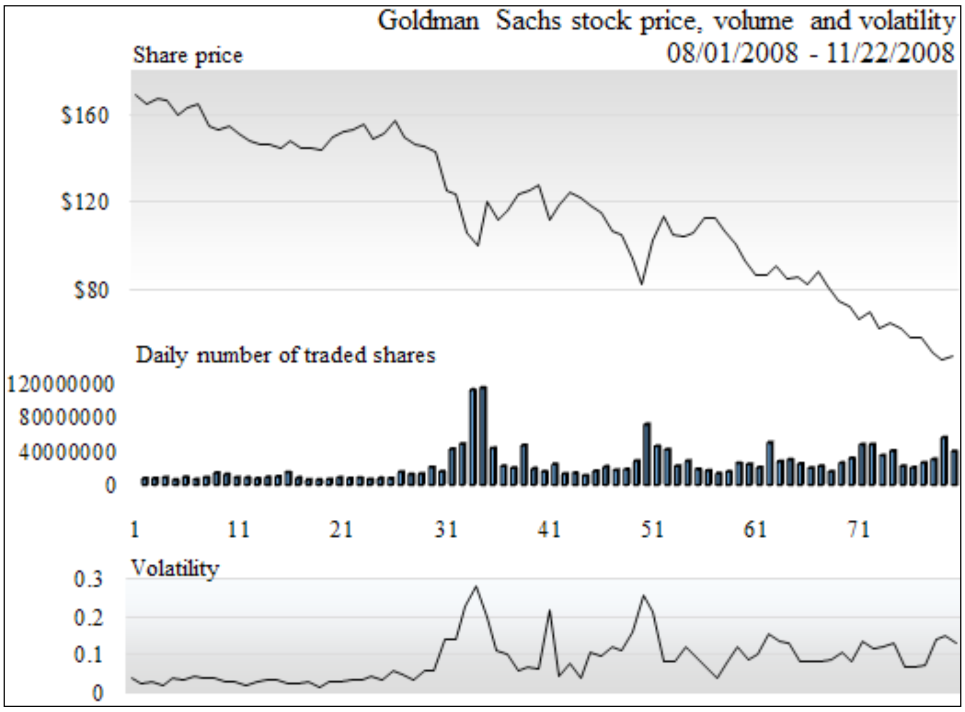
The threshold value for predicates is converted into an integer (the type `Int` or `Long`). The IEEE-732 binary representation of floating point values makes the bit addressing required to apply genetic operators quite challenging. A simple conversion consists of the following:

```
encoding e: (x: Double) => (x*100000).toInt
decoding d: (x: Int) => x*1e-5
```

All values are normalized; so, there is no risk of overflowing the 32-bit representation.

## Test case

The goal is to evaluate which trading strategy was the most relevant (fittest) during the crash of the stock market in fall 2008. Let's consider the stock price of one of the financial institutions, Goldman Sachs, as a proxy of the sudden market decline:



Goldman-Sachs fall 2008

Besides the variation of the price of the stock between two consecutive trading sessions (`deltaPrice`), the model uses the following parameters:

- `deltaVolume`: This is the relative variation of the volume between two consecutive trading sessions
- `deltaVolatility`: This is the relative variation of volatility between two consecutive trading sessions
- `relVolatility`: This is the relative volatility within a trading session
- `relCloseOpen`: This is the relative difference of the stock opening and closing price

The execution of the genetic algorithm requires the following steps:

1. Extraction of model parameters or variables.
2. Generation of the initial population of trading strategies.
3. Setting up the GA configuration parameters with the maximum number of reproduction cycles allowed, the crossover and mutation ratio, and the soft limit function for population growth.
4. Instantiating the GA algorithm with the scoring/unfitness function.
5. Extracting the fittest trading strategy that can best explain the sharp decline in the price of Goldman Sachs stocks.

## Data extraction

The first step is to extract the model parameters as illustrated for the variation of the stock price between two consecutive trading sessions:

```
val path = "resources/data/chap10/GS.csv"
val src = DataSource(path, false, true, 1)
val price = src |> YahooFinancials.adjClose
val deltaPrice = price.drop(1)
                     .zip(price.dropRight(1))
                     .map(p => (1.0 - p._2/p._1))
```

The extraction of relative variation in volume and volatility is similar to the extraction of the relative variation of the stock price.

## Initial population

The next step consists of generating the initial population of strategies that compete to become relevant to the decline of the price of stocks of Goldman Sachs. The factory is initialized with a set of signals:

```
val NUM_SIGNALS_PER_STRATEGY = 3
val factory = new StrategyFactory(NUM_SIGNALS_PER_STRATEGY)
factory += ("Delta_volume", 1.1, GREATER_THAN, deltaVolume,
deltaPrice)
factory +=  ("Rel_volatility", 1.3, GREATER_THAN, relVolatility.
drop(1), deltaPrice)
…
```

The test code generates `population` by retrieving the pool of `strategies`:

```
val limit = factory.strategies.size // 1 <<4
val population = Population[Signal](limit, factory.strategies)
```

The maximum size of the population (hard limit) is arbitrarily set as 16 times the number of the initial trading strategies (line 1).

At this stage, we need to instantiate a Discretization instance:

```
val R=1024.0
implicit val digitize = new Discretization(R)
```

## Configuration

The four configuration parameters for the GA are the maximum number of reproduction cycles (MAX_CYCLES) allowed in the execution, the crossover (XOVER), the mutation ratio (MU), and the soft limit function (softLimit) to control the population growth:

```
val XOVER = 0.2; val MU = 0.6; val MAX_CYCLES = 250
val CUTOFF_SLOPE = -0.003; val CUTOFF_INTERCEPT = 1.003

val softLimit = (n: Int) => CUTOFF_SLOPE*n + CUTOFF_INTERCEPT
val config = GAConfig(XOVER, MUTATE, MAX_NUM_ITERS, softLimit)
```

The soft limit is implemented as a linearly decreasing function of the number of cycles (n) to retrain the growth of the population as the execution of the genetic algorithm progresses.

## GA instantiation

Let's implement the chromosome scoring function using the formula introduced in the cost/unfitness section. The trading strategy/chromosome scoring function sums up the score for each gene and updates it:

```
val scoring = (chr: Chromosome[Signal]) =>  {
  val signals: List[Gene] = chr.code
  chr.unfitness = signals.foldLeft(0.0)((s, x) => s + x.score)
}
```

The configuration config and the scoring function, scoring, are all you need to create and execute the solver gaSolver:

```
val gaSolver = GASolver[Signal](config, scoring)
```

## GA execution

The execution of the genetic algorithm transforms an initial population to a very small group of the `NFITS` fittest trading strategies:

```
val NFITS = 2
val best = gaSolver |> population
best.fittest(NFITS)
    .getOrElse(ArrayBuffer.empty)
    .foreach(ch => Display.show(s"Best: ${ch.toString(" ")}", logger))
  …
```

## Tests

The cost function *C* and the unfitness score of each trading strategy are weighted for the rate of decline of the price of the Goldman Sachs stock. Let's run two tests:

- Evaluation of the genetic algorithm with an unweighted score function
- Evaluation of the configuration of the genetic algorithm with the weighted score

### The unweighted score

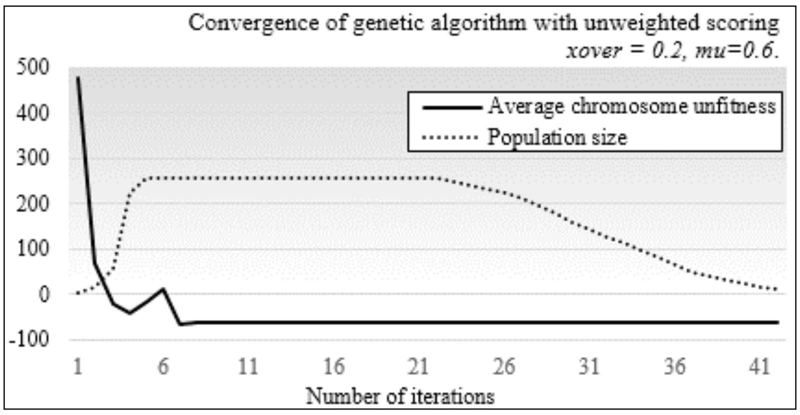
The test uses three different sets of crossover and mutation ratios: (0.6, 0.2), (0.3, 0.1), and (0.2, 0.6). The best trading strategy for each scenario are as follows:

- **0.6-0.2**: For this, `Delta_volume` > 1.10, `Rel_close-Open` > 0.75, and `Rel_volatility` > 0.97 with average chromosome `unfitness` = 0.025
- **0.3-0.1**: For this, `Delta_volatility` > 0.9, `Rel_close-Open` < 0.8, and `Rel_volatility` > 1.77 with `unfitness` = 0.100
- **0.2-0.6**: For this, `Delta_volatility` > 0.9 `Delta_volume` > 33.09, and `Rel_volatility` > 1.09 with `unfitness` = 0.099

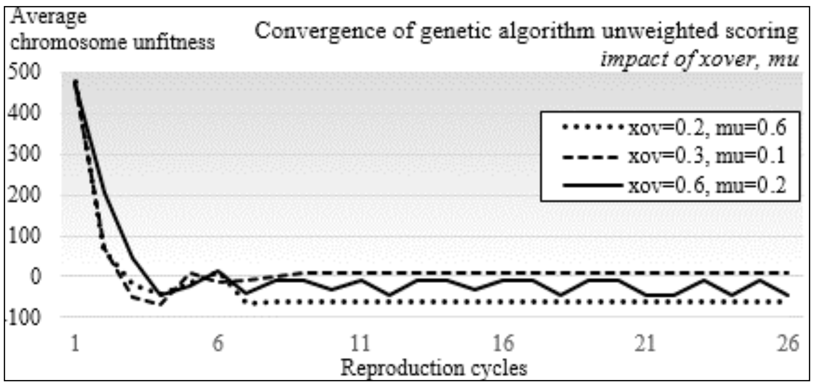
The fittest trading strategy for each case does not differ much from the initial population for one or several of the following reasons:

- The initial guess for the trading signals was good
- The size of the initial population is too small to generate genetic diversity
- The test does not take into account the rate of decline of the stock price

Let's examine the behavior of the genetic algorithm during execution. We are particularly interested in the convergence of the average chromosome unfitness score. The average chromosome unfitness is the ratio of the total unfitness score for the population over the size of the population: Have a look at the following graph:



The GA converges quite quickly and then stabilizes. The size of the population increases through crossover and mutation operations until it reaches the maximum of 256 trading strategies. The soft limit or constraint on the population size kicks in after 23 trading cycles. The test is run again with a different values of crossover and mutation ratio, as shown in the following graph:

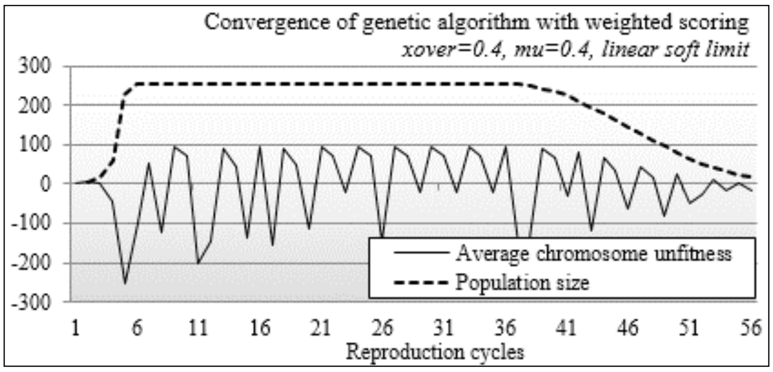


The profile of the execution of the genetic algorithm is not overly affected by the different values of crossover and mutation ratios. The chromosome unfitness score for the high crossover ratio, 0.6, oscillates as the execution progresses. In some cases, the unfitness score between chromosomes is so small that the GA recycles the same few trading strategies.

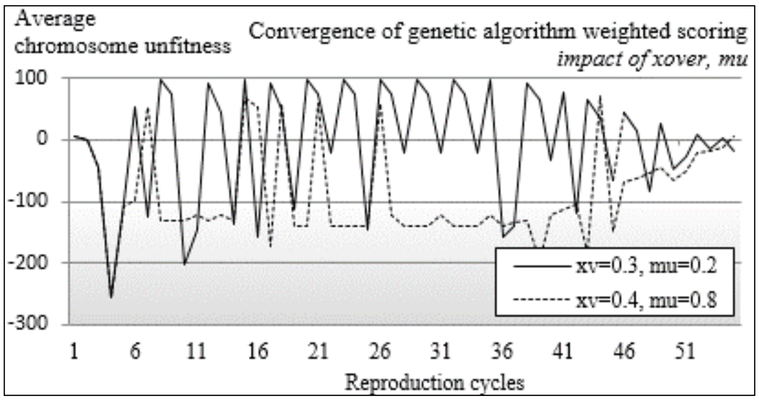
The quick decline in the unfitness of the chromosomes is consistent with the fact that some of the fittest strategies were part of the initial population. It should, however, raise some concerns that the GA locked on a local minimum early on.

## The weighted score

The execution of a test that is similar to the previous one with the weighted unfitness scoring formula produces some interesting results, as shown in the following graph:



The profile for the size of the population is similar to the test using unweighted unfitness. However, the average chromosome unfitness does not stabilize as the optimization goes on until the size of the population is reduced by the soft limit function. This phenomenon is confirmed by running the test using different configurations, as shown in the following graph:



The weighting function adds the rate of decline of the stock price into the scoring of the unfitness. The formula to compute the cost/unfitness of a trading strategy is not a linear function; its complexity increases the odds of the genetic algorithm not converging properly, which is confirmed with extra runs with different values of the crossover and mutation ratios.

The possible solutions to the convergence problem are as follows:

- Make the weighting function additive (less complex)
- Increase the size and diversity of the initial population

## Advantages and risks of genetic algorithms

It should be clear by now that genetic algorithms provide scientists with a powerful toolbox with which to optimize problems that:

- Are poorly understood.
- May have more than one good enough solutions.
- Have discrete, discontinuous, and non-differentiable functions.
- Can be easily integrated with the rules engine and knowledge bases (for example, learning classifiers systems).
- Do not require deep domain knowledge. The genetic algorithm generates new solution candidates through genetic operators. The initial population does not have to contain the fittest solution.
- Do not require knowledge of numerical methods such as the Newton-Raphson, conjugate gradient, or BFGS as optimization techniques, which frighten those with little inclination for mathematics.

However, evolutionary computation is not suitable for problems for which:

- A fitness function cannot be clearly defined
- Finding the global minimum or maximum is essential to the problem
- The execution time has to be predictable
- The solution has to be provided in real time or pseudo-real time

## Summary

Are you hooked on evolutionary computation, genetic algorithms in particular, and their benefits, limitations as well as some of the common pitfalls? If the answer is yes, then you may find learning classifier systems, introduced in the next chapter, fascinating. This chapter dealt with the following topics:

- Key concepts in evolutionary computing
- The key components and operators of genetic operators
- The pitfalls in defining a fitness or unfitness score using a financial trading strategy as a backdrop
- The challenge of encoding predicates in the case of trading strategies
- Advantages and risks of genetic algorithms
- The process for building a genetic algorithm forecasting tool from the bottom up

The genetic algorithm is an important element of a special class of reinforcement learning introduced in the *Learning classifier systems* section of the next chapter.