Extracted from:

Genetic Algorithms and Machine Learning for Programmers

Create AI Models and Evolve Solutions

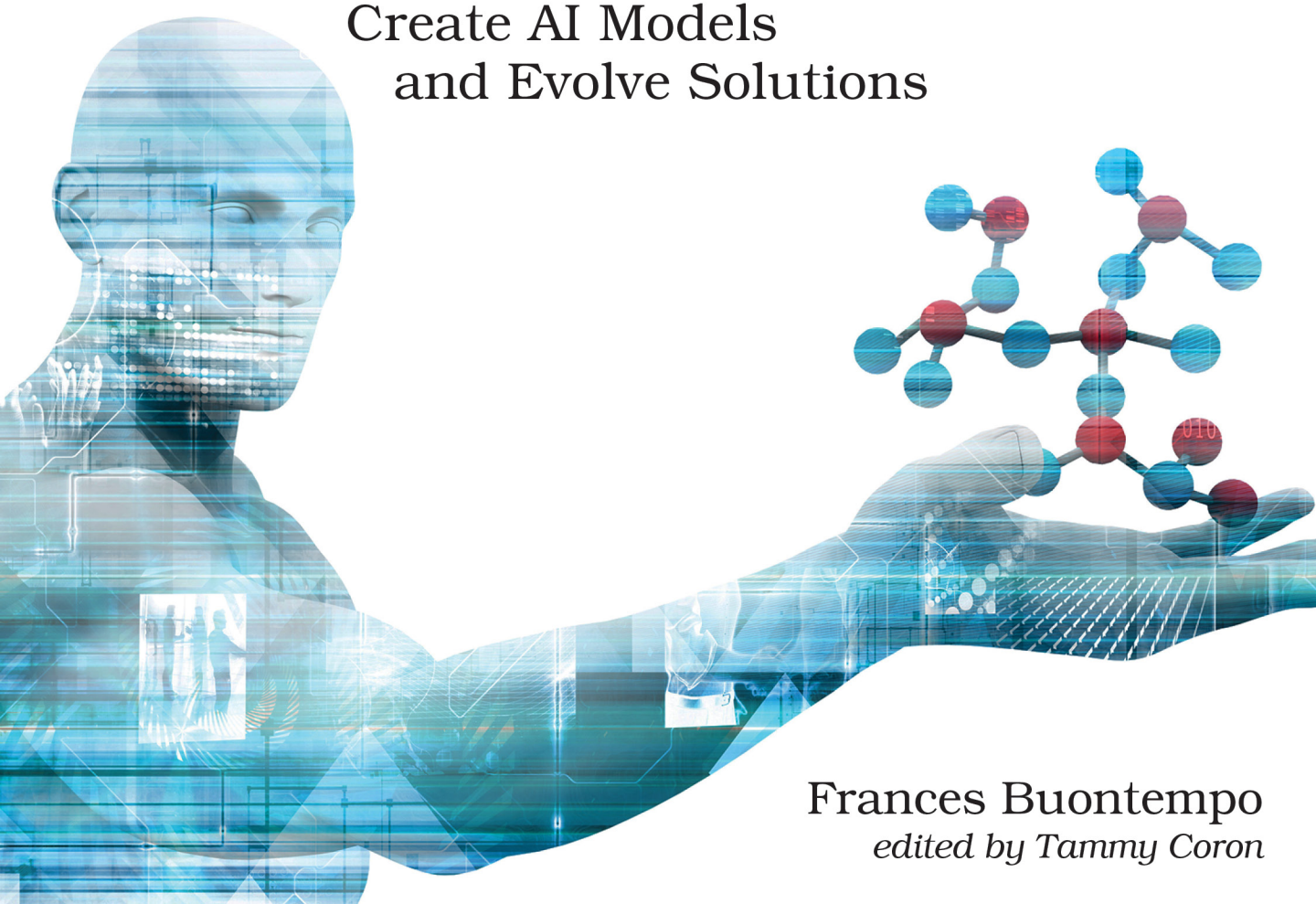
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Genetic Algorithms and Machine Learning for Programmers

Create AI Models and Evolve Solutions



Frances Buontempo
edited by Tammy Coron

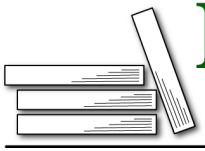
Genetic Algorithms and Machine Learning for Programmers

Create AI Models and Evolve Solutions

Frances Buontempo

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Tammy Coron
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-620-4
Book version: P1.0—January 2019

CHAPTER 3

Boom! Create a Genetic Algorithm

In the previous chapter, you used saved data points a turtle visited as he escaped a paper bag to build a decision tree. By splitting the data on attributes, specifically x and y coordinates, the tree was able to decide whether new points were inside or outside the original paper bag. This is one way to predict values for new data. There are many other ways to make predictions. So many, in fact, that they could fill several volumes with the different approaches. One is enough to get a feel for this flavor of AI. Let's try something completely different. Rather than predicting the future or outcomes, can you find combinations or suitable inputs to solve problems?

- How do you split your investments, maximizing your pension and avoiding buying shares in companies you have moral qualms over?
- How do you create a timetable for classes, making sure there are no clashes, and all the classes get taught and have an assigned room?
- How do you make a seating plan for a wedding, making sure each guest knows someone else at the table and keeping certain relatives as far apart as possible?

These seem like very different problems. The investments will be amounts of money in different schemes. The timetable will tell you what happens when and where. The seating plan will be seat numbers for each guest. Despite these differences, they have something in common. You have a fixed number of investments that need values or classes or guests to arrange in a suitable order. You also have some constraints or requirements, telling you how good a potential solution is. Any algorithm returning a fixed-length array organized or populated to fulfill conditions will solve the problem.

For some problems, you can work through the options or use mathematics to find a perfect solution. For one fixed-rate bond in a portfolio, the mathematics

to find its future value is straightforward. With one class, one teacher and one room, there is only one possible timetable. For a single guest at a wedding, the seating plan is apparent. For two or three guests you have more options, but can try out each and decide what to do. Once you have 25 guests, there are 15,511,210,043,330,985,984,000,000 possible arrangements.[1] Trying each of these, known as *brute force*, against your constraints will take far too long. You could reject a few combinations up front but will still be left with far too many to try. All you need is a list of seat numbers for 25 people. You could try a few at random and might get lucky, but might not. Ideally, you want a way to try enough of the possible arrangements as quickly as possible to increase the chance of finding a good enough seating plan (or timetable, or investment).

There is a machine learning or evolutionary computing method called a *genetic algorithm* (GA) that is ideal for problems like this. A GA finds a solution of fixed length, such as an array of 25 guests' seat numbers, using your criteria to decide which are better. The algorithm starts with randomly generated solutions, forming the so-called initial population, and gradually hones in on better solutions over time. It is mimicking Darwinian evolution, utilizing a population of solutions, and using the suitability criteria to mirror natural selection. It also makes small changes, from time to time, imitating genetic mutation.

The algorithm makes new populations over time. It uses your criteria to pick some better solutions and uses these to generate or *breed* new solutions. The new solutions are made by splicing together *parent* solutions. For investments of bonds, property, foreign exchange and shares, combine bonds and property from one setup with foreign exchange and shares from another, and you have a new solution to try out. For seating plans, swap half of one table with half of another, or swap parts of two seating plans. You might end up with the same seat used twice, so you need to do some fixing up. There are lots of ways to splice together arrays. The GA also mutates elements in the solution from time to time, such as swapping two people's seats. This can make things worse—splitting up a couple might not be good—but can make things improve too. Nonetheless, this keeps variety in the solutions thereby exploring several of the possible combinations.

There are many ways to select parent solutions, tournament and roulette wheel being common. We'll use roulette wheels in this chapter and try tournaments later in Chapter 9, *Dream! Explore CA with GA*, on page ?. Once we've created a GA, we'll have a look at mutation testing to evaluate unit tests, emphasizing mutation as a useful general technique. This chapter adds

1. www.perfecttableplan.com/html/genetic_algorithm.html

to your fundamental concepts, encourages you to question your options, and helps you build a simple genetic algorithm.

Imagine a paper bag with a small cannon inside, which can fire cannonballs at different angles and velocities. If your mission is to find a way to fire these cannonballs out of the bag, how would you go about doing this? You have a few options:

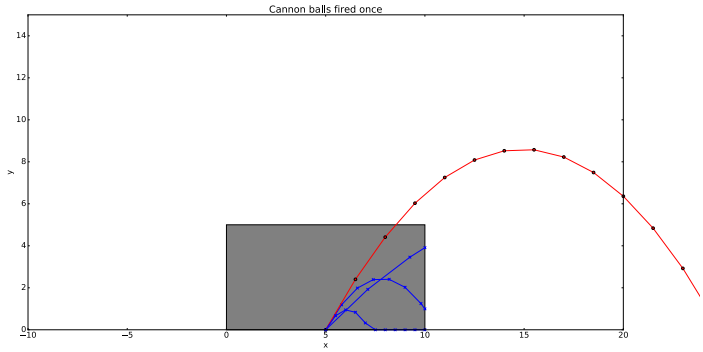
- Work through the mathematics to find suitable ranges of angles and velocities. This is possible for this problem, but won't show us how GAs work.
- Use brute force to try every combination, but this will take ages.
- Build a genetic algorithm to find pairs of angles and velocities that send the cannonballs out of the bag.

Your Mission: Fire Cannonballs

Let's create a genetic algorithm for firing virtual cannonballs out of a paper bag. Let's see how cannonballs move when fired, and start thinking about which paths are better. This will tell us the criteria for the GA. There are two ways these cannonballs can move: straight up or at an angle.

When fired at an angle, cannonballs travel up, either left or right, eventually slowing down due to gravity, following a parabola. When fired straight up at 90 degrees, a similar thing happens. However, instead of a parabola, they come straight down. Cannonballs fired fast enough go into orbit, or reach escape velocity, which is certainly out of the paper bag, but hard to draw.

The trajectories of a few cannonballs are shown in the next figure. They start from an invisible cannon located at the bottom/middle of a gray bag. One cannonball travels up a little, then falls to the bottom of the bag and rolls along. Another two go up and stick to the edge of the bag. Finally, one manages to escape the bag by going high enough and fast enough:



The higher up a cannonball gets at the edge of the bag, the better the angle, velocity pair. Any cannonball over the bag height at the edge escapes. You can use this height as the criteria, or fitness function. Let's see what equations to use to map these trajectories.

The coordinates of a cannonball, or any ballistic, at a specific time (t) can be found using an initial velocity (v) and an angle (θ). With the velocity in meters per second, and the angle in radians, on a planet that has gravity (g)—which, here on Earth is ~9.81 meters per second squared—the coordinates (x, y) of a cannonball (t) seconds after firing, are found using these equations:

$$x = vt\cos(\theta)$$

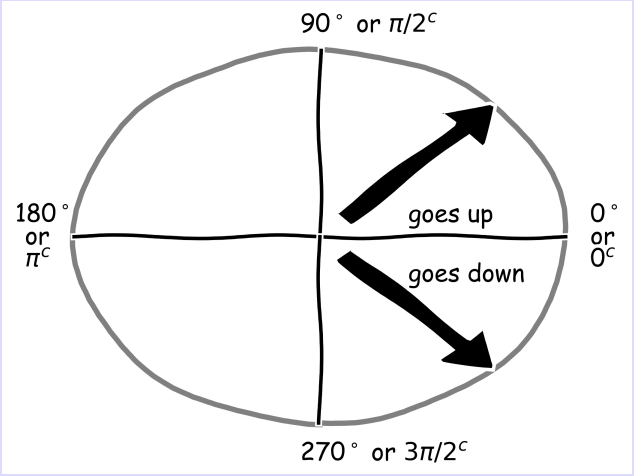
$$y = vt\sin(\theta) - \frac{1}{2}gt^2$$

Joe asks:

What's a Radian?

The trigonometry functions in Python use radians. They start with 0 on the horizontal axis and move around counter-clockwise; see the figure. There are 2π radians in a full circle, which equals 360 degrees. Therefore, 1 radian is equal to 180/π degrees, and 1 degree is equal to π/180. Radians were introduced to make some mathematics easier. You can use the radians conversion function or use radians directly.

The figure shows angles in degrees and radians, starting with 0 on the right and moving counter-clockwise:



The first step in using a GA is encoding your problem. The cannonballs need pairs of numbers—angles and velocities—for their trajectories. Dividing up investments into bonds, property, foreign exchange, and shares finds four numbers—how much in each. Finding a seating plan can use an array with one element per guest. Other problems find letters or strings of bits. Any fixed-length solution is amenable to GAs.

To get started, the GA creates some random pairs of numbers to plug into these equations. The wedding guest problem would also generate some random seating plans to get started. Two guests would also be a pair, while more guests needs a list or array. Armed with pairs of angles and velocities, the GA creates new pairs. It selects some better pairs as *parents* to create or *breed* new attempts. Using the *fittest* pairs to create new solutions to a problem alludes to Darwin's theory of evolution.[2] That's why a GA is an evolutionary algorithm. Two parents are spliced together to make a new solution. Some numbers come from one parent and some from the other. This is called *crossover*, using a simplified model of genetic recombination during sexual reproduction where parents' strands of DNA splice together when living creatures breed.

The GA also uses *mutation*, in this example making either the angle or velocity a bit bigger or smaller. The crossover and mutation operations keep some variety in the population of angle, velocity pairs. Remember, you don't have time to try all the numbers, but want to try as many as possible to increase the chance of finding something that works. The crossover and mutation operations take care of that for us.

How to Breed Solutions

You know the solutions are pairs of numbers, which you need to initialize. You can weed out a few numbers with a little thought since they are bound to fail. Armed with a handful of initial attempts, you need a way to pick better ones to breed even better solutions. Once you have a clear idea of the backbone of the algorithm, you can implement the crossover operator to breed new solutions and mutate these from time to time in Python (either 2.x or 3.x), as described in *Let's Fire Some Cannons*, on page ?. Feel free to write the code as you read, or start thinking about how you want to implement this. Be sure to write some unit tests so that you can use these for *Mutation Testing*, on page ?.

2. https://en.wikipedia.org/wiki/Survival_of_the_fittest

Starting Somewhere

The GA needs pairs of velocities and angles to get started. It can pick any numbers, but some are going to fail. The GA will weed these out, but you can help it along by sharing some things you already know about the problem. For a seating plan, you could ensure couples sit together. For the cannonballs, when the velocity is zero, the cannonball won't go anywhere, so the velocity needs to be a number greater than zero.

What about the angle? Anything more than a full circle is like spinning the cannon around a full circle, plus a bit more—the outcome is the same as just tilting the cannon by the bit more. In fact, anything less than 0 or greater than half a circle fires downwards, so your angles should be in the first half circle: between 0 and π.

How many pairs should you generate? At least one, maybe a few more. The idea is to start small and add more if you think it's needed. Once you have a list of pairs, you come to the clever part of the GA. It runs for a while, picking some of the better pairs from last time to combine into new pairs.

For a While...

The first set of solutions of velocity/angle pairs are created randomly, giving a variety of solutions, some better than others. This is the first *generation*. The GA will use these to make better solutions over time. A GA is a type of guided random search, or *heuristic search*, using these to find improved solutions, using a loop:

```
generation = random_tries()
for a while:
  generation = get_better(generation)
```

This form crops up again and again. This is often described as a random *heuristic* search. Something random happens and is improved guided by a heuristic, *fitness function*, or some other way to refine the guesses.

There are many ways to decide when to stop searching. You can stop after a pre-chosen number of attempts or *epochs*; you can keep going until every pair is good enough, or you can stop if you find one solution that works. It all depends on the problem you are trying to solve. For a seating plan, you could let it run for a few hours and have a look at the suggestions. There is no *one true way* with machine learning algorithms. Many variations have official names, and you might be able to think up some new variations. Do a search for "genetic algorithm stopping criteria" on the internet for further

details. What you find will include talk of NP-hard problems, probabilistic upper bounds, and convergence criteria. Before learning about those, let's create a GA since right now, you have a paper bag to code your way out of!

How to Get Better

The GA uses a heuristic to assess items in the current population or generation. Some of the better pairs are used to make a new generation of solutions. The driving idea here is the survival of the fittest, inspired by Darwinian evolution. Finding fitter parents might make fitter childeren.

To select fitter parents, the GA needs a way to compare solutions. For a seating plan, you can check how many criteria are satisfied. For an investment, you can check how much pension you will get. Which are the fittest (angle, velocity) pairs? There are options. Let's consider two approaches: either find pairs that get out of the bag, or decide a way to rank the pairs by finding a score for how well they did.

For the first approach, you can follow the arc of each cannonball and see which ones leave the bag. Does a ball hit an edge, land back inside the bag, or get out? For the second approach, you can do some math and figure out how far up the ball went, and how close to the edge it was. Look back at the cannonball paths on page 7 to get an idea of the different trajectories.

Both approaches provide a way to measure the viability of a solution. This is the heuristic part of the algorithm, and it is what guides the algorithm toward better solutions. With the first approach, you return a boolean based on whether or not the ball made it out of the bag. With the second approach, you return the y value when the ball is at the edge of the bag. Then you assign a score based on that value: the closer it was to escaping, the better the score.

So which is better? Let's unpack things a bit. If a cannonball nearly gets out but doesn't quite make it, the first option will brutally declare it a failure. With the second option, however, this information will be passed on to future generations. The mutation function might even nudge these toward success.

In the next section, we'll use the height for fitness. Feel free to compare this solution to the more brutal method of using a boolean. Any GA you create needs a heuristic or fitness function, so always needs a bit of thought. A fair bit is hiding behind the random start and gets better in a loop! There are still a few decisions to make, so let's consider these. After that, *Let's Fire Some Cannons*, on page ? walks through an implementation to select parents, breed solutions and mutate them from time to time, giving you a working GA.

Final Decisions

You now have an overview of a genetic algorithm—a random setup and a loop, with the all-important fitness function—but the GA needs some magic numbers to run. How many epochs? One epoch is fine for a trial run, but you need more to see improvement. Instead of a pre-chosen number of epochs, you can wait for an average fitness, minimum fitness, or until one or all pairs work. Looping for a pre-specified number of times is straightforward, so try that first. Of course, you can tweak this and try other options too.

How big should a population be? This number can make a big difference. Imagine if you just had one. How will it breed? If you have a couple, and you used a pass/fail *fitness function*, what happens if neither is good? You can try something random again, but then you're back to the start. Going to the other extreme is a waste too. Trying 1,000,000 solutions when only 10 work is a waste. It is best to start with a small number first, to flush out any problems, and to increase parameters only if needed.

Let's try twelve solutions and ten `epochs`, giving the backbone of the algorithm:

```
items = 12
epochs = 10
generation = random_tries(items)
for i in range (1, epochs):
  generation = crossover(generation)
  mutate(generation)
display(generation)
```

All genetic algorithms have the same core shape; try something random, then loop around trying to improve. In fact, many machine learning algorithms look like this. When you discover a new machine learning algorithm, look for how it starts and where it loops. The differences come from how solutions improve. Let's fill in the implementation details for crossover, mutation, and draw the arcs of the cannonballs with Matplotlib.