Extracted from:

Genetic Algorithms and Machine Learning for Programmers

Create AI Models and Evolve Solutions

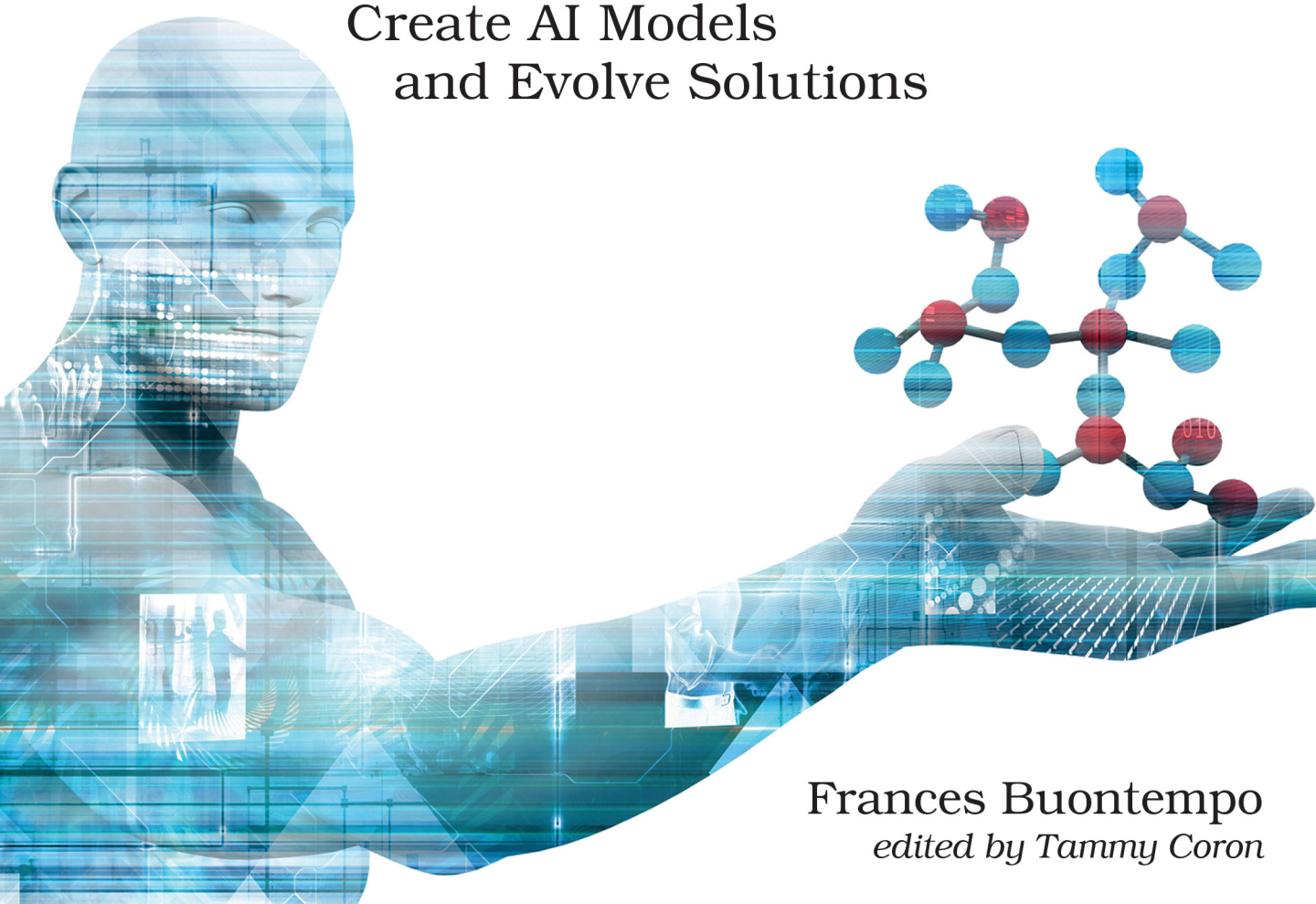
The Pragmatic Bookshelf

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

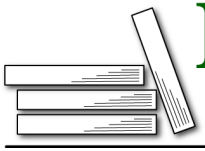
Genetic Algorithms and Machine Learning for Programmers

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 1

Escape! Code Your Way Out of a Paper Bag

This book is a journey into *artificial intelligence* (AI), *machine intelligence*, and *machine learning* aimed at reasonably competent programmers who want to understand how some of these methods work. Throughout this book, you'll use different algorithms to create models, evolve solutions, and solve problems, all of which involve escaping (or finding a way into) a paper bag. Why a paper bag?

In a blog post, Jeff Atwood, co-founder of Stack Overflow, reflects on many programmers' inability to program.[1] He quotes various people saying things like, "We're tired of talking to candidates who can't program their way out of a paper bag."

With that in mind, the paper bag escapology is a perfect metaphor and makes a great case study for applying the various algorithms you'll learn. Plus, this is your chance to stand out from the pack and break out of the proverbial bag.

The problems presented throughout this book demonstrate AI, machine learning, and statistical techniques. Although there's some overlap between the three, most will stick with machine learning. However, it's important to understand that all of them share a common theme: that a computer can learn without being explicitly programmed to do so.

AI isn't new. John McCarthy, the inventor of the Lisp programming language, coined the term artificial intelligence in a proposal for a conference in 1956. He proposed an investigation, writing:

> The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to

1. blog.codinghorror.com/why-cant-programmers-program

> make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves.[2]

Recently, the topic of AI has surfaced again. This is likely because of the increase in computing power.

With today's modern personal computer, AI is more accessible. Many companies now offer automated chatbots to help us online. Robots explore places that are far too dangerous for humans. And thanks to the many programming libraries and frameworks available to handle the complicated mathematics, it's possible to find a neural network implementation, train it, and have it ready to make predictions within minutes. In the 1990s, you'd have to code this yourself, and then wait overnight while it chugged through data.

Many examples of AI involve computers playing games like chess, Breakout, and Go.[3] More generally, AI algorithms solve problems and explore data looking for patterns. The problem-solving part of AI is sometimes called machine learning—which includes analyzing data, allowing companies to spot trends and make money.

Machine learning is also an old term. Arthur Samuel, who built the first self-learning program that played checkers or draughts, introduced the term in 1959.[4] He researched ways to make programs get better at playing games, thereby finding general-purpose ways to solve problems, hence the term *machine learning*.

Machine learning has become a buzzword recently. It's a huge topic, so don't expect to master it any time soon. However, you can understand the basics if you start with some common ideas. You might even spot people trying to blind you with science and think of probing questions to ask:

- How did you build it? If it needs data to learn, remember: Garbage in, garbage out. Bias in, bias out.[5]
- How did you test it? Is it doing what you expect?
- Does it work? Have you got a solution to your problem?
- What parameters did you use? Are these good enough or will something else work better?

2. aaai.org/ojs/index.php/aimagazine/article/view/1904
3. https://www.wired.com/story/vicarious-schema-networks-artificial-intelligence-atari-demo/
4. en.wikipedia.org/wiki/Arthur_Samuel
5. www.designnews.com/content/bias-bias-out-how-ai-can-become-racist/176888957257555

- Does it apply generally? Or does it only work for your current problem and data?

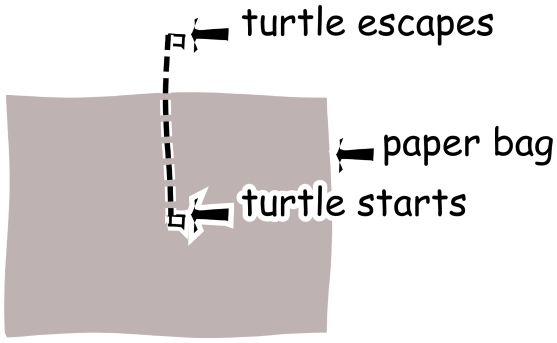
Let's Begin

You'll start your journey by plotting points that are connected by lines. This is not a formal machine learning algorithm, but it introduces a few important terms and provides a clearer picture of what machine learning is and why it matters. Later, you'll use a decision tree and launch into a more formal machine learning algorithm.

The programming language used in this exercise is Python, although the language itself isn't important. In fact, throughout this book, you'll use a combination of Python, C++, and JavaScript. However, you can use any language you want. Some people claim you need to use general-purpose computing on graphics processing units (GPGPU), C++, Java, FORTRAN, or Python to implement AI algorithms. For certain applications, you may need a specific tech stack and a room full of powerful server machines, especially if you're looking to get power and speed for *big data* applications. But the truth is, you can implement any algorithm in the language of your choice; but keep in mind, some languages run quicker than others.

Get Out of a Paper Bag

For this exercise, imagine there's a paper bag with a turtle inside. The turtle is located at a specific point, and his task is to move to different points within his environment until he makes it out of the bag. He'll make a few attempts, and you'll guide his direction, telling him when to stop. To help see what's going on, you'll draw a line that joins the points together. You'll also keep these points around for reference in case the turtle wants to try them again later. By the way, there's nothing stopping the turtle from busting through the sides.



Guided by a *heuristic*, the turtle can make it out alive. A heuristic is a guiding principle, or best guess, at how to solve a problem. Each attempt made is considered a *candidate solution*. Sometimes those solutions work, and sometimes they fail. In the case of your wandering turtle, you need to be careful that he doesn't end up going around in circles and never escaping. To prevent that from happening, you need to decide on the *stopping criteria*. Stopping criteria is a way to make sure an algorithm comes out with an answer. You can decide to stop after a maximum number of tries or as soon as a candidate solution works. In this exercise, you'll try both options.

It's time to get into the mission.

Your Mission: Find a Way Out

To solve this problem, you have lots of decisions to make:

- How do you select the points?
- When do you stop?
- How will you draw the lines?

No matter how precise a description of an algorithm is, you always have choices to make. Many require several parameters to be chosen in advance. These are referred to as *hyperparameters*. Trying to tune these is a difficult problem, but each algorithm presented comes with suggested values that work. They may not be optimal, but you can experiment with these to see if you can solve the problems more quickly, or use less memory.

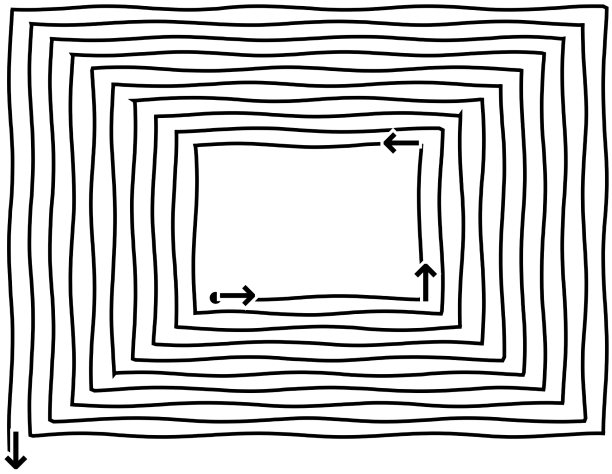
Remember, you need some kind of stopping criteria too. For this problem, you'll be trying two methods: guessing how many steps are needed, and letting the turtle move around until he escapes. For other problems, it's simpler to try a fixed number of iterations and see what happens. You can always stop the algorithms sooner if it solves the problem. Although, sometimes you might let them run past your first guess.

There are a few ways in which the turtle can escape the bag. He can start in the middle and move in the same direction, one step at a time, moving along a straight line. Once he's out, he'll stop, which means you don't need to build in a maximum number of attempts. You do, however, need to choose a step size—but beyond that, there's not much left to decide.

The turtle can also move forward a step and then change direction, repeatedly, increasing the step size each time. Taking an increasing step is a heuristic you can use to guide the turtle. Whichever direction you pick, the turtle is likely to end up outside the bag since he takes bigger steps each time. Using

a fixed angle to change direction and linearly increasing steps will build a *spirangle*.[6] A spirangle is like a spiral, but it has straight edges. Therefore, with this type of movement, the turtle will leave a spirangle trail behind.

If the wandering turtle turns through a right angle, he'll build up a rectangular, or four-angle spirangle. Starting with a smaller step size, he moves forward and turns through 90 degrees, twice. He increases the step size and does this again—forward, turn, forward, turn. By starting at the small circle, he'll leave a trail like the one in the following figure:



The arrows show which way he's moving. By choosing different angles, you get different shapes. If you can't decide what to try, pick a few different angles at random and vary at what point he changes the step size.

To recap, the turtle can move in straight lines or spirangles. He can also make lots of concentric shapes. For example, drawing a small square, then a larger one, and so on until he's drawn a few outside the bag. He'll have to jump to do this. But as long as he draws at least one point outside of the bag, he succeeds.

Of course, the turtle can also pick moves at random, but you'll have no guarantee that he'll end up on the outside of the bag. In fact, many of the algorithms in this book use randomness, whether they be random points in space or random solutions. However, these algorithms will either make candidate solutions guide each other, or they will compel their movement to behave in ways more likely to solve the problems. Learning needs more than random attempts, but it can start there.

6. en.wikipedia.org/wiki/Spirangle

How to Help the Turtle Escape

The turtle knows when to stop and has a few ways to pick the next points. We can pull these methods together into a program to try them all out. We want to be able to see what he's up to as well. The Python turtle package is ideal for showing movement from one point to another, and spirangles are often used to demonstrate its power. It comes with Python, so you don't need to install anything else. That's handy!

Turtle graphics pre-date Python, originating from the Logo programming language, invented by Seymore Papert.[7] The original version moved a robot turtle. He wrote a significant book with Marvin Minsky *Perceptrons: an introduction to computational geometry [MP69]* paving the way for later breakthroughs in AI, making the turtle package an excellent place to start discovering AI and machine learning.

Turtles and Paper Bags

When you import the package, you get a default, right-facing turtle with a starting position of (0, 0). You can choose your turtle shape, or even design your own. This turtle can rotate 90 degrees left, 90 degrees right, or any angle you need. He can also move forward, backward, or goto a specific location. With a little help, you can even get him to draw a paper bag, like this:

Escape/hello_turtle.py

```
Line 1  import turtle
        -
        -  def draw_bag():
        -    turtle.shape('turtle')
        5    turtle.pen(pencolor='brown', pensize=5)
        -    turtle.penup()
        -    turtle.goto(-35, 35)
        -    turtle.pendown()
        -    turtle.right(90)
       10    turtle.forward(70)
        -    turtle.left(90)
        -    turtle.forward(70)
        -    turtle.left(90)
        -    turtle.forward(70)
       15
        -  if __name__ == '__main__':
        -    turtle.setworldcoordinates(-70., -70., 70., 70.)
        -    draw_bag()
        -    turtle.mainloop()
```

7. https://en.wikipedia.org/wiki/Turtle_graphics

In the main function, on line 17, setworldcoordinates sets the window size. When you set your window size, be sure to pick something larger than the paper bag otherwise you won't see what the turtle is doing. Line 19, calls mainloop, which leaves the window open. Without the last line, the window shuts immediately after the turtle makes his move.

On line 4, you set the turtle's shape. Since the turtle starts at the origin, move him left and up on line 7. Because he starts off facing right, rotate him by 90 degrees, on line 9, so that he faces downwards. Then move him forward by 70 steps on line 10. Keep turning, then moving forward to outline the paper bag.

The finished bag is 70 units across, from x=-35 to +35, and 70 units high, also from y=-35 to +35. When you're done, you'll see the three edges of the bag and the turtle:

Now that you have a paper bag and know how to move a turtle, it's time to get to work.

Let's Save the Turtle

The goal is to help the turtle escape the bag you saw earlier on page 6. The easiest way is to make him move in a straight line. He might then march through the sides of the bag. You can constrain him to only escape through the top, but let him go where he wants for now. When he's out, you need to get him to stop. But how do you know when he's out? The left edge of the bag is at -35, and the right is at +35. The bottom and top are also at -35 and +35, respectively. This makes checking his escape attempts easy:

Escape/escape.py

```
def escaped(position):
  x = int(position[0])
  y = int(position[1])
  return x < -35 or x > 35 or y < -35 or y > 35
```

Now all you need to do is set him off and keep him going until he's out:

Escape/escape.py
```
def draw_line():
  angle = 0
  step = 5
  t = turtle.Turtle()
  while not escaped(t.position()):
    t.left(angle)
    t.forward(step)
```

Simple, although a little boring. Let's try some concentric squares.

Squares

To escape using squares, the turtle will need to increase their size as he goes. As they get bigger, he'll get nearer to the edges of the paper bag, eventually going through it and surrounding it. To draw a square, move forward and turn through a right angle four times:

Escape/escape.py
```
def draw_square(t, size):
  L = []
  for i in range(4):
    t.forward(size)
    t.left(90)
    store_position_data(L, t)
  return L
```

Store the position data, including whether or not it's in or out of the paper bag:

Escape/escape.py
```
def store_position_data(L, t):
  position = t.position()
  L.append([position[0], position[1], escaped(position)])
```

You'll need to choose a number of squares to draw. How many do you think you need to get the turtle out of the bag? Experiment if you can't work it out. Now, move your turtle to the bottom left corner and draw a square, increasing the size as you go:

Escape/escape.py
```
def draw_squares(number):
  t = turtle.Turtle()
  L = []
  for i in range(1, number + 1):
    t.penup()
    t.goto(-i, -i)
    t.pendown()
    L.extend(draw_square(t, i * 2))
  return L
```

You extend your list L of positions each time your turtle draws a square so you can save them:

Escape/escape.py

```
def draw_squares_until_escaped(n):
  t = turtle.Turtle()
  L = draw_squares(n)
  with open("data_square", "wb") as f:
    pickle.dump(L, f)
```

You'll use this data in the next chapter.