Numerical Python

CS*101* `lec17`

Heuristic Optimization

2019-11-18

Announcements

quiz: `quiz17` due on Tues 19/11

lab: `lab` on Fri 22/11

hw: `hw09` due this wed

Register for Matlab website

exam02 result? when do you want to know?

Roadmap

Week 1 (9/9)	L0 Intro	L1 PY - Intro
Week 2 (16/9)	L2 PY – Data& Bool	L3 PY – if & Loops
Week 3 (23/9)	L4 PY - Lists	L5 PY - Funcs
30/9	L6 PY - Files	
9/10	L7 PY - Muta	
12/10	Exam 01	
Week 6 (14/10)	L8 PY - Dict	L9 PY - IO
Week 7 (21/10)	L10 PY - Numpy	L11 PY - Plot
Week 8 (28/10)	L12 PY - State	L13 PY - Rand
Week 9 (4/11)	L14 PY – De-bug	L15 PY - Eqn
Week 10 (11/11)	L16 Brute	Exam 02
Week 11 (18/11)	L17 PY - Opt II	L18 PY - Mod
Week 12 (25/11)	L19 PY – Sym I	L20 PY – Sym II
Week 13 (2/12)	L21 PY – Sym III	L22 ML - Intro
Week 14 (9/12)	L23 ML - IO	L24 ML - Poly
Week 15 (16/12)	L25 ML - Stats	L26 ML - Fit
Week 16 (23/12)	Review	
Week 17 (30/12)	Finals	

Objectives

A. Identify when a problem is a good candidate for a heuristic solution.

B. Apply two heuristic optimization techniques (hill climbing, random walk) to solve problems.

Clarification

```
import numpy as np
np.random.seed( 666 )
np.random.uniform( size=5 )
```

Do the 5 random numbers change from one run to another?
Remove np.random.seed(666)
Now, do the 5 random numbers change from one run to another?

Optimization Redux

Question

```
x = '12345'
y = '67890'

for a in itertools.product( x,y ):
    print( ' '.join( a ) )
```

Which of the following is *not* printed?

A `'1 6'`

B `'4 6'`

C `'6 7'`

D `'5 0'`

Question

```
x = '12345'
y = '67890'

for a in itertools.product( x,y ):
    print( ' '.join( a ) )
```

Which of the following is *not* printed?

A `'1 6'`

B `'4 6'`

C `'6 7'` ⋆

D `'5 0'`

Optimization

Brute-force search of a password:

```
def check_password( pwd ):
    if pwd == 'pas':
        return True
    else:
        return False


chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ
       abcdefghijklmnopqrstuvwxyz0123456789'
for pair in itertools.product( chars, repeat=3 ):
    pair = ''.join( pair )
    if check_password( pair ):
        print( pair )
```

Optimization

Brute-force search of a password:

$$
\begin{aligned}
& 2 \times n(\text{alphabet}) + n(\text{digits}) + n(\text{special}) \\
= \quad & 2 \times 26 + 10 + \{24\text{–}32\} \\
= \quad & \{86\text{–}94\}
\end{aligned}
$$

per letter!

Brute-force search

Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

Brute-force search

Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

Characters	Search Space
1	86
2	$86^2 = 7\,396$

Brute-force search

Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

Characters	Search Space
1	86
2	$86^2 = 7\,396$
3	$86^3 = 636\,056$
4	$86^4 = 54\,700\,816$
5	$86^5 = 4\,704\,270\,176$

Brute-force search

Assume that a password can contain characters from the alphabet (upper- and lower-case); digits; and a selection of special characters (ampersand, dash): 86 characters.

Characters	Search Space
1	86
2	$86^2 = 7\,396$
3	$86^3 = 636\,056$
4	$86^4 = 54\,700\,816$
5	$86^5 = 4\,704\,270\,176$
10	$86^{10} = 2.2 \times 10^{19}$
20	$86^{20} = 4.9 \times 10^{38}$

Brute-force search

If Python can try a password attempt every 1×10^{-7} s, how long does it take to crack a password of length *n*?

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	$7\,396$	7.4×10^{-4} s
3	$636\,056$	6.4×10^{-2} s
4	$54\,700\,816$	5.4 s
5	$4\,704\,270\,176$	470.4 s

Brute-force search

If Python can try a password attempt every 1×10^{-7} s, how long does it take to crack a password of length *n*?

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	$7\,396$	7.4×10^{-4} s
3	$636\,056$	6.4×10^{-2} s
4	$54\,700\,816$	5.4 s
5	$4\,704\,270\,176$	470.4 s
10	2.2×10^{19}	2.2×10^{12} s $= 6.9 \times 10^{4}$ years

Brute-force search

If Python can try a password attempt every 1×10^{-7} s, how long does it take to crack a password of length *n*?

Characters	Search Space	Time
1	86	8.6×10^{-6} s
2	$7\,396$	7.4×10^{-4} s
3	$636\,056$	6.4×10^{-2} s
4	$54\,700\,816$	5.4 s
5	$4\,704\,270\,176$	470.4 s
10	2.2×10^{19}	2.2×10^{12} s $= 6.9 \times 10^{4}$ years
20	4.9×10^{38}	4.9×10^{31} s

Optimization

On vacation, you purchase a collection of *n* souvenirs of varying weight and value. When it comes time to pack, you find that your bag has a weight limit of 50 kg. What is the best set of items to take on the flight?

We also used Brute-force method to solve...

Heuristic Optimization

Heuristic optimization

Used when a best solution is impossible or impractical

Heuristic optimization

Used when a best solution is impossible or impractical

Using a **figure of merit**, we can classify candidate solutions by how good they are.

Heuristic optimization

Used when a best solution is impossible or impractical

Using a **figure of merit**, we can classify candidate solutions by how good they are.

Heuristic algorithms don’t guarantee the ‘best’ solution, but are often adequate (and the only choice!)*.

* A functional program will be pretty long, you are not expected to write one without any hints/helps

Heuristic optimization strategy

Hill-climbing

Random sampling

Random walk

M1: Hill-climbing algorithm

Strategy: Always selecting the ”next best” neighbour which improves on present one.

M1: Hill-climbing algorithm

Strategy: Always selecting the ”next best” neighbour which improves on present one.

Analogy: Trying to find the highest hill by only taking a step uphill from where you are.

M1: Hill-climbing algorithm

Strategy: Always selecting the ”next best” neighbour which improves on present one.

Analogy: Trying to find the highest hill by only taking a step uphill from where you are.

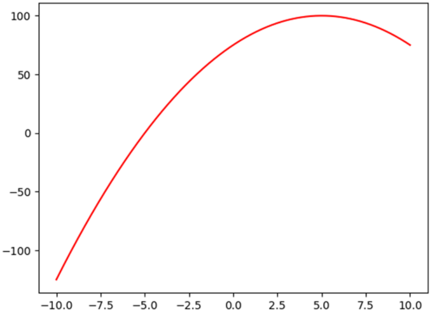
Pitfall: Finding a *local* optimum instead of the global optimum.

M1: Hill-climbing algorithm

A. Set up a figure of merit, *f*. Something that can be used to compare.
B. Select a starting guess, x_0.
C. Change a feature of the guess.
D. If this improves, keep it and cycle.
E. If no improvement is possible, terminate.

Example: One variable

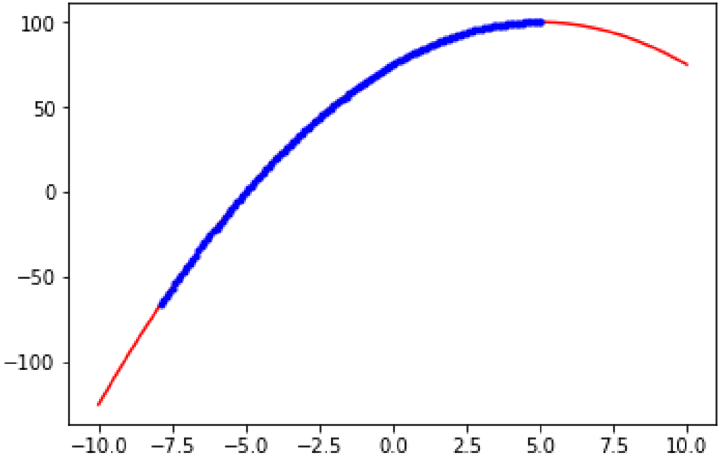
$$f(x) = 100 - (x-5)^2 \qquad x \in \{-10, +10\},$$




```
# Hill-climbing algorithm begins here. ################################
X = np.linspace( -10,10,1001 ) # set up a grid in x
x = -8
l = -0.1
r = +0.1

i = 0
num_steps = 10000
best_x = x
best_f = f( best_x )
last_x = np.nan
last_f = np.nan
steps = np.empty( ( num_steps+1,2 ) )
while ( not np.isclose( last_f,best_f,rtol=1e-5 ) ) and ( i < num_steps ):
    # Check the neighbors, accept the best improvement.
    last_x = best_x
    last_f = best_f
    trial_x_l = best_x + l
    trial_x_r = best_x + r
    if f( trial_x_l ) > best_f:
        best_x = trial_x_l
        best_f = f( best_x )
    elif f( trial_x_r ) > best_f:
        best_x = trial_x_r
        best_f = f( best_x )
    else:
        # either the absolute best scenario has been found,
        # or the step size is too large
        l = 0.5*l
        r = 0.5*r
```

Example: One variable

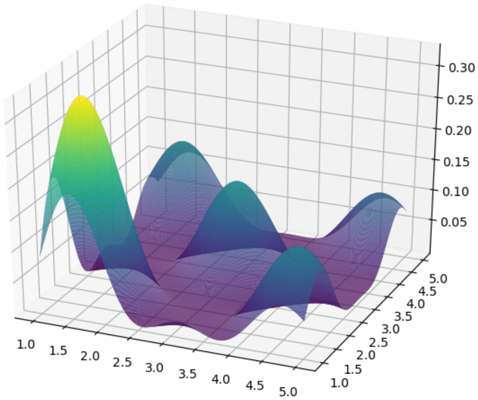
Result:



Example: Two-variable

$$f(x, y) = \frac{1}{\sqrt{2x^2 + 2y^2}} \left(\cos^4 x - 2\cos^2 x \sin^2 y + \sin^4 y\right)$$

$$x \in \{+1, +5\}, y \in \{+1, +5\}$$



Example: Two-variable code

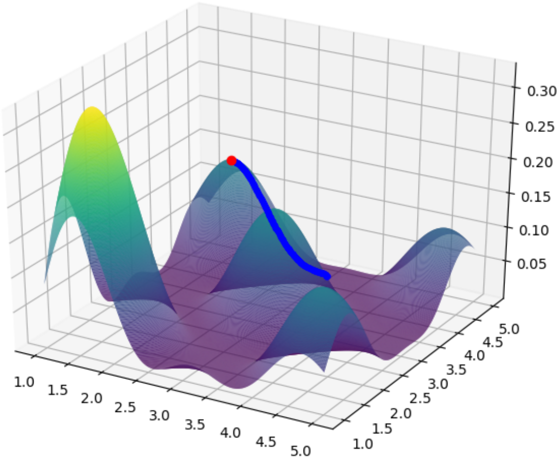
Full code on RELATE website:

```
# Hill climbing algorithm begins here. ############
X = np.linspace( 1,5,401 ) # set up a grid in x
Y = np.linspace( 1,5,401 ) # set up a grid in y
xy = np.random.randint( 401,size=(2,) )
u = np.array( ( 0,-1 ) )  # "up"
d = np.array( ( 0,+1 ) )  # "down"
l = np.array( ( -1,0 ) )  # "left"
r = np.array( ( +1,0 ) )  # "right"

num_steps = 500
best_xy = xy
best_f = f( xy[ 0 ],xy[ 1 ] )
steps = np.empty( ( num_steps,3 ) )
```

Example: Two-variable code

```
for i in range( num_steps ):
  # Try a step in each direction until
      no improvement is possible.
  trial_xy = xy.copy()
  # cycle through directions to step
  step_dir = ( u,d,l,r )[ i % 4 ]
  trial_xy = ( xy + step_dir ) % X.shape[ 0 ]
  xt = X[ trial_xy[ 0 ] ]
  yt = Y[ trial_xy[ 1 ] ]

  if f( xt, yt )>best_f:
    # If the solution improves, accept it.
    best_f = f( xt, yt )
    best_xy = trial_xy.copy()
    xy = trial_xy.copy()
```

Example: Two-variable



Observe the red dot on the hill top. A ”good enough” solution that is local maxima.

M2: Random sampling

Strategy: Choosing at random a candidate solution (sometimes within a constrained space).

M2: Random sampling

Strategy: Choosing at random a candidate solution (sometimes within a constrained space).

Analogy: Picking random heights in the region of a hill, accepting the tallest as the highest.

M2: Random sampling

Strategy: Choosing at random a candidate solution (sometimes within a constrained space).

Analogy: Picking random heights in the region of a hill, accepting the tallest as the highest.

Pitfall: Without good constraints, missing the optimum value.

M3: Random walk

Also uses random numbers, but:

> **S**trategy: Tweaking the current candidate solution at random, and **possibly** rejecting the solution if worse.

M3: Random walk

Also uses random numbers, but:

Strategy: Tweaking the current candidate solution at random, and **possibly** rejecting the solution if worse.

Analogy: Choose random steps near a hill, but **maybe not** take the step if it's worse.

M3: Random walk

Also uses random numbers, but:

Strategy: Tweaking the current candidate solution at random, and **possibly** rejecting the solution if worse.

Analogy: Choose random steps near a hill, but **maybe not** take the step if it's worse.

Pitfall: Converging slowly, can still miss best candidate solution.

M3: Random walk

Also uses random numbers, but:

Strategy: Tweaking the current candidate solution at random, and **possibly** rejecting the solution if worse.

Analogy: Choose random steps near a hill, but **maybe not** take the step if it's worse.

Pitfall: Converging slowly, can still miss best candidate solution. BUT: has a way to avoid getting stuck in a local optima.

M3: Random walk algorithm

A. Set up a figure of merit *f*.
B. Select a starting guess x_0.
C. Change a random feature of the guess.
D. If this improves, keep it and cycle.
E. If this does not improve, *sometimes* keep it anyway.
F. When number of trials has been reached, terminate.

M3: Random walk

Full code on RELATE website:

```
# Random walk algorithm begins here. ##############
X = np.linspace( 1,5,401 ) # set up a grid in x
Y = np.linspace( 1,5,401 ) # set up a grid in y
xy = np.random.randint( 401,size=(2,) )
u = np.array( ( 0,-1 ) )  # "up"
d = np.array( ( 0,+1 ) )  # "down"
l = np.array( ( -1,0 ) )  # "left"
r = np.array( ( +1,0 ) )  # "right"

num_steps = 10000
best_xy = xy
best_f = f( xy[ 0 ],xy[ 1 ] )
steps = np.empty( ( num_steps,3 ) )
```

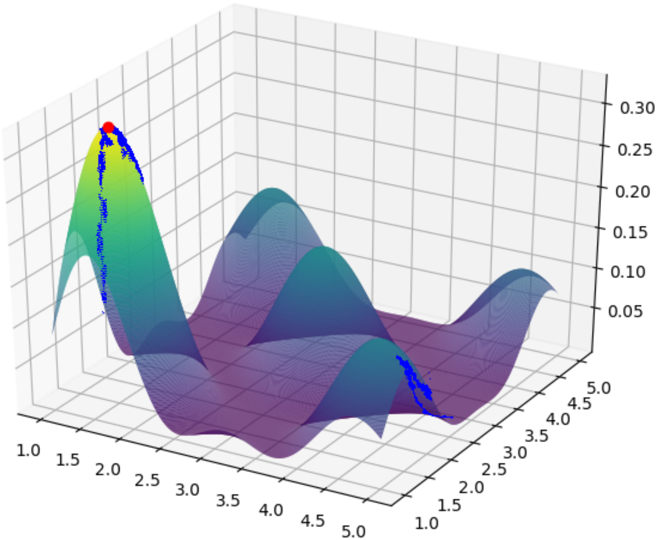
M3: Random walk

```
for i in range( num_steps ):
  # Take a random step, 25% chance in each directio
  trial_xy = xy.copy()
  chance = np.random.uniform()
  if chance < 0.25:
    trial_xy = ( xy + u ) % Y.shape[ 0 ]
  elif chance < 0.5:
    trial_xy = ( xy + d ) % Y.shape[ 0 ]
  elif chance < 0.75:
    trial_xy = ( xy + l ) % X.shape[ 0 ]
  else:
    trial_xy = ( xy + r ) % X.shape[ 0 ]
```

M3: Random walk

```
xt = X[ trial_xy[ 0 ] ]
yt = Y[ trial_xy[ 1 ] ]
if f( xt, yt ) > best_f:
  # If the solution improves, accept it.
  best_f = f( xt, yt )
  best_xy = trial_xy.copy()
  xy = trial_xy.copy()
else:
  # If the solution does not improve,
  # sometimes accept it.
  chance = np.random.uniform()
  if chance < 0.25:
    xy = trial_xy.copy()
```

M3: Random walk



Heuristic Optimization

When we use heuristic optimization methods, we are ok with a ”good enough” solution

If we want to crack a password, can we have a ”good enough” solution?

So to use heuristic optimization, we require:

A. A problem with relative solution assessment
B. An algorithm to assess solutions

Example

Our different optimization strategies, so far:

A. Brute-force (last lecture)
B. Hill-climbing
    Select heaviest item, then add next heaviest, etc.
    Select most valuable item, then add next most valuable item, etc.
C. Random sampling
D. Random walk: sample randomly, then iteratively allow changes based on probability

Setup - Your LV bags

```
import numpy as np
import matplotlib.pyplot as plt
import itertools

n = 10
#Num of bags
items   = list( range( n ) )
#Weight of each bag
weights = np.random.uniform( size=(n,) ) * 50
#Value of each bag
values  = np.random.uniform( size=(n,) ) * 100
```

Setup - How you decide

```
def f( wts, vals ):
    total_weight = 0
    total_value = 0

    for i in range( len( wts ) ):
        # Add weight
        total_weight += wts[ i ]
        # Add value
        total_value  += vals[ i ]

    if total_weight >= 50:
        return 0
    else:
        return total_value
```

Brute-force search

```
import itertools

max_value = 0.0
max_set = None
for i in range(n):
    for set in itertools.combinations( items,i ):
        wts  = []
        vals = []
        for item in set:
            wts.append( weights[ item ] )
            vals.append( values[ item ] )
        value = f( wts,vals )
        if value > max_value:
            max_value = value
            max_set = set
```

Hill-climbing search

```
max_wt = 50.0

wts_orig  = wts[ : ]
vals_orig = vals[ : ]

best_vals = [ ]
best_wts  = [ ]
best_vals.append( max( vals ) )
best_wts.append( wts[ vals.index( max( vals ) ) ] )
wts.remove( wts[ vals.index( max( vals ) ) ] )
vals.remove( max( vals ) )
```

Hill-climbing search

```
while sum(best_wts) + wts[vals.index(max(vals))]
        < max_wt:
    best_vals.append( max( vals ) )
    best_wts.append( wts[ vals.index( max( vals ) )
    wts.remove( wts[ vals.index( max( vals ) ) ] )
    vals.remove( max( vals ) )
```

Random walk - structure

```
# try a configuration at random
# alter it at random with small likelihood
# of getting worse
for t in range( 1000 ):
  # two possible moves:  adding or removing
  if f( next_wts,next_vals ) >
           f( trial_wts,trial_vals ):
    # if improvement, accept the change
    ...............
  else:
    # if no improvement, *maybe* accept the change
    ...............
  # if all-time best, track it
  ...............
```

See full code in random-walk.py in lec17 in RELATE

Comparing Results

Comparing results

`array`s don't play nicely with comparisons:

```
one = np.ones( ( 5, ) )
if one == 1:
    print( 'setup correct' )
```

Comparing results

`array`s don't play nicely with comparisons:

```
one = np.ones( ( 5, ) )
if one == 1:
    print( 'setup correct' )
```

ValueError: The truth value of an array with more than one element is ambiguous.

Comparing results

`array`s don't play nicely with comparisons:

```
one = np.ones( ( 5, ) )
if one == 1:
    print( 'setup correct' )
```

ValueError: The truth value of an array with more than one element is ambiguous.

Which element is compared? It's ambiguous.

Comparing results

`array`s have the built-in methods `any` and `all`:

```
one = np.ones( ( 5, ) )

if ( one == 1 ).all():
    print( 'setup is all ones' )
```

Comparing results

`array`s have the built-in methods `any` and `all`:

```
one = np.ones( ( 5, ) )

if ( one == 1 ).all():
    print( 'setup is all ones' )

domain = np.linspace( 0,10,11 )
if ( domain == 1 ).any():
    print( 'setup contains one' )
```

Summary

A. Heuristic optimization - when optimal is not practical but ”good enough” is good enough
B. Hill-climbing method
C. Random sampling and random walk
D. Need way to quantify to say it is ”good enough” - figure of merit or cost function
E. numpy comparing elements of an array: `.all( )` or `.any( )`