Gradient Descent

MATH1900: Machine Learning

Location: http://people.sc.fsu.edu/∼jburkardt/classes/ml_2019/gradient_descent/gradient_descent.pdf

How low can we go?

Minimization (Gradient Descent)

Given a function $f(x)$ with derivative $f'(x)$, find a value x for which $f(x)$ attain its minimum value.

1 Minimize a quartic function using gradient descent

Consider the example function named *quartic* whose formula is:

$$f(x) = 2x^4 - 4x^2 + x + 20$$

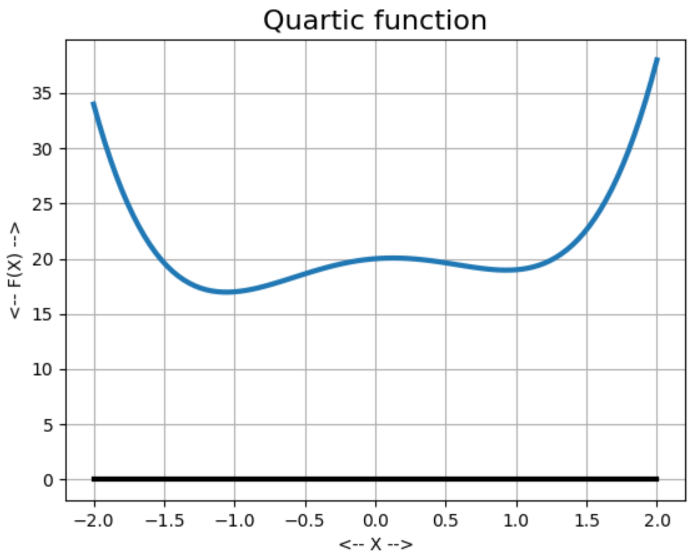
Suppose that this function measures a cost, based on the choice of x. We want to choose a value $-2 \le x \le 2$ which minimizes the cost $f(x)$.

Random sampling, or plotting, might give us some clues, but we would like to find an automatic procedure that can minimize a function based on its formula. If we start with some randome point x in the interval, then we would naturally want to take a small step to the right or left, if only we knew in which direction the function decreases. But if we have access to the derivative, $f'(x)$, then that information is computable too. If $f'(x)$ is positive, the function decreases to the left, and if negative, it decreases to the right.

For our quartic function, the derivative is

$$f'(x) = 8x^3 - 8x + 1$$

If we started our investigation at $x = 1.5$, then $f(x) = 22.625$, and $f'(x) = 16.0$ so this tells us that if we are looking for lower values of $f(x)$, we should look to the left (at lower values of x) - at least in the nearby neighborhood. We can see that this is so by plotting the function:



The quartic() function

This suggests a method for moving towards a low point of the plot of $f(x)$. Just take a small step in the direction opposite to the derivative. Keep doing that until the derivative gets so small that we are evidently near a critical point, which we can hope is a local minimum. On the other hand, if we take a step in what should be the direction of decrease, but the function rises, then we may have stepped past a minimizer, and we should back up and try a smaller step.

This general technique is known as *gradient descent*.

2 Gradient descent pseudocode for 1D case

Pseudocode for a gradient descent method might look like this,

```
gradient_descent1 ( f, df, x, r, dxtol, dftol, itmax )

# gradient descent for a function of 1 parameter

  it = 0

  Loop1:

    if |df(x)| < dftol ) return

    xold = x
    beta = 1.0
    Loop2:
      it = it + 1
      if ( itmax < it ) return x
      dx = - beta * r * df(xold)
      if ( |dx| < dxtol ) return
      x = xold + dx
      if f(x) < f(xold) break Loop2
      beta = beta / 2
    Loop2 end

  Loop1 end

  Return x

gradient_descent1 end
```

Listing 1: Pseudocode for gradient descent.

The input quantities `f` and `df` define the function and its derivative, `x` is the starting point, `r` is the *learning rate*, a sort of stepsize control, `dxtol` and `dftol` are tolerances for the minimum size of the step and derivative, and `itmax` limits the number of iterations.

3 Example: Applying gradient descent to quartic()

For our quartic function, we prepare a file *quartic.py* containing our definitions of $f(x)$ and $f'(x)$:

```
def quartic ( x ):
  f = 2.0 * x ** 4 - 4.0 * x ** 2 + x + 20.0
  return f

def quartic_df ( x ):
  df = 8.0 * x ** 3 - 8.0 * x + 1.0
  return df
```

Listing 2: Definitions of f(x) and f'(x)

and then call `gradient_descent1()`:

```
  from quartic import quartic
  from quartic import quartic_df
  from gradient_descent1 import gradient_descent1
  x0 = -1.6309821
  r = 0.05
  dxtol = 0.001
  dftol = 0.001
  itmax = 100
  x, it = gradient_descent1 ( quartic, quartic_df, x0, r, dxtol, dftol, itmax )
```

Listing 3: Calling gradient_descent1 for the quartic function.

Our results are:

it	x	f(x)	f'(x)
0	-1.6309821	21.880899	-20.660779
1	-0.5979431	18.227577	4.0732556
2	-0.8016058	17.453904	3.2921312
3	-0.9662124	17.042614	1.5135112
4	-1.0418880	16.972743	0.28709358
5	-1.0562427	16.970507	0.022776196
6	-1.0573815	16.970493	0.0013615096
7	-1.0574496	16.970493	7.9303703e-05

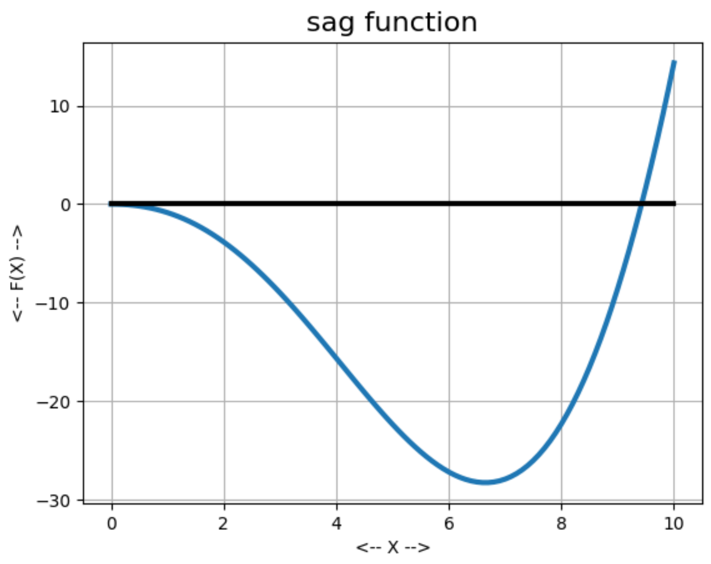
From these results, you can see we ended up on the left side of the plot, because our starting point was nearest to that local minimum.

4 Exercise: Applying gradient descent to sag()

The example function *sag()* has the definition:

$$f(x) = x^2 * \cos(\frac{x + 3\pi}{4})$$

A plot suggests that there is a minimizer between 0 and 10:



The sag() function

Create a file *sag.py* defining $f(x)$ and $f'(x)$: and then call `gradient_descent1()` with a starting point in $[0, 10]$ and see if you can find a value of x for which $f(x)$ is minimized.

5 Gradient Descent in Higher Dimensions

Now let's consider what happens if we have a function of multiple variables. For simplicity, let's consider the two-dimensional case, which we may think of as involving a function $f(x_1, x_2)$ or $f(x, y)$. An example of such a function is named *hex2()*, and has the form

$$f(x, y) = 2x^2 - 1.05x^4 + x^6/6 + xy + y^2$$

We can ask for a minimizer of this function, which we suspect is somewhere in the range $-3 \leq x, y \leq +3$. As before, we will use derivative information, known as the *gradient*, symbolized by ∇f:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots \frac{\partial f}{\partial x_n} \right]$$

For our problem, this information is $\left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]$ with values:

$$\frac{\partial f}{\partial x} = 4x - 4.2x^3 + x^5 + y$$
$$\frac{\partial f}{\partial y} = x + 2y$$

6 Gradient descent pseudocode for multiple dimensions

The gradient descent code for multiple dimensions is almost the same, except for two places where we need to use a norm instead of an absolute value; also `x`, `xold`, `dx` and `df()` are vectors now, rather than scalars.

```
gradient_descent2 ( f(), df(), x, r, dxtol, dftol, itmax )

# gradient descent for a function of multiple parameters

  it = 0

  Loop1:

    if $||df(x)||$ < dftol ) return
    xold = x
    beta = 1.0

    Loop2:
      it = it + 1
      if ( itmax < it ) return
      dx = - beta * r * df(xold)
      if ( $||dx||$ < dxtol ) break Loop1
      x = xold + dx
      if f(x) < f(xold) break Loop2
      beta = beta / 2
    Loop2 end

  Loop1 end

  Return x, it

gradient_descent2 end
```

Listing 4: Pseudocode for gradient descent in multiple dimensions.

7 Example: Applying gradient descent to hex2()

For our *hex2()* function, we prepare a file *hex2.m* containing our definitions of $f(x)$ and $f'(x)$. Notice that the input quantity `x` and the output quantity `df` are now vectors of length 2:

```
def hex2 ( x ):
  value = 2.0 * x[0]**2 - 1.05 * x[0]**4 + x[0]**6 / 6.0 + x[0] * x[1] + x[1]**2
  return value

def hex2_df ( x ):
  import numpy as np
  df = np.array ( [ \
    4.0 * x[0] - 4.2 * x[0]**3 + x[0]**5 + x[1], \
    x[0] + 2.0 * x[1] ] )
  return df
```

Listing 5: Definitions of f(x) and f'(x) for hex2().

and then call `gradient_descent2()`:

```
  import numpy as np
  from gradient_descent2 import gradient_descent2
  x0 = np.array ( [ 2.0, 1.5 ] )
  r = 0.10
  dxtol = 0.00001
  dftol = 0.001
  itmax = 100
  x, it = gradient_descent2 ( hex2, hex2_df, x0, r, dxtol, dftol, itmax )
```

Listing 6: Calling gradient_descent2 for hex2().

Our results are:

it	x	y	f(x)	dfdx	dfdy
0	2.000000	1.500000	7.116666	7.970000	5.000000
1	1.210000	1.000000	3.410503	0.993186	3.210000
2	1.110681	0.679000	2.397414	1.057327	2.468681
3	1.004948	0.432131	1.741589	1.214253	1.869212
4	0.883523	0.245210	1.277457	1.420986	1.373944
5	0.741424	0.107816	0.901377	1.585770	0.957056
..	...	...	...		...
41	0.000248	-0.000599	3.340249e-07	0.000393	-0.000950
42	0.000209	-0.000504	2.364863e-07	0.000331	-0.000800

We can see that the function value has been greatly decreased, and that the derivative components are so small that we do not expect any significant improvement by continuing the computation.

8 Gradient Descent for data fitting

Many machine learning problems start with a mass of data, in which one y is approximately determined by one or more variables that we think of as x. For instance, we may have n pairs of values (x_i, y_i). We look for parameters b and m in a linear model $y = b + m * x$ that approximates our data well.

The numbers b and m are unknown. For any particular i, we will measure the error e_i as the square of the difference between the model's prediction based on x_i, and the actual data y_i:

$$e_i(b, m) = (y_i - b - mx_i)^2$$

Now we want to minimize the sum of all those functions. We call this single function $E(b, m)$:

$$E(b, m) = \sum_{i=1}^{n} e_i(b, m)$$

We want to make this problem this look like a typical case for gradient descent. We start by defining a single variable $bm = [b, m]$, which packs all our variables into one. Correspondingly, we can now write $f(bm) = f([b, m]) = E(b, m)$.

We need to create a procedure to evaluate $f(bm)$, which might look like this:

```
def ford_f(bm):
  b = bm[0]
  m = bm[1]
  x, y = ford_data ( )
  f = np.sum ( ( y - b - m * x )**2 )
  return f
```

Listing 7: Evaluate f(bm) for Ford data

and a corresponding procedure to evaluate $df(bm)$:

```
def ford_df(bm):
  b = bm[0]
  m = bm[1]
  x, y = ford_data ( )
  dfdb = - 2.0 * np.sum ( ( y - b - m * x ) )
  dfdm = - 2.0 * np.sum ( ( y - b - m * x ) * x )
  df = np.array ( [ dfdb, dfdm ] )
  return df
```

Listing 8: Evaluate gradient $\nabla(f)$ for Ford data

and now we need to write a function `ford_data()` which returns a copy of our data. The gradient descent method does not perform well if the variables have very different scales. To be safe, we can always rescale each column to lie between 0 and 1. However, this means that our values of `b` and `m` will apply to the normalized data. If we want to write the formula in terms of the original data, we have to undo the normalization (a cleanup task which we will not do here!)

```
def ford_data ( ):

  data = np.loadtxt ( 'ford_data.txt' )

  x = data[:,0]
  y = data[:,1]

  x = ( x - np.min ( x ) ) / ( np.max ( x ) - np.min ( x ) )
  y = ( y - np.min ( y ) ) / ( np.max ( y ) - np.min ( y ) )

  return x, y
```

Listing 9: Return the values of the Ford data

After writing these three procedures, we can call `gradient_descent2()` just like we did for the simpler *hex2()* problem:

```
  import numpy as np
  from gradient_descent2 import gradient_descent2
  bm0 = np.array ( [ 0.5, 0.0 ] )
  r = 0.01
  dxtol = 0.001
  dftol = 0.001
  itmax = 1000
  bm, it = gradient_descent2 ( ford_f, ford_df, bm0, r, dxtol, dftol, itmax )
```

Listing 10: Calling gradient_descent2 for the Ford data.

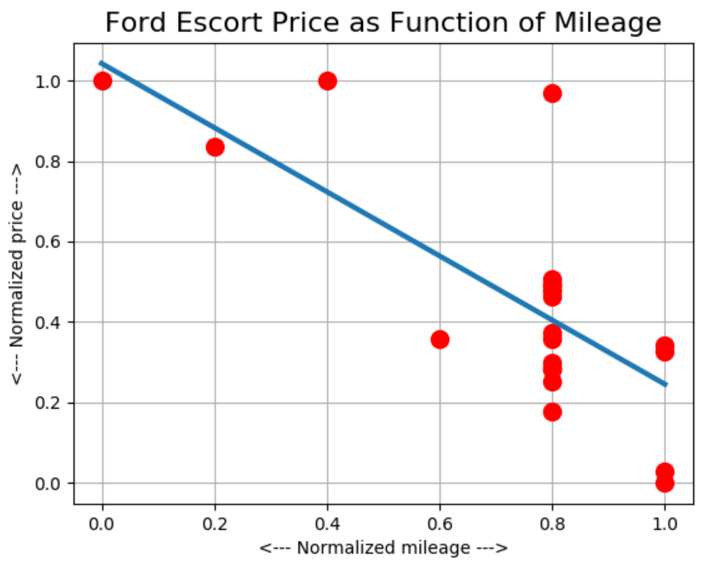
9 Example: Gradient descent to fit Ford data

We run `gradient_descent2` on the Ford problem, with a starting guess of `bm0=[b,m]=[0.5,0.0]`. Here is a sample of the results:

```
ford_gradient2:
  Python version: 3.6.8
  Given mileage x and price y for used Fords,
  seek (m,b) so that y=b+mx approximates the data.
  Use gradient descent2 to estimate best b and m.

it:   0 x: [0.5          0.        ] f(x) 2.5773498181818186 df(x) [-9.05163636  -1.34333243]
it:   1 x: [0.59051636   0.01343332] f(x) 1.954602113941928  df(x) [-4.61119726   0.68933256]
it:   2 x: [0.63662834   0.00654   ] f(x) 1.7798747361978844 df(x) [-2.63202845   1.55273919]
it:   3 x: [ 0.66294862 -0.00898739] f(x) 1.6955150280382623 df(x) [-1.74111328   1.90032506]
 ... ... more output ... ...
it: 117 x: [ 1.0471238  -0.78465017] f(x) 0.5783556887827288 df(x) [-0.04539271   0.09537134]
it: 118 x: [ 1.04757773 -0.78560388] f(x) 0.578245646967721  df(x) [-0.04415571   0.09277238]
it: 119 x: [ 1.04801929 -0.78653161] f(x) 0.5781415209296815 df(x) [-0.04295242   0.09024424]

  Step size ||dx|| less than dxtol =  0.001

  Norm of initial error is 2.57735
  Norm of total error is 0.578142
```

Listing 11: Output for gradient_descent2 on Ford data

Our parameters are `b=1.048, m = -0.786`. We plot the normalized data:



Linear formula to approximate the normalized Ford data

10 Computing Assignment #5

The file *moneyball_data.txt* contains 15 records. Each record lists a total payroll `pay` in the hundreds of millions of dollars, and a winning rate `win` (between 0 and 1) for a baseball team. We would like to construct a model that approximately predicts the winning percentage based on the payroll, assuming that more money means a higher win rate, of the form:

$$win = b + m * pay$$

Write a python program *hw5.py* that estimates `b` and `m` using the *gradient_descent2()* function. You might modify the code *ford_gradient2.py* to do this.

1. include a function `moneyball_data` to set up the moneyball data, split it into `pay` and `win` vectors, normalize both vectors to lie between 0 and 1, and return these as output;
2. include functions `moneyball_f` and `moneyball_df` to evaluate the error function and derivative;
3. set input and call `gradient_descent2`;
4. print the parameters `b` and `m` computed by the program, and the norm of the error `moneyball_f()` at these parameters;

You may need to adjust the starting values for `b` and `m` to get the calculation to converge. You should expect to need less than 1,000 iterations to get a result. Because in this case the data is very scattered, the gradient descent method might only be able to reduce your initial error norm by a moderate amount.

Email a copy of your program to Dr Schneier **mhs64@pitt.edu** before Wednesday, 16 October.