Week 3 Lab

The questions below are due on Sunday October 01, 2017; 11:00:00 PM.

You are not logged in.

If you are a current student, please Log In (https://introml.mit.edu/fall17/labs/lab03?loginaction=login) for full access to this page.

Lab 3 is divided into two parts:

1. Exploring gradient descent
2. Implementing gradient descent

Part 1 will require a checkoff with a staff member.

1) EXPLORING GRADIENT DESCENT

1.1) Explore gradient descent

We established that we are interested in the SVM objective function, which combines average hinge loss on the data with the norm of the weight vector. Our goal is to find θ, θ_0 to minimize that objective. We will start by studying general strategies for finding the minimum of a function. Generally speaking, unless the function is convex, it is very computationally difficult to find its global minimum. In machine learning, we will sometimes study convex objectives, and sometimes content ourselves with finding a local minimum (where the gradient is zero) that may not be a global minimum.

One way (there are many, much better ways, but this one is simple and computationally efficient in high dimensions and with lots of data) to find the minimum of a function is called *gradient descent*. The idea is that you start with an initial guess, x_0, and then move "downhill" in the direction of the gradient, to try another guess:

$$x_1 = x_0 - \alpha \nabla_x f(x_0)$$

where α is a "step size" parameter. You do this until x_i does not differ too much from x_{i+1}. It is guaranteed to find the minimum if the function is convex and if you set α to a small enough value.

We will try to get some intuition by trying it out. Functon `t1` performs gradient descent on a parabola:

$$f(x) = (2x + 3)^2$$

It takes optional (named) parameters

- `step_size` that defaults to 0.1 (keep this less than 1)
- `init_val` (the initial guess) that defaults to 0 (keep this in the interval [-3, 0])

All the functions we will be experimenting with perform gradient descent; they terminate after 1000 iterations or when the x value has changed by less than .00001 in subsequent iterations.

1. Write down the update rule that will be executed on every step, when performing gradient descent on this function. You may use `alpha` and `x` in your Python expression.

```
x = x+alpha*(4*(2*x + 3))
```

Ask for Help

Now play with it! Experiment with `step_size` and `init_val`.

In the question below, when you click Submit, it generates a plot of f in blue, and the history of x values you have tried in red. To see the plot you need to click the "Show/Hide Detailed Results" button. You can change the values for `step_size` or `init_val` and click Submit again. Note that all Submits get 100%, so you don't have to worry about scores.

```
1 def run():
2     return t1(step_size= 0.1, init_val = 0)
3
```

Ask for Help

2. What is a step size that makes it converge without oscillating?
3. What is a step size that makes it converge, but with oscillating?
4. What is a step size that makes it diverge?
5. What is a step size that makes it not diverge, but also not get within .1 of the optimum?
6. Is there a value of the inital value that will make it fail to converge to the optimum, no matter what the step size is? If so, what is it?

The function `t2` performs gradient descent on a more complicated function. It takes optional (named) parameters

- `step_size` that defaults to 0.1
- `init_val` that defaults to 0.

Experiment with them.

```
1 def run():
2     return t2(step_size= 0.1, init_val = 0)
3
```

Ask for Help

7. Is there a value of the initial value that will make `t2` fail to converge to the global optimum, no matter what the step size is? If so, what is it?

1.2) Put on your 2D glasses!

The procedure `t3` will test the same gradient descent method, but in two dimensions. It will generate two plots:

- The objective function is shown with dark blue corresponding to low values and yellow to high values and the trajectory of (x,y) values visited during the optimization drawn in red
- The value of the objective versus iteration

It takes optional (named) parameters

- `step_size` that defaults to 0.01 (note difference)
- `init_val` that defaults to [0, 0]

Experiment with them.

```
1 def run():
2     return t3(step_size= 0.01, init_val = [0.0, 0.0])
3
```

Ask for Help

8. What is a step size that makes it converge?
9. What is a step size that makes it diverge?
10. Is there a value of the initial value that will make `t3` fail to converge to the global optimum, no matter what the step size is? If so, what is it?

1.3) Adaptive step size

Procedure `t4` is like `t3`, but on every iteration through the data, it decreases the step size. So, the learning rate

on iteration t through the data is

$$\frac{\alpha}{\sqrt{t+1}}$$

It takes optional (named) parameters

- `init_step` (for α) that defaults to 0.1
- `init_val` that defaults to [0, 0]

Experiment with them.

```
def run():
    return t4(init_val = [0.0, 0.0], init_step = 0.1)

```

Ask for Help

11. What is an initial step size that makes it converge?
12. What is an initial step size that makes it diverge?
13. What are the advantages (if any) and disadvantages (if any) of using this adaptive step-size rule?

1.4) SVM objective

If you have not yet finished HW3, you can skip these last two questions and go straight to the checkoff. But if you have done HW3 it's worth staying around to think about this and talk to a staff member about it.

We can use gradient descent to do machine learning! Remember the SVM objective:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} L_h(y^{(i)}(\theta^T x^{(i)} + \theta_0)) + \lambda \|\theta\|^2$$

The test `t5` does gradient descent on that metric, on one-dimensional separable data. In one dimension, with offset, we have two parameters θ and θ_0.

- `lam` (for λ) that defaults to 0.001
- `init_step` (for α) that defaults to 0.1
- `max_iter` that defaults to 100
- `init_val` that defaults to [0, 0]

```
def run():
    return t5(lam = 0.001, init_step = 0.1,
              max_iter = 100, init_val = [0.0, 0.0])

```

Ask for Help

Try (at least) the following cases:

14. Initialize with parameter `init_val = [-1, 4]` and try `lam = 0, init_step = 0.5`. Explain what happens.
15. Using the same initialization, adjust `lam` so that it finds something much closer to the maximum margin separator.

1.5) Checkoff

Checkoff 1:

Have a check-off conversation with a staff member, to explain your answers.

Ask for Help | Ask for Checkoff

2) IMPLEMENTING GRADIENT DESCENT

In this section we will implement generic versions of gradient descent and apply these to the SVM Loss.

2.1) Gradient descent

We want to find the x that minimizes the value of $f(x)$, the *objective function* for an arbitrary scalar function f. The function f will be implemented as a Python function of one argument, that will be a numpy column vector. For efficiency, we will actually work with Python functions that return not just the value of f at $f(x)$ but also return the gradient vector at x, that is, $\nabla_x f(x)$.

We will now implement a generic gradient descent function, `gd`, that is given as arguments:

- `fdf`: a function whose input is an `x`, a column vector, and returns a tuple `(f, df)`, where `f` is a scalar and `df` is a column vector representing the gradient of `f` at `x`.
- `x0`: an initial value of x, `x0`, which is a column vector.
- `step_size_fn`: a function that is given the iteration index (an integer) and returns a step size. *
- `max_iter`: the number of iterations to perform

It returns a tuple:

- `x`: the value at the final step

- `fs`: the list of values of f found during all the iterations
- `xs`: the list of values of x found during all the iterations

Hint: This is a short function; our implementation is around 12 lines.

The test cases are:

```
import numpy as np

def rv(value_list):
    return np.array([value_list])

def cv(value_list):
    return np.transpose(rv(value_list))

def fdf1(x):
    return float((2 * x + 3)**2), 2 * 2 * (2 * x + 3)

def f(x):
    return(x - 2.) * (x - 3.) * (x + 3.) * (x + 1.)

def fdf2(v):
    x = float(v[0]); y = float(v[1])
    return f(x) + (x + y -1)**2,\
           cv([(-3. + x) * (-2. + x) * (1. + x) + \
               (-3. + x) * (-2. + x) * (3. + x) + \
               (-3. + x) * (1. + x) * (3. + x) + \
               (-2. + x) * (1. + x) * (3. + x) + \
               2 * (-1. + x + y),
               2 * (-1. + x + y)])
```

```
1 import numpy as np
2
3 def gd(fdf, x0, step_size_fn, max_iter):
4     x = x0; fs = []; xs = []
5     iters = 0
6     while (iters < max_iter):
7         fx, grad_fx = fdf(x)
8         xs.append(x); fs.append(fx)
9         x -= step_size_fn(iters)* grad_fx
10        iters+=1
11    return (x, fs, xs)
12
```

Ask for Help

2.2) Numerical Gradient

Getting the analytic gradient correct for complicated functions is tricky. A very handy method of verifying the analytic gradient or even subsituting for it is to estimate the gradient at a point by means of *finite differences*.

Given a function $f(x)$ that takes a column vector as its argument and returns a scalar value. In gradient descent, we will want to estimate its gradient at a particular $x0$.

The i^{th} component of $\nabla_x f(x0)$ can be estimated as

$$\frac{f(x0 + \delta^i) - f(x0 - \delta^i)}{2\delta}$$

where δ^i is a column vector whose i^{th} coordinate is δ, a small constant such as 0.001. Note that this is really just incrementing the i^{th} component of $x0$, $x0_i = x0_i + \delta$. Importantly, this has to be done for each value of i independently, each such computation (requiring 2 function evaluations) gives the i^{th} component of the gradient.

Implement this as a function `num_grad` that is given as argument the objective function `f` and a value of `delta` and returns a new function that takes an `x` (a column vector of parameters) and returns a gradient column vector.

You should make sure that your gradient function does not modify its input vector. Note that if you do:

```
temp_x = x
```

then `temp_x` is just another name for the same vector as `x` and changing an entry in one will change an entry in the other. So, you either need to copy `x.copy()` (which will make your code slow if `x` is a big matrix or array) or remember to change entries back (which may not always be practical). Caveat Emptor!

The test cases use the functions defined in the previous exercise.

```
1 import numpy as np
2
3 def num_grad(f, delta=0.001):
4     def grad_fx(x):
5         d = x.shape[0]
6         deltas = np.ones((d,1))*delta
7         fx_minusdelta = f(x-deltas) #np.apply_along_axis(f, 1, x-
8         fx_plusdelta = f(x+deltaas) #np.apply_along_axis(f, 1, x+
9         return np.apply_along_axis\
10                (lambda i: i/(2*delta), 1, fx_plusdelta - fx_minus
11     return grad_fx
12
13
```

Ask for Help

A faster (one function evaluation per entry), though sometimes less accurate, estimate is to use:

$$\frac{f(x0 + \delta^i) - f(x0)}{\delta}$$

for the i^{th} component of $\nabla_x f(x0)$.

2.3) Using the Numerical Gradient

Our normal gradient descent function takes a function that returns both a value and a gradient. Write a function `minimize` that takes a function that only computes a value (not a gradient) and which calls gradient descent to find a local minimum. All the other arguments and return values of `minimize` are the same as for `gd`. We have provided you with our version of `num_grad` and `gd`, you should not define them again.

Your definition of minimize should call `num_grad` exactly once. Of course, the function that `num_grad` returns will be called many times.

```
1 import numpy as np
2
3 def minimize(f, x0, step_size_fn, max_iter):
4     grad_x = num_grad(f)
5     def my_fdf(x):
6         return f(x),grad_x(x)
7     return gd(my_fdf, x0, step_size_fn, max_iter)
8
```

Ask for Help