Gradient Descent
ML_2022: Machine Learning

https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/gradient_/lab/gradient_lab.pdf

The gradient descent method seeks the lowest value of a function, taking a series of descending steps.

Gradient Descent for Optimization

Gradient descent allows us to approach the minimum value of a function.

- *Gradient descent can be more efficient than the direct least squares approach from linear algebra, especially if the data set is very large;*
- *The basic algorithm is very simple;*
- *The Playfair data shows that data normalization is vital;*
- *There are several parameters that will allow us to improve performance;*
- *Stochastic gradient descent works on a single component at a time;*
- *Mini-batch gradient descent works on several components at a time;*

1 Copying the data:

Each of the exercises will be carried out on a particular datafile. These datafiles are available on the *datasets* page at the class website:
https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/datasets/datasets.html

You might go ahead now and download them all:

- *playfair_data.txt*

2 Prepare for Exercise 1:

The lecture presented a simple version of the gradient descent method. Suppose that we start with the data from *playfair_data.txt*, for which x is the year (column 0) and y is the cost (ratio of column 1 / column 2), We wish to use gradient descent to determine a coefficient vector `c` so that

$$y \approx c_0 + c_1 x$$

Based on the simple description of gradient descent, the following code outlines how we might try to compute c:

```
X = np.c_ [ np.ones ( n ), data[:,0] ]
y = data[:,1] / data[:,2]

r = 0.5  # r = learning rate
iterations = 500
c = np.zeros(2)

for it in range(iterations):
  gradient_vector = (2/n) * X.T.dot(X.dot(c6) - y)
  c = c - r * gradient_vector

mse = np.sum ( ( np.dot ( X, c ) - y )**2 ) / n
print ( "  C = ", c )
print ( "  MSE = ", mse )
```

Listing 1: Basic gradient descent

Gradient descent is a simple idea; our values of n and d are not very large, so with luck we should get an answer. The values of `r` and `iterations` were left up to us, but we just copied the example from the text. Let's see what happens.

3 Exercise 1:

Write a program *exercise1.py* which carries out the following steps:

1. Read `data` from the file *playfair_data.txt*;
2. Set up `X` and `y` as appropriate for the Playfair data.
3. Run the simple gradient descent code above;
4. The results are probably bad, as you can see from the value of `mse`;
5. Try reducing the value of the learning rate `r` five times;
6. For each try, print `r`, `c`, and `mse`.

Were you able to find any value of `r` for which `mse` looked to be reasonable?

Values of `mse` which are huge, such as 10^{10}, or set to `nan`, are not reasonable. They are a symptom that the iteration did not converge, but *diverged* ("blew up"). If the value of `r` is very very small, your iteration might not blow up, but it probably won't get anywhere either, that is, the value of `mse` will not be very good.

4 Prepare for Exercise 2:

We have already seen that badly scaled data can cause problems for our iterations. The variable x is a year, so it's in the thousands. It will be worth trying to normalize our data and repeat the experiment.

Let's be careful about the normalization. We will create `X` and `y` as before, and then create normalized copies `Xn` and `yn`. Moreover, we will save the values `xmax`, `xmin`, `ymax`, and `ymin` that we used for normalization.

We will run our gradient descent code on the normalized data `Xn, yn`. Although almost every value of the learning rate failed for us in Exercise 1, we will start again with `r=0.5`, and hope that works this time!. Assuming the iteration runs well, we will call our resulting coefficient vector `cn`, to remind us that it goes with the normalized data.

To verify that `cn` provides a decent linear fit to the data, we plot the function

$$yn = cn_0 + cn_1\, xn$$

and compare it to our normalized data values.

Unfortunately, the normalized linear coefficients `cn` can't be used with our original data. It might be worth while if we could translate our `cn` results to a correspond set of coefficients `c` which can be used directly with our raw, unnormalized data. This will be our second goal in this exercise.

Because we saved the original data, and the minimum and maximum values that we used to normalize it, we can actually work out the corresponding coefficient vector for the unnormalized data, that is

$$y = c_0 + c_1 x$$

This is because we can start with the normalized equation, rewrite it in terms of the original data, and then regroup to get coefficients c_0 and c_1, as suggested by the following:

$$\begin{aligned} yn &= cn_0 + cn_1\, xn \\ \frac{y - ymin}{ymax - ymin} &= cn_0 + cn_1 * \frac{x - xmin}{xmax - xmin} \\ y &= ymin + (ymax - ymin)(cn_0 + cn_1 * \frac{x - xmin}{xmax - xmin}) \\ y &= \ldots \quad \text{(separate constant and linear coefficients)} \\ y &= c_0 + c_1 * x \end{aligned}$$

Once you have worked out the values of `c` you can make another plot, this time of the original data and the corresponding linear fit. If we do everything correctly, the plots will look identical, but the scales on the axes will no longer run from 0 to 1. This means our coefficients c represent a valid model for our unnormalized data, which is much more meaningful to us, even though the gradient descent method doesn't like to work with it directly.

5 Exercise 2:

Write a program *exercise2.py*, starting by copying `exercise1.py`.

1. Set `data` by reading from the file *playfair_data.txt*;
2. Set up `X` and `y` as appropriate for the Playfair data.
3. Create normalized data `Xn` and `yn`, saving the values of `xmin`, `xmax`, `ymin`, and `ymax`.
4. Note that, for both `X` and `Xn`, column 0 should be a vector of 1's. Only the next column needs to be normalized to create `Xn`.
5. Run the simple gradient descent code on the normalized data, with `r=0.5`;
6. The iteration should converge this time, giving you coefficients `cn`;
7. Plot the normalized data, and the linear fit function together;
8. Work out the formula that gives you the coefficient vector `c` for the original unnormalized data.
9. Plot the unnormalized data, and the linear fit function together.

6 Prepare for Exercise 3

Scaling issues can cause a gradient descent method to perform poorly. However, calculus tells us that, by the definition of the derivative, the function we are trying to reduce must decrease in the direction of the negative gradient, at least if we take a small enough step, that is, if the learning rate is small enough.

We also expect that, even if the iteration is proceeding well, we will need to take smaller and smaller steps as we approach the minimizer. However, in our first drafts of the gradient descent method, we tried to use a fixed value `r` for the learning rate. Perhaps we should pay more attention to this value. In particular, if we try a step using a given value of `r` and the function value goes up, we should cancel the step, reduce `r` and try again. If necessary, we should keep reducing `r` until the corresponding step produces a lower value of the function, (that is, the mean square error that we have been symbolizing by `mse`.

In the following exercise, we will try this idea. Instead of immediately accepting the next iterate, we will compute the value of `mse` that it produces. If this value is larger than our current value, we will reduce `r` and try again. Theoretically, this will guarantee that we will never encounter an increase in `mse`.

7 Exercise 3

Write a program *exercise3.py*. For this exercise, we don't want to worry about normalization, so we are looking at data that has a reasonable range.

1. Create the arrays `X` and `y` for this problem as follows:

```
n = 100
X = 2.0 * np.random.rand ( n, 1 )
y = 4.0 + 3.0 * X + 5.0 * np.random.rand ( n, 1 )
X = np.c_ [ np.ones ( n ), X ]
```

2. Initialize your gradient descent iteration as follows:

```
r = 1.0
c = np.random.rand ( 2, 1 )
mse = 1.0 / n * sum ( ( np.dot ( X, c ) - y )**2 )
iterations = 1000
rplot = np.zeros ( iterations + 1 )
rplot[0] = r
```

3. The original gradient descent loop looks something like this:

```
for it in range ( iterations ):
  gradient  = (2.0/n) * np.dot ( np.transpose ( X ), ( np.dot ( X, c ) - y ) )
  c = c - r * gradient
```

4. Rewrite the second line of the loop so that the new iterate is stored as the value `c_next`;
5. Still inside the gradient descent loop:
   - compute `mse_next` for this tentative value `c_next`.
   - If `mse` is less than `mse_next`, cut `r` in half.
   - Else `c` $\leftarrow$ `c_next` and `mse` $\leftarrow$ `mse_next`;
   - Set `rplot[it+1] = r`;
6. The iteration should converge, giving you `c` and `mse`;
7. Plot the data, and the linear fit function together;
8. Set `itplot = range ( 0, iterations+1)`, and plot `itplot` versus `rplot`.

Your first plot should show a good linear fit to the data. Your second plot should show how the value of `r` decreased as the iteration proceeded.

8 Prepare for Exercise 4

By controlling the value of the learning rate `r`, we were able in Exercise 3 to avoid having our gradient descent method blow up unexpectedly. But, as we have mentioned before, as the sizes of `r` and the gradient

vector decrease, our suggested stepsize can become so small that we are not changing the iterate at all. That's not a bad thing; it probably means we have converged. However, the way we wrote the loop, we are guaranteed to take the full 500 or 1000 iterations that we specified.

It would be more intelligent to check whether our iteration has stalled, and simply exit the loop early. It turns out that this can be done quite easily. In exercise 3, we have a value `c` and a new iterate `c_next`, which is about to replace `c`. But if our stepsize has dropped too low, then in fact `c` and `c_next` will be identical. We can check by computing $||c - c_{next}||_\infty$, that is, `np.max(np.abs(c-cnext))`. If this quantity is 0, we should exit the loop immediately.

9 Exercise 4

Write a program *exercise4.py*, starting from a copy of `exercise3.py`.

1. Set up the the same arrays `X` and `y` that you did for Exercise 3. (Note that these arrays won't actually contain the same values as before. Why not?)
2. The second line of the gradient descent loop defines a new coefficient vector `c_next`. It's possible that `r` or `gradient` is so small that `c_next` is actually identical to `c`. If that occurs, we want to `break`, that is, exit the loop (because we have done our best to reach the minimizer.
3. Therefore, just after the second line of the gradient descent loop, we will insert a test. Figure out an appropriate statement which will be true if the vectors `c` and `c_next` are identical. Hint: it's NOT as easy as `if ( c == c_next ):`!

```
c_next = c - r * gradient
if ( ??? ):
  print ( 'Early termination at iteration ', it )
  break
```

4. Run your code. You should expect to see that it terminates early. The resulting coefficient vector `c` will look reasonable, but probably will not be the same as the one you computed in Exercise 3. Why not?

Gradient descent, and its varieties of stochastic gradient descent and mini-batch gradient descent, are very powerful tools for optimization in machine learning. However, as you should have seen from these experiments, there are several ways this "simple" algorithm can go bad, and a number of adjustments we can make to improve its performance.