Topics in Artificial Intelligence: Machine Learning for Bioinformatics Applications (CSI 5180)
Fall 2019

Assignment 2
(Last modified on November 8, 2019)

Deadline: November 12, 18:00

Learning objective

- Further **understanding** of the gradient descent algorithm

Gradient descent for linear regression

Suppose that you have a set of data points in the plan, and you are trying to find a line that "fits" this set, meaning that it minimizes the average distance between the line and the set of data points. For example, on Figure 1, the red line fits the set of blue points.

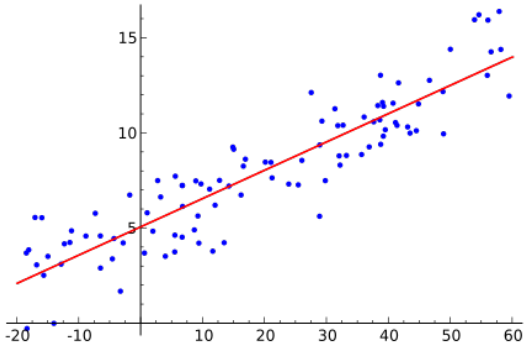

Figure 1: Linear regression (Source: Sewaqu https://commons.wikimedia.org/w/index.php?curid=11967659)

Of course, you could write a series of equations and solve them in order to find that line (the closed-form solution uses the normal equation). But there is another way: you can use algorithms that are common in machine learning, and get your program to "learn" from your data set.

In this assignment, we are going to implement one such algorithm, called *gradient descent*. We will first look at the case of a single variable, in which the solution we are looking for is a straight line in the plane, and then we will generalize our solution so that it can work with any number of variables. This means that our dataset can accumulate any number of informations (*features*) by data point, and we will find a linear equation that fits the set. This task is known a linear regression.

A practical example of this would be the prediction of the molecular activity (a real number) of a small compound. As a first approximation, we could be simply looking at the size of the molecule (the number of atoms, for instance). Collecting a large set of samples, we could build a dataset that gives the molecular activity of a molecule as a function if its size (i.e. one variable). We will then use our application to find a good line fit for this data set. We could then use our function to predict the activity of a new molecule given its size.

This may work to some extent, but we will soon realize that size alone isn't enough to predict the molecular activity with good accuracy (performance). A number of other features are important: hydrophobicity, net charge, molecular weight, isoelectric point, etc. So we can enrich our dataset with all of these variables (called *features*), and express the molecular

activity as a function of all this information. We will then use our application to find a good fit, and we will now be able to predict the activity of a new compound, given its set of features. Hopefully, that prediction will now be more accurate.

This approach has some serious limitations. One of the main one is that not everything is linear, so a linear solution will be a poor solution in many cases. But the goal of this assignment is not to study linear regression or gradient descent. We will simply be using the provided algorithms to implement our solution. You can easily find some information online; a good starting point, which was the inspiration for this assignment, is Andrew Ng's *Machine Learning* online course:

- https://www.coursera.org/learn/machine-learning

Gradient descent for one variable

In this first version of our application, we are working inside the plane. We are working with a set of sample data, called our *training examples*. We have a total of N training examples. Each example is a pair (x_i, y_i), where x_i is the input and y_i its label (the point (x_i, y_i) in the plane).

We are looking for a straight line that would best fit all of our data points. Remember that the equation for a line in the plane is of the form $y = ax + b$. We are looking for the line that would best fit our data points, that is, the best equation. We are going to call our function h_θ. In other words, we are looking for a function $h_\theta(x_i) = \theta_0 + \theta_1 \times x_i$.

We are calling the function h_θ the *hypothesis* function, since this is the function that is supposed to "explain" our data set (to fit it). We will start with some (probably quite bad) hypothesis function, and improve it over time.

In order to improve our current hypothesis, we need to measure how accurate it is. We are using a *cost function* $J(\theta_0, \theta_1)$ such as:

$$J(\theta_0, \theta_1) = \frac{1}{N} \sum_{i=1}^{N} (h_\theta(x_i) - y_i)^2$$

This cost function (mean squared error) tells us how "far" our hypothesis function is from the actual values of our training set. As we modify θ_0 and θ_1, the value of $J(\theta_0, \theta_1)$ increases or decreases.

The goal is to iteratively modify θ_0 and θ_1 in order to decrease the value of $J(\theta_0, \theta_1)$ (that is, in order to get a better hypothesis function). To guide us, we will use the derivative of $J(\theta_0, \theta_1)$, it gives us a "direction" in which to go to reduce the current value of J, as well as the magnitude of the change (small or large, slow or fast). We will use that derivative to iteratively reduce the value. As we approach a (local) minimum, the derivative will approach 0 and we will stop moving[1].

Our solution is to use the so-called *gradient descent* algorithm. This algorithm is as follows:

repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1), \text{for } j = 0 \text{ and } j = 1$$

}

In this algorithm, θ_0 and θ_1 **must be updated simultaneously**. Otherwise, the result would be mathematically ill-defined. The algorithm uses a *step size* α, which will control how much correction to θ_0 and θ_1 we will provide at each step. Loosely speaking, "convergence" means that new iterations of the algorithm do not improve the solution very much anymore.

We need the partial derivative J with respect to θ_0 and θ_1. They are as follows:

$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{2}{N} \sum_{i=1}^{N} (h_\theta(x_i) - y_i)$

and

$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{2}{N} \sum_{i=1}^{N} (x_i \times (h_\theta(x_i) - y_i))$

Thus, our gradient descent algorithm for linear regression can be written:

[1] Note that in our case, the error function happens to be a *convex* function (a "bowl shaped" function) and thus has a single minimum. If that wasn't the case, this approach could lead us to a local minimum, which may not be what we want.

repeat until convergence: {

$$\theta_0 := \theta_0 - \alpha \frac{2}{N} \sum_{i=1}^{N} (h_\theta(x_i) - y_i)$$

$$\theta_1 := \theta_1 - \alpha \frac{2}{N} \sum_{i=1}^{N} (x_i \times (h_\theta(x_i) - y_i))$$

}

Implementation

For the first part, there will be three (3) Python programs:

- **model.py** declares a class called **LinearRegression**.
- **a2q11.py** is a solution to question 1.1.
- **a2q12.py** is a solution to question 1.2.

All three sub-questions will be using the class **LinearRegression**. For instance, in **a2q11.py** and **a2q12.py**, you will include the following directive:

```
from model import LinearRegression
```

An object of the class **LinearRegression** is created as follows:

```
model = LinearRegression()
```

In our implementation, we will initialize the values of θ_0 and θ_1 to 0 for any linear regression, and the method **fit** will take two arguments, α as described above, as well as the number of steps to run, instead of "until convergence".

Question 1.1 (4 points)

We will first implement the methods of the class **LinearRegression**.

- The constructor has no other parameters, other than **self**.
- The class has a method **addSample**, which is used to an example, **x**, and its value, **y**. In order to add **N** examples to a **LinearRegression**, there will be **N** calls to the method **addSample**.
- Once all the examples have been provided, the method **fit** can be called. That method has two parameters: the α to use, and the number of iterations to be performed during that call of the method.
- **getHypothesis** is an instance method receiving an example as input and returning the (value) value of the hypothesis evaluated on the given example, in other words $h_\theta(x)$.
- **__str(self)__** returns a string representation of the hypothesis function represented by the **LinearRegression** object. E.g.:

  ```
  4.03 + 4.09 x
  ```

- **getLoss** returns the mean squared error for the current set of examples and the current values of θ_0 and θ_1.
- **getTheta0** returns the value of θ_0.
- **getTheta1** returns the value of θ_1.
- **getIteration** returns the current number of iterations. This value is incremented by 1 for each step performed by the method **fit**. Specifically, after a call **model.fit(0.003, 100)**, **model.getIteration()** should return 100, after a second call, **model.fit(0.003, 100)**, **model.getIteration()** should return 200, etc.
- **getSamples** returns the list of all the examples.
- **getValues** returns the list of all the labels. The order of the elements is the same as for **getSamples**.

In order to test our implementation, we will first use a trivial set of examples made of 1,000 values on the line $y = x$. We will select the values from $x = 0$ to $x = 999$.

In the Python program, **a2q11.py**, implement the method **setLine** with the following characteristics.

- Creates a **LinearRegression** object.
- Iterates the gradient descent a total of 5,000 times. We will do this by performing gradient descent 100 times with a small positive $\alpha = 0.000000003$ and **numOfSteps** set to **100**, looped **50** times. At each iteration of the loop, you should print the current value of the hypothesis and cost function.
- **a2q11.py** calls **setLine**.

If our implementation is correct, the hypothesis function should tend toward the line $y = x$. Your console output for the first few iterations might look like this.

```
> python a2q11.py
Current hypothesis: 0.00 + 0.00 x, cost = 332833.5000
Current hypothesis: 0.00 + 0.18 x, cost = 223149.4956
Current hypothesis: 0.00 + 0.33 x, cost = 149611.4345
...
Current hypothesis: 0.00 + 1.00 x, cost = 0.0010
Current hypothesis: 0.00 + 1.00 x, cost = 0.0007
```

Figures 2, 3 and 4 show this information visually (see bonus question below). The line in red is the current hypothesis. As you can see, initially the red line is $y = 0$, our starting point, and the more we iterate, the closer if gets to $y = x$.

Question 1.2 (1 point)

The previous test was a start, but not particularly interesting as our set of points already define a line to begin with, so we will generate a "randomish" line. To generate random numbers, you can use **numpy.random.rand** and **numpy.random.randn**.

You will then need to figure out how to scale this number to sample from the intervals specified below.

The goal is to provide the implementation of the method **randomLine** for the Python program **a2q12**. This method should do the following.

- Creates a **LinearRegression** object.
- Generate a random line of the form $y = ax+b$, where a is randomly sampled from the interval $[-5, 10]$ and b is randomly sampled from the interval $[-5, 5]$.
- Generate 500 random points that satisfy the following conditions. The x value must be sampled randomly from $[-4, 6]$ and the corresponding y value has Gaussian noise added to it, normal distribution with mean 0.0 and variance 1.0.
- Finally, you have to find a number of iterations and a value for α (hint: think small, think positive) that works well with these randomly generated sets, and then perform gradient descent the same way as before.
- The Python program **a2q12** calls **randomLine()**.

```
> python a2q12
Target: a = -1.0, b = 3.6
Current hypothesis: 0.00 + 0.00 x, cost = 43.9785
Current hypothesis: 0.50 - 0.74 x, cost = 1.8136
Current hypothesis: 0.98 - 0.78 x, cost = 1.5835
...
Current hypothesis: 3.42 - 1.03 x, cost = 1.0452
Current hypothesis: 3.42 - 1.03 x, cost = 1.0452
```

Given that we added Gaussian noise to every data point with **mean = 0.0** and **variance = 1.0**. It is not too surprising that residual mean squared error converges to 1.0!

Figures 5, 6 and 7 show the system after different number of iterations on a sample run.

Bonus (2 points)

Make all the necessary changes to your implementation so that method **fit** displays the examples and the current hypothesis as shown in Figures reffig:demo1 to 7. Store this implementation in a subdirectory called **bonus**. You might actually find the graphic representation useful for debugging purposes.

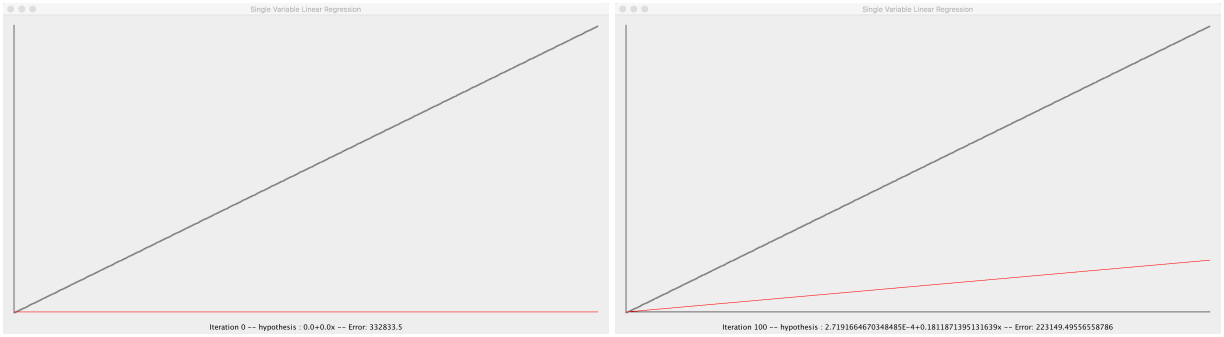

Figure 2: Initial situation, and after 100 iterations.

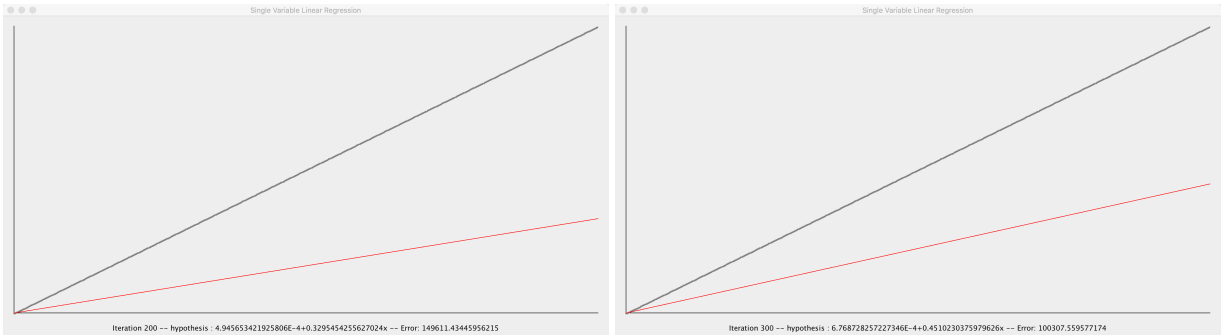

Figure 3: After 200 and 300 iterations.

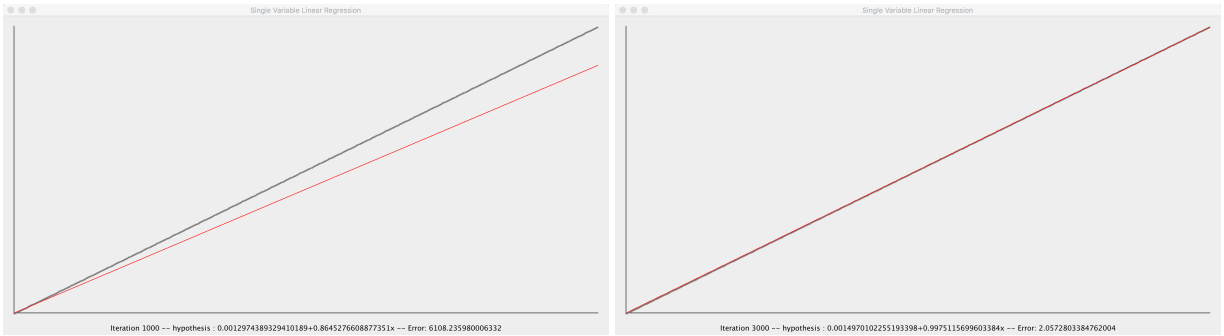

Figure 4: After 1000 and 3000 iterations.

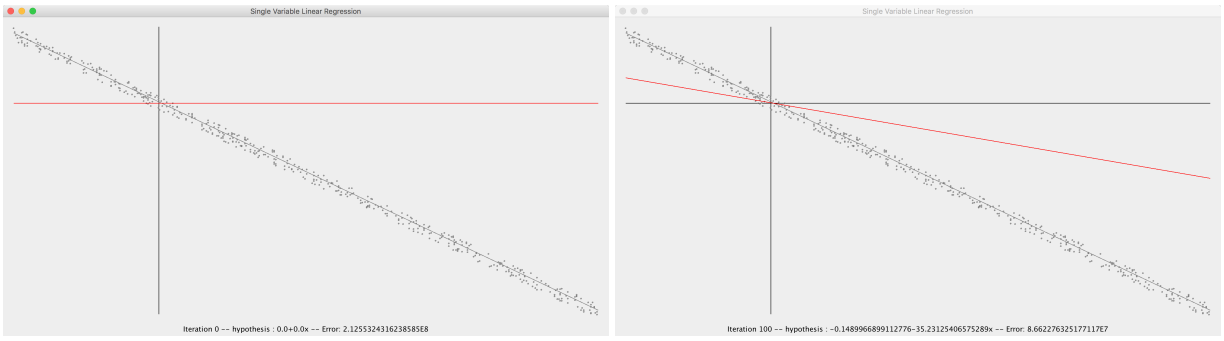

Figure 5: Initial situation, and after 100 iterations.

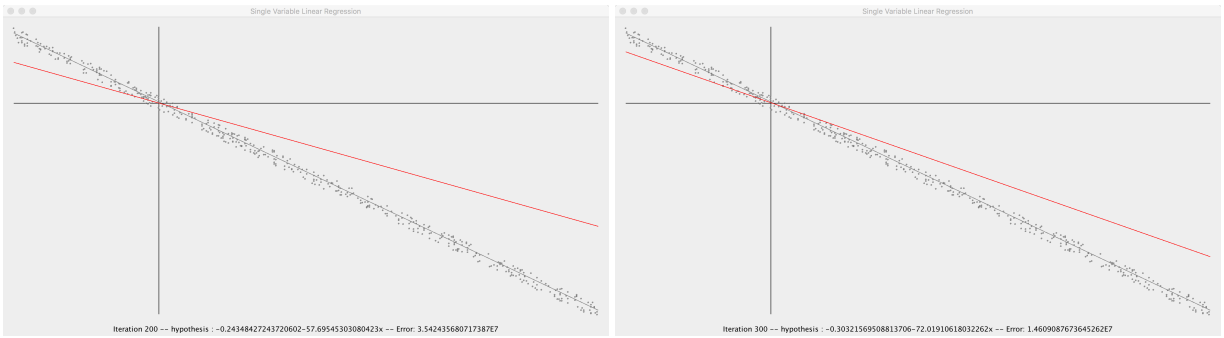

Figure 6: After 200 and 300 iterations.

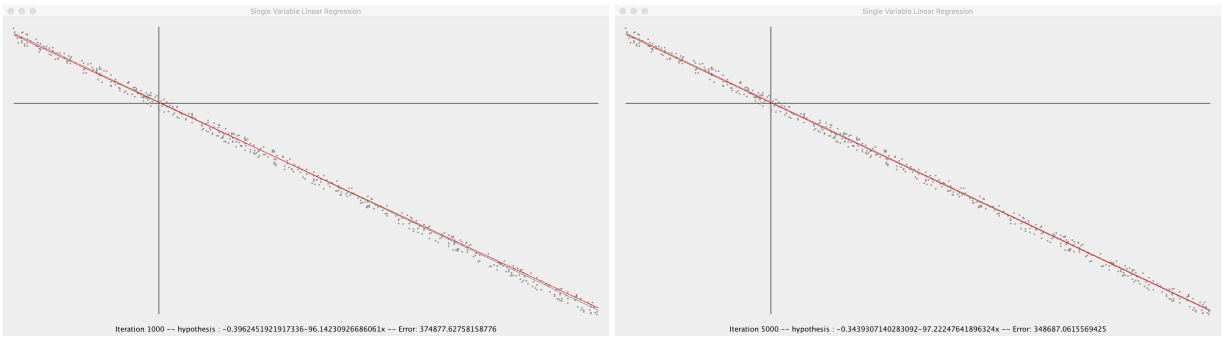

Figure 7: After 1000 and 5000 iterations.

Generalization: multivariate linear regression

If we have more than one feature (the "x") before, we can use the following hypothesis function: assume that we have D features $x^{(1)}, x^{(2)}, \ldots, x^{(D)}$, the new hypothesis function is:

$$h_\theta(x) = \theta_0 + \theta_1 x^{(1)} + \theta_2 x^{(2)} + \theta_3 x^{(3)} + \cdots + \theta_D x^{(D)}$$

Notations:

$$\begin{aligned} x_i^{(j)} &= \text{value of feature } j \text{ in the } i^{th} \text{ training example} \\ x_i &= \text{the input (features) of the } i^{th} \text{ training example} \\ N &= \text{the number of training examples} \\ D &= \text{the number of features} \end{aligned}$$

For convenience, we are adding a $D+1$ "feature" $x^{(0)} = 1$ (that is, $\forall i \in [1, \ldots, N], x_i^{(0)} = 1$). The hypothesis function $h_\theta(x)$ can now be rewritten:

$$h_\theta(x) = \theta_0 x^{(0)} + \theta_1 x^{(1)} + \theta_2 x^{(2)} + \theta_3 x^{(3)} + \cdots + \theta_D x^{(D)}$$

Gradient Descent for Multiple Variables

The new cost function is:

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{N} \sum_{i=1}^{N} \left(h_\theta(x_i) - y_i\right)^2$$

The gradient descent algorithm becomes:

repeat until convergence: $\{$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, \ldots, \theta_D)$$

for $j \in [0, \ldots, D]$ (update simultaneously)

$\}$

Which is

repeat until convergence: $\{$

$$\begin{aligned} \theta_0 &:= \theta_0 - \alpha \frac{2}{N} \sum_{i=1}^{N} x_i^{(0)} \times \left(h_\theta(x_i) - y_i\right) \\ \theta_1 &:= \theta_1 - \alpha \frac{2}{N} \sum_{i=1}^{N} x_i^{(1)} \times \left(h_\theta(x_i) - y_i\right) \\ \theta_2 &:= \theta_2 - \alpha \frac{2}{N} \sum_{i=1}^{N} x_i^{(2)} \times \left(h_\theta(x_i) - y_i\right) \\ &\cdots \end{aligned}$$

$\}$

that is,

repeat until convergence: $\{$

$$\theta_j := \theta_j - \alpha \frac{2}{N} \sum_{i=1}^{N} x_i^{(j)} \times \left(h_\theta(x_i) - y_i\right)$$

for $j \in [0, \ldots, D]$ (update simultaneously)

$\}$

Scaling, bias term, and all the good things in life

In the lecture on **Feature Engineering and Data Imputation**, we discussed the need to scale the values of our features. We considered two specific approaches, **normalization** and **standardization**.

- http://www.site.uottawa.ca/~turcotte/teaching/csi-5180/lectures/04/04/slides.pdf

On screen 20 (out of 51), I was saying the following:

> Many learning algorithms work best if the numerical values of the features have a similar range of values, say [-1,1] or [0,1]. Namely, the optimization (say **gradient descent**) may converge more rapidly.

Likewise, in the lecture on **Gradient Descent**, on screen 36 (out of 52), I was saying the following:

> Literature suggests that the algorithm might take more time to converge if the features are on different scales.

- http://www.site.uottawa.ca/~turcotte/teaching/csi-5180/lectures/04/03/slides.pdf

When I published the initial version of this assignment, for questions 2.2 and 2.3, I was suggesting that the values for each attribute would be in the range $[50, 4000]$. As you can below, in the gradient descent, each $(h_\theta(x_i) - y_i)$ is multiplied by $x_i^{(j)}$. For $j \in [1..D]$, this means multiplying "on average" each term by 1,975 ($= \frac{4{,}000-50}{2}$), whereas for $j = 0$, all the values, $x_i^{(j)}$, are 1.0. Consequently the value to be subtracted from θ_0 is proportional to $\alpha \times \frac{2 \times N \times 1.0}{N} \times \Delta = \alpha \times 2 \times \Delta$, where Δ is the sum of the differences. For all the other cases, i.e. $j \in [1..D]$, the value to be subtracted from θ_j is proportional to $\alpha \times 2 \times 1,975 \times \Delta$. Which means that the convergence of θ_0 would 2,000 times slower for this particular example!

repeat until convergence: {

$$\theta_0 := \theta_0 - \alpha \frac{2}{N} \sum_{i=1}^{N} x_i^{(0)} \times (h_\theta(x_i) - y_i)$$

$$\theta_1 := \theta_1 - \alpha \frac{2}{N} \sum_{i=1}^{N} x_i^{(1)} \times (h_\theta(x_i) - y_i)$$

$$\theta_2 := \theta_2 - \alpha \frac{2}{N} \sum_{i=1}^{N} x_i^{(2)} \times (h_\theta(x_i) - y_i)$$

$\ldots$

}

Here is what was happening when I was running my program. As you can see, the value of θ_0 does not seem to move!

```
> python a2q22.py
Target: -43.16 - 91.69 x_1 + 10.38 x_2
Current hypothesis: 0.00 + 0.00 x_1 + 0.00 x_2, cost = 41355220844.1388
Current hypothesis: -0.02 - 91.67 x_1 + 10.34 x_2, cost = 2306.3497
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4308
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4298
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4295
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4291
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4288
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4284
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4280
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4277
Current hypothesis: -0.02 - 91.70 x_1 + 10.37 x_2, cost = 370.4273
```

To circumvent this problem, we could scale the values of each attribute. However, given that our model is linear, the simplest thing to do seems to use values for our attributes that are proportional to 1.0 (the value of $x_i^{(0)}$).

Implementation

We will generalize our implementation from Question 1 to perform multivariate linear regression. You will have to modify your **LinearRegression** class to support inputs that are vectors of doubles, instead of single numbers. You will similarly have to update the gradient descent method itself, as well as all other necessary helper methods to support higher dimension inputs. We will then proceed with three tests below.

- **model.py** declares a class called **LinearRegression (2 points)**
- **a2q21.py** is the solution to the question **2.1**
- **a2q22.py** is the solution to the question **2.2**
- **a2q23.py** is the solution to the question **2.3**

Question 2.1 (1 points)

We will first test with a fixed plane (i.e. 2-dimensional input, or 2 features). Your **setPlane** method should do the following.

1. Create a **LinearRegression** object.
2. Add the points $((x, 2x), 5x)$ and $((2x, x), 4x)$ for $0 \leq x \leq 999$. Note that these points all satisfy the equation $z = x+2y$.
3. As before, run gradient descent with $\alpha = 0.000000003$ and number of steps set to 1000, a total of 10 times, and print the hypothesis and cost at each iteration.

Here is the output of my program:

```
> python a2q21.py
Current hypothesis: 0.00 + 0.00 x_1 + 0.00 x_2, cost = 6823086.7500
Current hypothesis: 0.00 + 1.18 x_1 + 1.82 x_2, cost = 11283.6859
Current hypothesis: 0.00 + 1.07 x_1 + 1.93 x_2, cost = 1530.1269
Current hypothesis: 0.00 + 1.02 x_1 + 1.98 x_2, cost = 207.4950
Current hypothesis: 0.00 + 1.01 x_1 + 1.99 x_2, cost = 28.1377
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 3.8156
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.5174
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.0702
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.0095
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.0013
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.0002
```

Since there is no noise, we are able to obtain an **MSE** close to zero (using a more aggressive learning rate and letting the algorithm run longer, I obtained an **MSE** of 0.0000005472).

Question 2.2 (1 point)

As in part 1, we will now move on to a plane with randomized points. Your **randomPlane** method should do the following.

- Create a **LinearRegression** object.
- Sample random coefficients $a, b, c \in [-100, 100]$. The plane we are aiming for is $y = ax^{(1)} + bx^{(2)} + c$.
- Now add 5000 points that satisfy the following conditions. The input $(x^{(1)}, x^{(2)})$ should be sampled with $x^{(j)} \in [0, 1]$. We set the noise this time to 20, so that for any given $(x^{(1)}, x^{(2)})$, we should have $y = ax^{(1)} + bx^{(2)} + c + \delta$, where δ is randomly sampled from $[-20, 20]$.
- As before, pick sensible values for α and a number of steps, and print the current hypothesis, current cost, and plane being aimed for at each iteration.

Question 2.3 (1 point)

We now generalize the previous work to generate a random equation for any given number of dimensions. The method **randomDimension** should take an integer D as an argument, specifying the required dimension D for the input vector, and the main program will test this with $D = 5$. Implement the method to do the following.

- Create a LinearRegression object with D features and 5000 examples. An input vector will be of the form $(x^{(1)}, x^{(2)}, \ldots, x^{(D)})$.
- We generate a random equation to aim for as follows. We randomly sample coefficients $t_0, t_1, \ldots, t_n$ from $[-100, 100]$. The equation we are modelling is

$$r = t_0 + \sum_{j=1}^{D} t_j x^{(j)} = t_0 + t_1 x^{(1)} + t_2 x^{(2)} + \ldots + t_D x^{(D)}. \tag{1}$$

- Add 5000 points that satisfy the following conditions. For each input vector $(x^{(1)}, x^{(2)}, \ldots, x^{(D)})$, sample $x^{(j)}$ randomly from $[0.0, 1.0]$. Then, let the result r be as in Equation 1, plus or minus a noise of 20. The value that is added is then $((x^{(1)}, x^{(2)}, \ldots, x^{(D)}), r)$.
- Again, choose sensible values for α and a number of steps, and print the current hypothesis, current cost, and equation being aimed for at each iteration.

Do not worry about the execution time. In particular, there is no need to create a vectorized implementation.

Files

You must hand in a zip file containing the following files.

- Two sub-directories, **A2Q1** and **A2Q2**.
  - **A2Q1** contains:
    - **model.py**, **a2q11.py**, **a2q12.py**
    - **bonus** (optional)
  - **A2Q2** contains:
    - **model.py**, **a2q21.py**, **a2q22.py**, **a2q23.py**
- **report.pdf** - a short report explaining your choices and any shortcomings of your implementation.

Last Modified: November 8, 2019

Sample output for setLine

```
> python a2q11.py
Current hypothesis: 0.00 + 0.00 x, cost = 332833.5000
Current hypothesis: 0.00 + 0.18 x, cost = 223149.4956
Current hypothesis: 0.00 + 0.33 x, cost = 149611.4345
Current hypothesis: 0.00 + 0.45 x, cost = 100307.5596
Current hypothesis: 0.00 + 0.55 x, cost = 67251.5877
Current hypothesis: 0.00 + 0.63 x, cost = 45089.0847
Current hypothesis: 0.00 + 0.70 x, cost = 30230.1496
Current hypothesis: 0.00 + 0.75 x, cost = 20267.9197
Current hypothesis: 0.00 + 0.80 x, cost = 13588.7044
Current hypothesis: 0.00 + 0.83 x, cost = 9110.5989
Current hypothesis: 0.00 + 0.86 x, cost = 6108.2360
Current hypothesis: 0.00 + 0.89 x, cost = 4095.2902
Current hypothesis: 0.00 + 0.91 x, cost = 2745.7030
Current hypothesis: 0.00 + 0.93 x, cost = 1840.8671
Current hypothesis: 0.00 + 0.94 x, cost = 1234.2164
Current hypothesis: 0.00 + 0.95 x, cost = 827.4851
Current hypothesis: 0.00 + 0.96 x, cost = 554.7906
Current hypothesis: 0.00 + 0.97 x, cost = 371.9615
Current hypothesis: 0.00 + 0.97 x, cost = 249.3830
Current hypothesis: 0.00 + 0.98 x, cost = 167.1998
Current hypothesis: 0.00 + 0.98 x, cost = 112.0997
Current hypothesis: 0.00 + 0.98 x, cost = 75.1577
Current hypothesis: 0.00 + 0.99 x, cost = 50.3898
Current hypothesis: 0.00 + 0.99 x, cost = 33.7840
Current hypothesis: 0.00 + 0.99 x, cost = 22.6506
Current hypothesis: 0.00 + 0.99 x, cost = 15.1862
Current hypothesis: 0.00 + 0.99 x, cost = 10.1816
Current hypothesis: 0.00 + 1.00 x, cost = 6.8263
Current hypothesis: 0.00 + 1.00 x, cost = 4.5767
Current hypothesis: 0.00 + 1.00 x, cost = 3.0685
Current hypothesis: 0.00 + 1.00 x, cost = 2.0573
Current hypothesis: 0.00 + 1.00 x, cost = 1.3793
Current hypothesis: 0.00 + 1.00 x, cost = 0.9248
Current hypothesis: 0.00 + 1.00 x, cost = 0.6200
Current hypothesis: 0.00 + 1.00 x, cost = 0.4157
Current hypothesis: 0.00 + 1.00 x, cost = 0.2787
Current hypothesis: 0.00 + 1.00 x, cost = 0.1869
Current hypothesis: 0.00 + 1.00 x, cost = 0.1253
Current hypothesis: 0.00 + 1.00 x, cost = 0.0840
Current hypothesis: 0.00 + 1.00 x, cost = 0.0563
Current hypothesis: 0.00 + 1.00 x, cost = 0.0378
Current hypothesis: 0.00 + 1.00 x, cost = 0.0253
Current hypothesis: 0.00 + 1.00 x, cost = 0.0170
Current hypothesis: 0.00 + 1.00 x, cost = 0.0114
Current hypothesis: 0.00 + 1.00 x, cost = 0.0076
Current hypothesis: 0.00 + 1.00 x, cost = 0.0051
Current hypothesis: 0.00 + 1.00 x, cost = 0.0034
Current hypothesis: 0.00 + 1.00 x, cost = 0.0023
Current hypothesis: 0.00 + 1.00 x, cost = 0.0015
Current hypothesis: 0.00 + 1.00 x, cost = 0.0010
Current hypothesis: 0.00 + 1.00 x, cost = 0.0007
```

```
>  python a2q21.py
Current hypothesis: 0.00 + 0.00 x_1 + 0.00 x_2, cost = 6823086.7500
Current hypothesis: 0.00 + 1.18 x_1 + 1.82 x_2, cost = 11283.6859
Current hypothesis: 0.00 + 1.07 x_1 + 1.93 x_2, cost = 1530.1269
Current hypothesis: 0.00 + 1.02 x_1 + 1.98 x_2, cost = 207.4950
Current hypothesis: 0.00 + 1.01 x_1 + 1.99 x_2, cost = 28.1377
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 3.8156
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.5174
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.0702
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.0095
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.0013
Current hypothesis: 0.00 + 1.00 x_1 + 2.00 x_2, cost = 0.0002
```

Object-oriented programming in Python

For those of you that are new to Python, here is a simple pair of programs having a lot in common with the programs for this assignment.

- collection.py

The file **collection.py** declares a class called **Bag**. A **Bag** is simply a collection of objects. It has method **add** for adding an element at the end of the collection, a method **get** to retrieve the element stored at a given location, as well as a method **size** that returns the number of elements in the data structure. As you can see, this class has a lot in common with our class **LinearRegression**.

```
class Bag:

    def __init__(self):
        self.data = []

    def add(self, item):
        self.data.append(item)

    def size(self):
        return len(self.data)

    def get(self, pos):
        return self.data[pos]

    def __str__(self):
        return "Bag: [" + ','.join(map(str,self.data)) + "]"
```

- main.py

The file **main.py** provides a simple **main** program for our application. It imports the name **Bag**, creates two **bags**, adds elements to them, prints the size, access elements, and prints their content (implicit call to **__str()__**). This program has a lot in common with **a2q11.py** and similar programs from our assignment. It shows how to import a name, create objects, and interact with them.

```
from collection import Bag

a = Bag()
b = Bag()

print(a)
print(b)

a.add(1)

b.add('a')
b.add('b')

print(a)
print(b)

print(a.size())
print(b.size())

print(b.get(0))
print(b.get(1))
```

Hope this helps and let me know if you have questions about these.