ITI 1121. Introduction to Computing II
Winter 2018

Assignment 1

(Last modified on 27 janvier 2018)

Deadline: February 2, 11:30 pm

[PDF]

Learning objectives

- Applying basic object oriented programming concepts
- Using arrays to store information
- Using reference variables
- Editing, compiling and running Java programs
- Raising awareness concerning the university policies for academic fraud

Introduction

Artificial Intelligence is a "hot" topic : every day, you can read about it in the news. These articles are usually about machine learning, which is one of the branches of artificial intelligence. It is in fact a domain in which Canada is a leader. Researchers in machine learning are developing new algorithms that can learn from data. In practice, these are algorithms that are able to find good values for the parameters of models, given a set of sample data as input. These models are then used to predict values for new input data.

For this assignment, we are going to look at one such model, called **linear regression**. It is a simple, well established model, used e.g. in statistics. We are going to write a program that will iteratively find good values for a linear model, given sample data as input.

This assignment, an in particular all the code for the equations below, was designed based on Andrew Ng's *Machine Learning* online course :

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

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.

Of course, you could write a series of equations and solve them in order to find that line. But there is another way : you can use some algorithms that are common in machine learning, and get your program to "learn" your data set and find a solution using some kind of guided trial and error method.

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.

A practical example of this would be the housing market. As a first approximation, we could be simply looking at the house's size. Collecting a large set of sample sales in our area, we could build a dataset that gives the selling price of a house 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 price at which a house will sale, given its size.

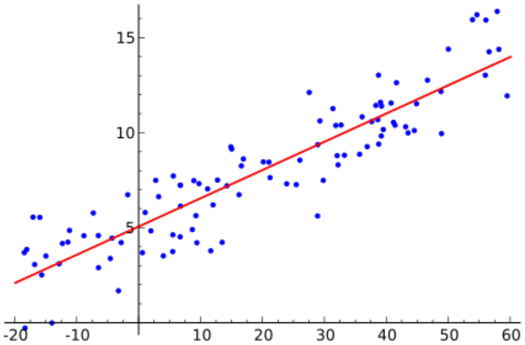


FIGURE 1 – Linear regression (source : Sewaqu https://commons.wikimedia.org/w/index.php?curid=11967659.

This may work to some extent, but we will soon realize that size alone isn't enough to predict the sale price with accuracy. A number of other factors are important : the number of rooms, the neighbourhood of the house, the condition of the building, the year of construction, having a finished basement, a garage etc. So we can enrich our dataset with all of these variables (called *features*), and express the sale price 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 price at which a house will sale, given the set of features of that house. 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 will learn more about these topics later in your studies if you enroll in the corresponding courses. If you are curious about this topic, 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 at 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 m training samples. Each sample is a pair (x_i, y_i), where x_i is the sample and y_i its value (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 $h_\theta(x_i) = \theta_0 + \theta_1(x_i)$.

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

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

This cost functions 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, if we take the derivative of $J(\theta_0, \theta_1)$, it gives us a "direction" in which to go to reduce the current value of J. We will use that derivative to iteratively reduce the value. As we approach a (local) minima, 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 :

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

$$
\begin{aligned}
&\text{repeat until convergence : } \{ \\
&\qquad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1), \text{for } j = 0 \text{ and } j = 1 \\
&\qquad \}
\end{aligned}
$$

In this algorithm, θ_0 and θ_1 must be updated simultaneously. The algorithm uses a *step size* α which will controls 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 of J for θ_0 and θ_1. They are as follows :

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

and

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

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

$$
\begin{aligned}
&\text{repeat until convergence : } \{ \\
&\qquad \theta_0 := \theta_0 - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i) \\
&\qquad \theta_1 := \theta_1 - \alpha \frac{2}{m} \sum_{i=1}^{m} ((h_\theta(x_i) - y_i)x_i) \\
&\qquad \}
\end{aligned}
$$

Implementation

For the first part, there will be three classes, **Display** (which is given to you), **LinearRegression** which will be your implementation of the algorithm, and **Assignment** which will be runnable tests of the algorithm.

The **Display** class will allow us to visualize the lines and points graphically. You do not need to understand the details of this class, but simply use it as required. We will talk about user interfaces later in the semester.

If **linearRegression** is a reference variable, referencing an instance of the class **LinearRegression**, you can create an instance of the class **Display** as follows :

```
Display graph;
graph = new Display(linearRegression);
```

The update the graph after you have modified the parameters of the model, you simply call the method **update()** of the object referenced by the reference variable **graph** :

```
graph.update();
```

In our implementation, we will choose the initial values $\theta_0 = \theta_1 = 0$ for any linear regression, and our gradient-Descent method will take two arguments - an α as described above, as well as an integer number of steps to run (instead of "until convergence").

Question 1.1

We will first implement the algorithm in the class **LinearRegression**. The constructor of this class receives as a parameter the number of sample points with which it will work. These samples are then provided one by one using the method **addSample**. Once all the samples are provided, the method **gradientDescent** can be called. That method received two parameters : the step α to use, and the number of iterations to be performed during that call of the method (the method will itself be called several times). The class provides a number of other methods that are required for its own purpose or by other classes that you will have to fill in the details of.

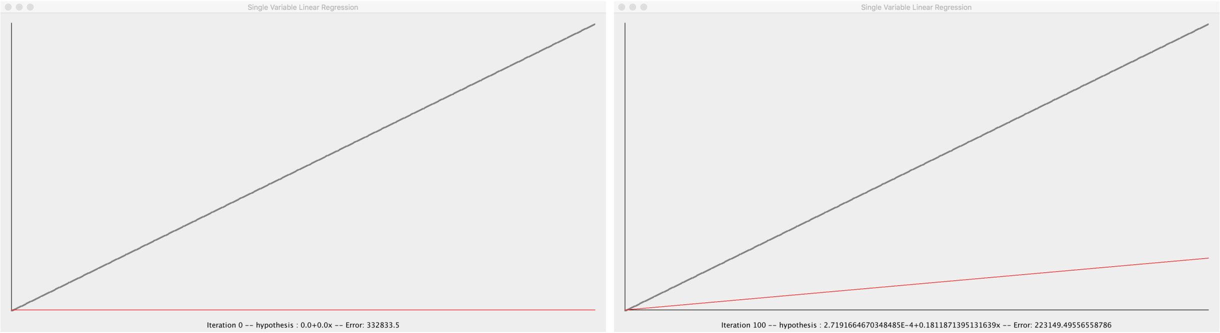
In order to test our implementation, we will first use a trivial sample set made of 1,000 values on the line $y = x$; we will select the values from $x = 0$ to $x = 999$. The method **setLine** of the class **Assignment** is what we will use for this. This method should do the following.

- Create a LinearRegression object of the appropriate size to create a line consisting of the points (i, i) for $0 \leq i \leq 999$, and create a corresponding display object.
- Iterate the algorithm 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 update the graph and print the current value of the hypothesis and cost function.

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.

```
> java Assignment
setLine
Current hypothesis: 0.0+0.0x
Current cost: 332833.5
Press return to continue....

Current hypothesis: 0.0012974389329410189+0.8645276608877351x
Current cost: 6108.235980006332
Press return to continue....

Current hypothesis: 0.001473201544681895+0.9816455619876667x
Current cost: 112.09973445673143
Press return to continue....
```

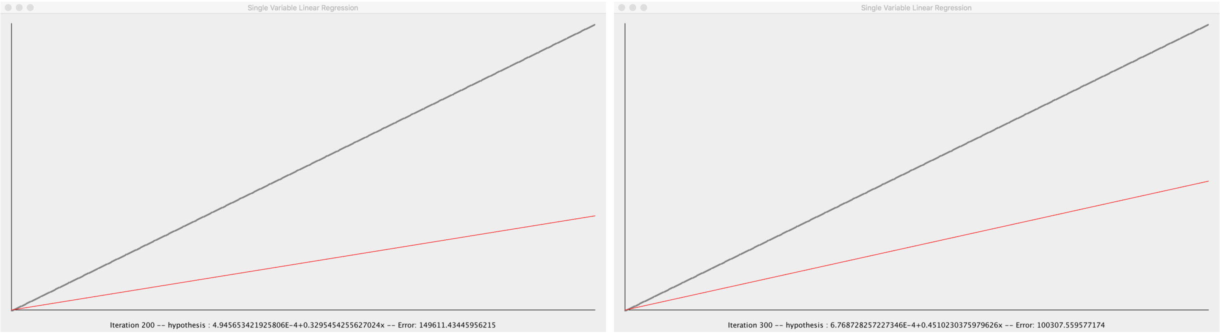
Figures 2, 3 and 4 show what the Display object representing the system looks like after different number of iterations. The line in red is the current hypothesis. As you can see, initially the red line is $y = 0$, our starting point, and the more we iterate, the closer if gets to $y = x$.


Single Variable Linear Regression
Single Variable Linear Regression

```
Iteration 0 -- hypothesis : 0.0+0.0x -- Error: 332833.5
```

```
Iteration 100 -- hypothesis : 2.7191664670348485E-4+0.1811871395131639x -- Error: 223149.49556558786
```

Figure 2 – Initial situation, and after 100 iterations.

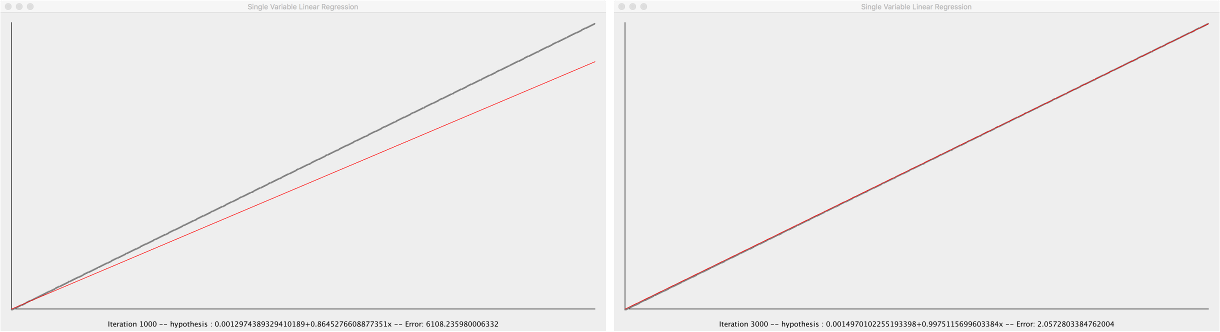

Single Variable Linear Regression
Single Variable Linear Regression

```
Iteration 200 -- hypothesis : 4.945653421925806E-4+0.3295454255627024x -- Error: 149611.43445956215
```

```
Iteration 300 -- hypothesis : 6.768728257227346E-4+0.4510230375979626x -- Error: 100307.559577174
```

Figure 3 – After 200 and 300 iterations.

The classes and JavaDocs are found here :


Single Variable Linear Regression
Single Variable Linear Regression

```
Iteration 1000 -- hypothesis : 0.0012974389329410189+0.8645276608877351x -- Error: 6108.235980006332
```

```
Iteration 3000 -- hypothesis : 0.0014970102255193398+0.9975115699603384x -- Error: 2.0572803384762004
```

Figure 4 – After 1000 and 3000 iterations.

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java
- Display.java
- StudentInfo.java

You have to fill out all the missing methods to obtain the described behaviour.

Question 1.2

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 will create a java.util.Random object and call the nextDouble() method, which returns a random double between 0 and 1. You will then need to figure out how to scale this number to sample from the intervals specified below.

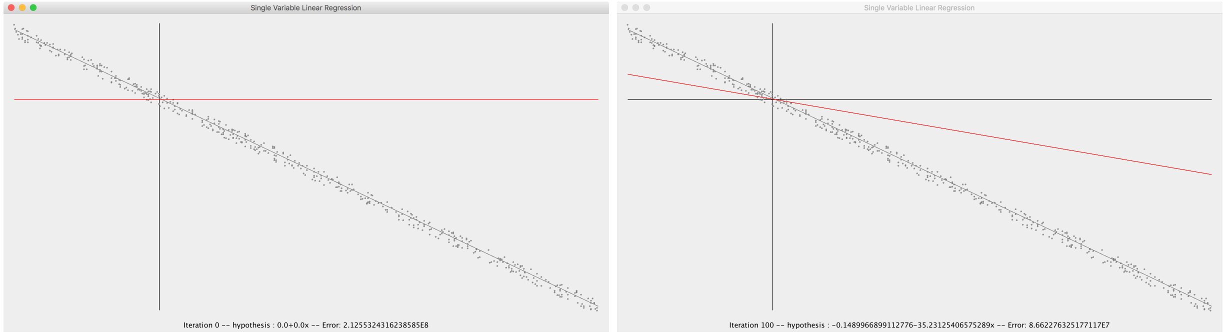
The goal is to provide the implementation of the method **randomLine** for the class **Assignment**. This method should do the following.

- Create a LinearRegression object with 500 points and a corresponding display object.
- Generate a random line of the form $y = ax + b$, where a is randomly sampled from the interval $[-100, 100]$ and b is randomly sampled from the interval $[-250, 250]$, and call **graph.setTarget(a,b)** to draw it on the display - this is the line we are aiming for.
- Generate 500 random points that satisfy the following conditions. The x value must be sampled randomly from $[-100, 300]$ and the corresponding y value is a random value no more than 1000 away from the actual point on the line. That is, for any given x, y should be sampled randomly from $[ax + b - 1000, ax + b + 1000]$. The number 1000 here is known as the *noise*.
- 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 first few iterations of a sample run might look like this.

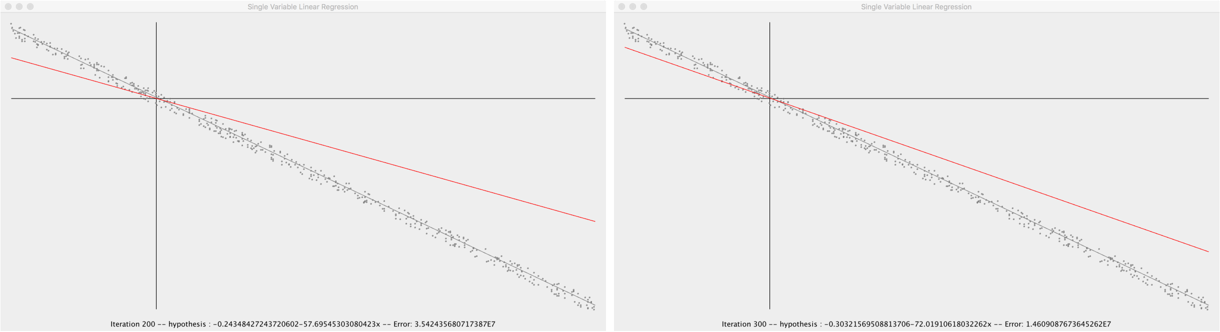
```
> java Assignment
randomLine
Current hypothesis: 0.0+0.0x
Current cost: 1405392.4099890895
Aiming for: 123.72084222501928+6.347541365496268x_1
Press return to continue....

Current hypothesis: 0.044132282664095177+6.9297857925854x
Current cost: 338394.84501478786
Aiming for: 123.72084222501928+6.347541365496268x_1
Press return to continue....

Current hypothesis: 0.058759137512926656+7.020202855455091x
Current cost: 338210.92943203804
Aiming for: 123.72084222501928+6.347541365496268x_1

Press return to continue....
```

Figures 5, 6 and 7 show the system after different number of iterations on a sample run. Note that the class **Display** provides a method **setTarget** which can be used to display on the screen the line that was used to generate the data (the black line on the figures)


Single Variable Linear Regression
Single Variable Linear Regression

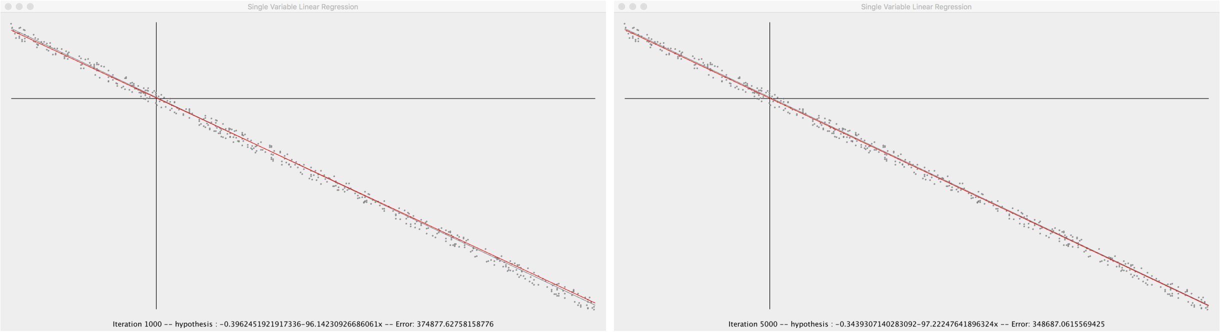
```
Iteration 0 -- hypothesis : 0.0+0.0x -- Error: 2.1255324316238585E8
Iteration 100 -- hypothesis : -0.1489966899112776-35.23125406575289x -- Error: 8.662276325177117E7
```

Figure 5 – Initial situation, and after 100 iterations.


Single Variable Linear Regression
Single Variable Linear Regression

```
Iteration 200 -- hypothesis : -0.24348427243720602-57.69545303080423x -- Error: 3.542435680717387E7
Iteration 300 -- hypothesis : -0.30321569508813706-72.01910618032262x -- Error: 1.4609087673645262E7
```

Figure 6 – After 200 and 300 iterations.


Single Variable Linear Regression
Single Variable Linear Regression

```
Iteration 1000 -- hypothesis : -0.3962451921917336-96.14230926686061x -- Error: 374877.62758158776
Iteration 5000 -- hypothesis : -0.3439307140283092-97.22247641896324x -- Error: 348687.0615569425
```

Figure 7 – After 1000 and 5000 iterations.

The classes and JavaDocs are found here :

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java
- Display.java
- StudentInfo.java

You have to provide the implementation of method **randomLine** for the class **Assignment**.

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 n features $x_1, x_2, \ldots, x_n$, the new hypothesis function is :

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$

Notations :

$$\begin{aligned}
x_j^{(i)} &= \text{value of feature } j \text{ in the } i^{th} \text{ training example} \\
x^{(i)} &= \text{the input (features) of the } i^{th} \text{ training example} \\
m &= \text{the number of training examples} \\
n &= \text{the number of features}
\end{aligned}$$

For convenience, we are adding a $n+1$ "feature" $x_0 = 1$ (that is, $\forall i \in [1, \ldots, m], x_0^{(i)} = 1$). The hypothesis function $h_\theta(x)$ can now be rewritten

$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$

Gradient Descent for Multiple Variables

The new cost function is

$J(\theta_0, \theta_, \ldots, \theta_n) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$

The gradient descent algorithm becomes :

$$\begin{aligned}
&\text{repeat until convergence : } \{ \\
&\qquad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, \ldots, \theta_n) \\
&\qquad \text{for } j \in [0, \ldots, n] \\
&\}
\end{aligned}$$

All the θ_j have to be updated simultaneously. This can be written as

$$\begin{aligned}
&\text{repeat until convergence : } \{ \\
&\qquad \theta_0 := \theta_0 - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\
&\qquad \theta_1 := \theta_1 - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)} \\
&\qquad \theta_2 := \theta_2 - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)} \\
&\qquad \ldots \\
&\}
\end{aligned}$$

that is,

$$
\begin{aligned}
&\text{repeat until convergence, updating } \theta_j\text{'s simultaneously : } \{\\
&\qquad \theta_j := \theta_j - \alpha \frac{2}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}\\
&\qquad \text{for } j \in [0, \ldots, n]\\
&\}
\end{aligned}
$$

Implementation

We will generalize our implementation from Question 1 to perform multivariate linear regression. However, we will no longer use Display objects, for dimensionality reasons. You will have to modify your **LinearRegression** class to support inputs that are vectors of doubles (represented by arrays), 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 in the **Assignment** classes.

Question 2.1

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 with 2 features and 2000 sample points.
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.

The classes and JavaDocs are found here :

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java
- StudentInfo.java

Question 2.2

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

- Create a LinearRegression object with 2 features and 5000 points.
- Sample random coefficients $a, b, c \in [-100, 100]$. The plane we are aiming for is $x_3 = c + ax_1 + bx_2$.
- Now add 5000 points that satisfy the following conditions. The input (x_1, x_2) should be sampled with $x_i \in [50, 4000]$. We set the noise this time to 20, so that for any given (x_1, x_2), we should have $z = ax_1 + bx_2 + c + \delta$, where δ is randomly sampled from $[-20, 20]$.
- As before, pick sensible values for α and numbers of steps, and print the current hypothesis, current cost, and plane being aimed for at each iteration.

The classes and JavaDocs are found here :

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java
- StudentInfo.java

Question 2.3

We now generalize the previous part to allow a random equation for any given dimension. The randomDimension method should take an integer n as an argument, specifying the required dimension n of the input vector, and the main program will test this with $n = 50$. Implement the method to do the following.

- Create a LinearRegression object with n features and 5000 points. A general input vector will be of the form $(x_1, x_2, \ldots, x_n)$.
- 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

$$r = t_0 + \sum_{i=1}^{n} t_i x_i = t_0 + t_1 x_1 + t_2 x_2 + \ldots + t_n x_n. \tag{1}$$

- Add 5000 points that satisfy the following conditions. For each input vector $(x_1, \ldots, x_n)$, sample x_i randomly from $[50, 4000]$. Then, let the result r be as in Equation 1, plus or minus a noise of 20. The value that is added is then $((x_1, \ldots, x_n), r)$.
- Again, choose sensible values for α and numbers of steps, and print the current hypothesis, current cost, and equation being aimed for at each iteration.

The classes and JavaDocs are found here :

- JavaDoc Documentation
- Assignment.java
- LinearRegression.java
- StudentInfo.java

Academic fraud

This part of the assignment is meant to raise awareness concerning plagiarism and academic fraud. Please read the following documents.

- http://www.uottawa.ca/administration-and-governance/academic-regulation-14-other-important-information
- http://web5.uottawa.ca/mcs-smc/academicintegrity/home.php

Cases of plagiarism will be dealt with according to the university regulations.

By submitting this assignment, you acknowledge :

1. **You submission is your own work**
2. **having read the academic regulations regarding academic fraud, and**
3. **understanding the consequences of plagiarism**

WARNINGS

- Failing to strictly follow the submission instructions will cause automated test tools to fail on your submission. Consequently, your submission will **not** get marked.
- A tool will be used to detect similarities between submissions. We will run that tool on all submissions, across all the sections. Submissions that are flagged by the tool will receive a mark of 0.
- It is your responsibility to ensure that your submission is indeed received by the back-end software, BrigthSpace. If you submission is not there by the deadline, it will obviously **not** get marked.
- Late submission will not be accepted.

Rules and regulations

Follow all the directives available on the assignment directives web page, and submit your assignment through the online submission system Brightspace.

Files

You must preferably do the assignment in teams of two, but you can also do the assignment individually.

You must hand in a **zip** file. No other format will be accepted. The main directory must be called **a1_300000_300001**, where

- **300000** and **300001** are the student number of the two students submitting this file
- **a1** is the lowercase letter **a** followed by the assignment number (**1** in this case
- the separators are **underscores**, not dashes
- there are no spaces in the directory name
- if you are working along, list your student number twice.

The zip file you submit be contain the following files (and only these files) :

- A text file README.txt which contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- For each subquestion, in separate directories called **Q1-1**, **Q1-2**, **Q2-1**, etc... :
  - The source code of **all** your classes. Each directory must be self contained and have all the files.
  - The corresponding JavaDoc doc directory.
  - StudentInfo.java, properly completed and properly called from your main.

Last Modified : 27 janvier 2018