1 Big Picture of Machine Learning
2 Computational Optimization - General Comments
3 The Gradient Descent Search Algorithm
4 Gradient Descent for Linear Regression
5 Nonlinear Regression via Gradient Descent
6 Stochastic Gradient Descent
7 In Class Challenge Task
8 Homework

Code

Optimization for Machine Learning

The Gradient Descent Method

Dr. Eric Sullivan – DS401

1 Big Picture of Machine Learning

Almost all machine learning algorithms share the same following features:

- We have data which we want to use to build a model with prediction as a goal
- We define a **cost function** (or **objective function**) that measures the errors made with the model.
- Minimize the cost function.

> In simple terms: **almost every machine learning algorithm is a minimization process** where we tweak the parameters in our model so as to minimize some measure of cost.

A *cost function* might be to count the number of classification errors that we make, it might be the sum of the squares of the residual differences, or it might be a function that simultaneously measures the residual error in a model along with the complexity of the model.

For example, in simple linear regression with one predictor and one response we seek to minimize the **mean squared errors**

$$MSE = \frac{1}{n}\sum_{k=1}^{n}\left(y_i - \hat{y}_i\right)^2$$

where $\hat{y}_i$ is the predicted response for x_i and

$$\hat{y}_i = \beta_0 + \beta_1 x_i.$$

Therefore, the goal of single variable regression can be stated as

$$\text{minimize } MSE(\beta_0, \beta_1)$$

$$\text{by allowing } \beta_0 \text{ and } \beta_1 \text{ to vary}$$

You may recognize this as a classic calculus problem wherein we could just take the derivatives $\frac{\partial MSE}{\partial \beta_0}$ and $\frac{\partial MSE}{\partial \beta_1}$, set them to zero, and solve the resulting simultaneous system of equations to get the familiar equations

$$\beta_0 = \bar{y} - \beta_1 \bar{x} \quad \text{and} \quad \beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

after some significant algebra.

While in the single variable case the algebra is tractable and possible, in the multi-variable case this quickly becomes infeasible to do by hand. Moreover, in more complicated machine learning algorithms it might be very challenging to even state the objective function in a succinct algebraic form. Of course, in statistical model building there are often many parameters to find (think about multiple regression, for example) and we are almost never in the single variable case.

Often times in Machine Learning we focus on solving this type of problem approximately so as to avoid the messy (or potentially impossible) algebra and calculus.

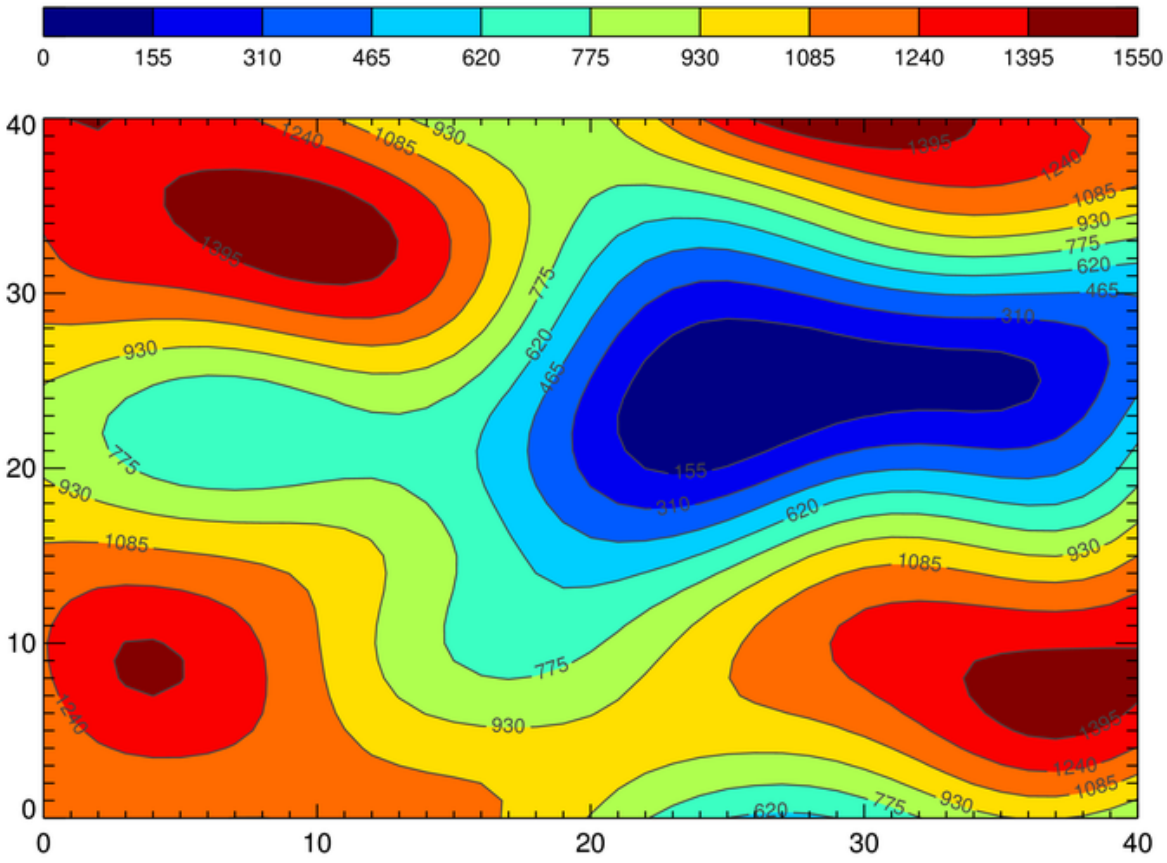
2 Computational Optimization - General Comments

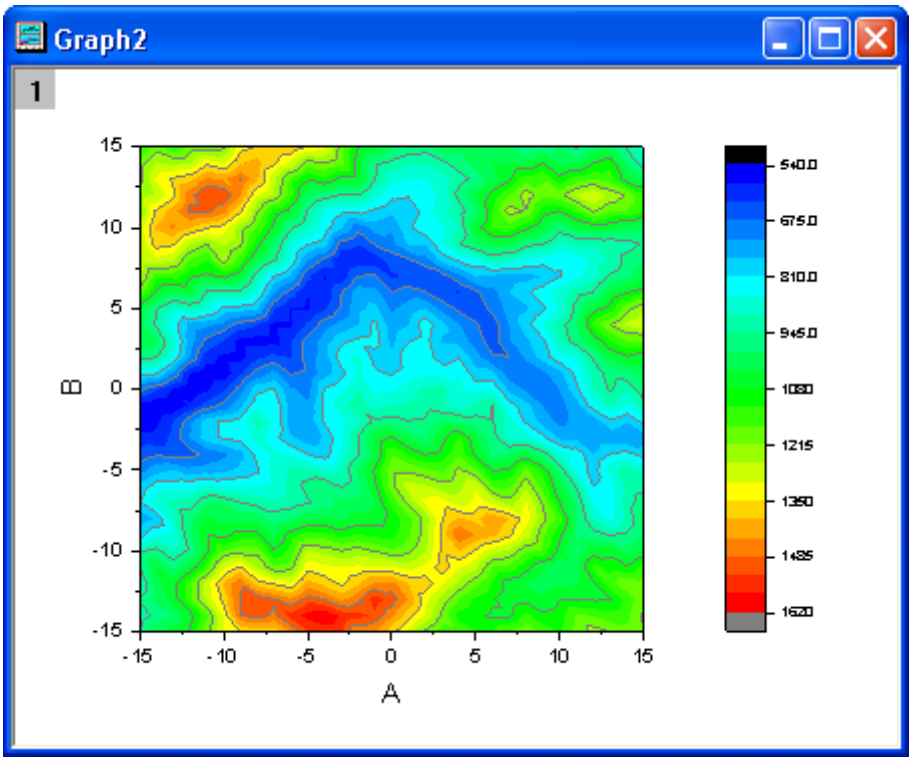
For the sake of simplicity here we will remove the statistical context for a brief moment and focus on a purely mathematical context.

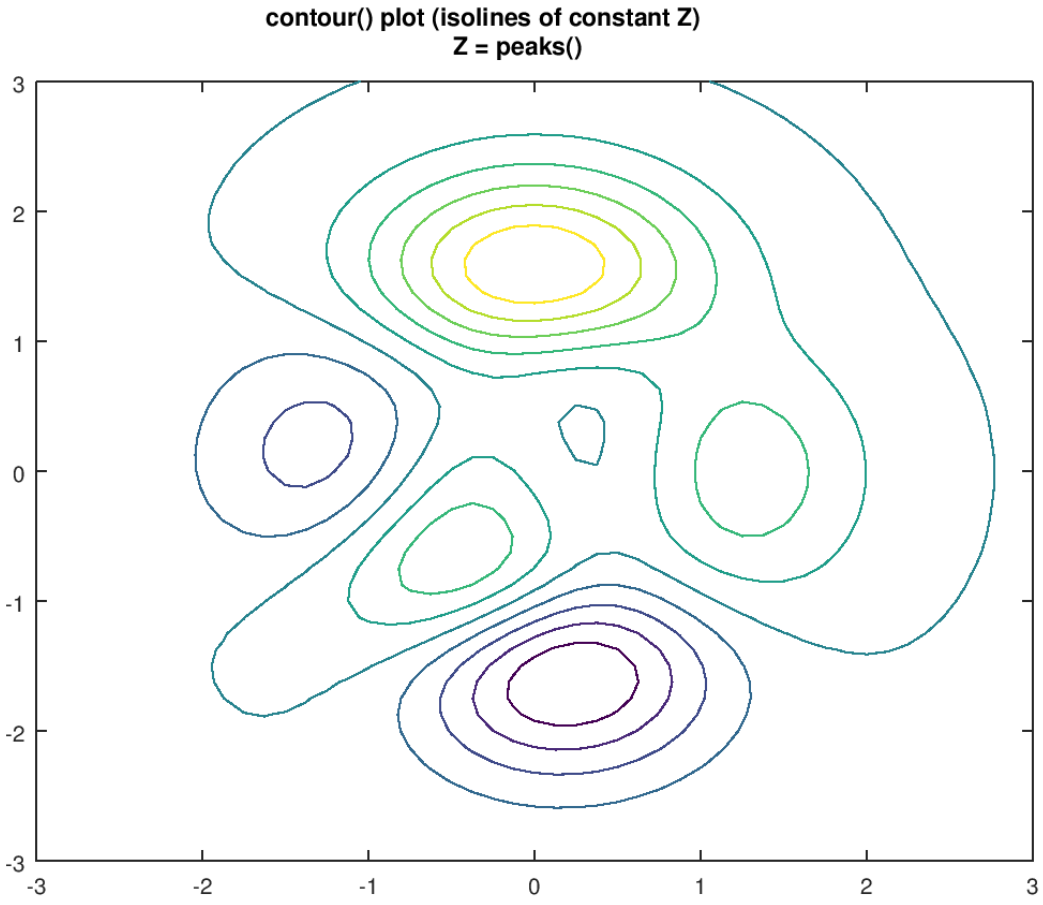
If you're trying to minimize a multivariable function there is one technique that works well above all else:

> Stand on the function, face downhill, take a small step, adjust course, and repeat until you reach a minimum.

The trouble with this technique is finding what "downhill" actually means. That's where the gradient descent method comes in. It is helpful to think of the idea graphically. On the contour plots below you can imagine standing at a point, facing downhill, and taking small steps in this way until you find the minimum.







3 The Gradient Descent Search Algorithm

Definition: The **gradient** of a two variable function is the vector

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right).$$

The **gradient** of a many-variable function is the vector

$$\nabla f = \left(f_{x_1}, f_{x_2}, f_{x_3}, \ldots, f_{x_p} \right)$$

where subscripts are used in place of the standard derivative notation.

Theorem The gradient always points in the direction of maximum increase.

Proof (requires some multivariable calculus) Let f be a differentiable function of p variables and let $\boldsymbol{u}$ be a unit vector in $\mathbb{R}^p$. If we take the dot product of $\boldsymbol{u}$ and ∇f we get

$$\nabla f \cdot \boldsymbol{u} = \|\nabla f\| \|\boldsymbol{u}\| \cos\theta$$

where θ is the angle between the two vectors. Alternatively, the directional derivative is defined as the dot product of a function's gradient and a given unit direction. Therefore the directional derivative of f in the direction of $\boldsymbol{u}$ is

$$D_{\boldsymbol{u}} f = \|\nabla f\| \|\boldsymbol{u}\| \cos\theta.$$

For this theorem we don't actually care about the magnitude of the directional derivative – just where it point. Hence we can always turn ∇f into a unit vector by dividing by the length and get

$$D_{\boldsymbol{u}} f = \cos\theta.$$

The cosine function is maximized when $\theta = 0 \pm 2\pi k$ for any integer k, which means that the direction of maximal change is the same direction as the gradient. QED

The key takeaway here is:

> The gradient of a multivariable function always points in the direction of maximum increase

It is the previously stated theorem that we'll be using over and over again., if we can compute the gradient of our objective function we know which direction is *uphill* at any given point. If we flip the direction we know where *downhill* is. This gives us the gradient descent method

3.1 The Gradient Descent Algorithm

- Pick a feasible starting point
- Calculate the gradient at your point

- Follow the negative gradient downhill for a short distance
  - make the gradient into a unit vector
  - subtract your current point minus some scalar times the unit gradient vector
- Repeat the process from your new point

Mathematically, gradient descent is the iterative process

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \alpha \frac{\nabla f(\boldsymbol{x}_n)}{\|\nabla f(\boldsymbol{x}_n)\|}$$

where $\boldsymbol{x}_n$ is the vector $(x_{1n}, x_{2n}, \ldots, x_{pn})$ and α is called the **learning rate**. The value of the learning rate controlls how large the steps downhill are.

3.2 Simple 2D Example:

If we consider the quadratic function $f(x, y) = (x-2)^2 + (y-3)^2$ then we know fully well that the minimum occurs at the point $(x, y) = (2, 3)$ (this way we can check our answers when we get them)

The gradient vector of $f(x, y)$ is

$$\nabla f = (2(x-2),\ 2(y-3))$$

If we start at the point $(4, 7)$ and take steps of size 0.5 then the algorithm proceeds as follows:

Step Number	(x, y)	**Unit Gradient Vector**
0	$(4, 7)$	$\frac{(4,8)}{\|(4,8)\|} \approx (0.447, 0.894)$
1	$(4, 7) - \frac{1}{2}(0.447, 0.894) = (3.833, 6.667)$	
2	$(3.667, 6.333)$	
3	$(3.5, 6.0)$	
⋮	⋮	
12	$(2, 3)$	

3.3 A Note About Learning Rates:

In the simplest case we can leave the learning rate fixed, but this causes some problems if the step is too large (e.g. you could repeatedly overshoot your target minimum). Therefore we typically allow α to depend on some condition. A few options are:

- The learning rate could depend on the step number so we would choose a function for α that monotonically decreases as the iteration proceeds.

- The step size could depend on the magnitude of the gradient – the further we are away from a gradient of zero the large the step size should be.

All of these are called **adaptive learning rates**, where the learning rate changes as you proceed through the gradient descent process.

3.4 Coding Gradient Descent

Write code that takes a gradient function (defined by you), a starting point, and a learning rate (or learning rate function) and returns the optimal decision values as well as a plot that shows how your cost function behaves over successive iterations.

Try your code on the function given in the example above to verify that you converge to $(2, 3)$. Then try your code on a new function where you don't necessarily know your minimum. Experiment with different rules for adaptive learning rates.

As always, some code is given below for a solution, but only use it if you are completely stuck and need to *cheat* for some reason. ... Always try to code these things on your own first!!

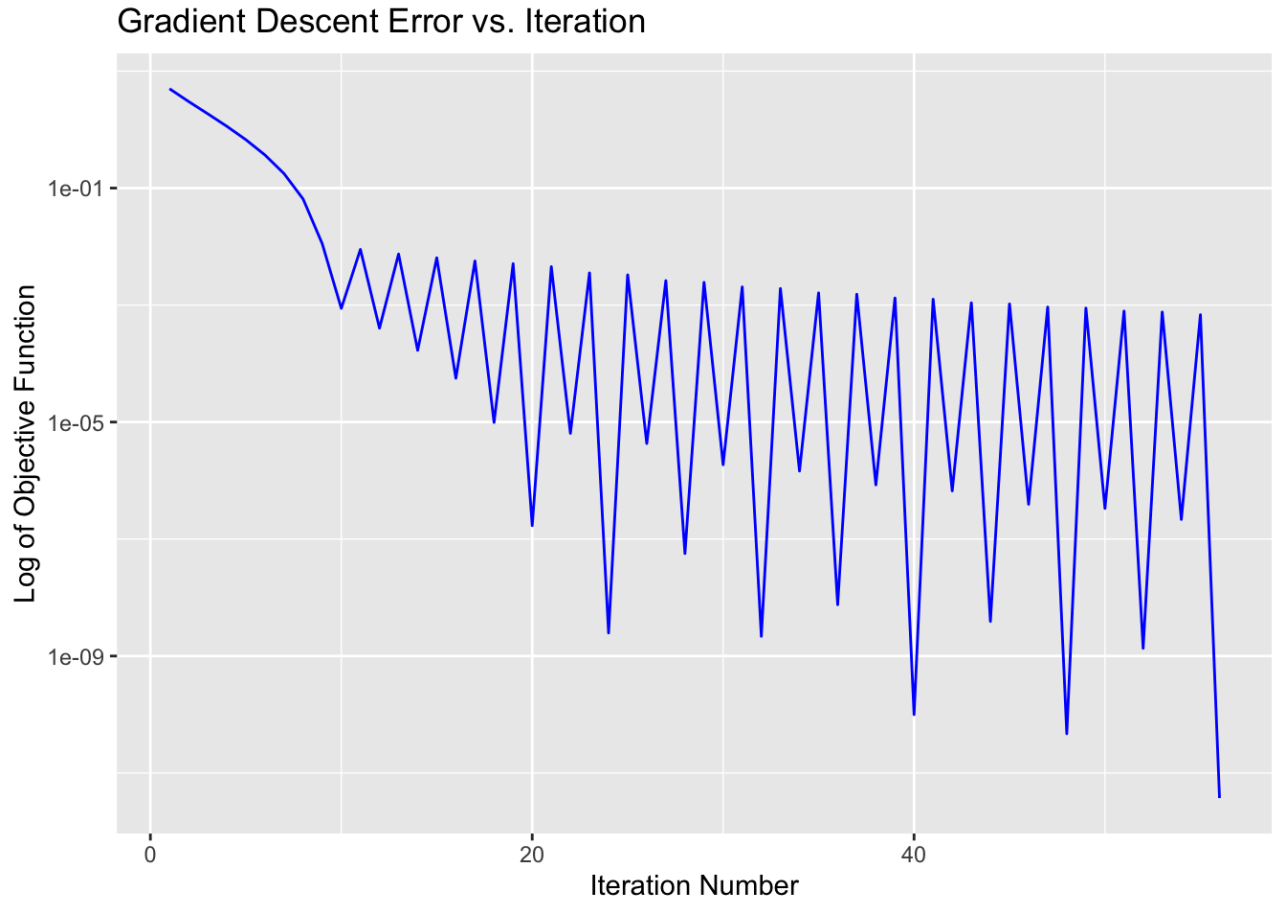
Hide

Code

```
## [1] 2.000001 3.000002
```

Hide

Code



Note that the erratic looking behavior of the log of the cost function is due to the fact that the gradient descent algorithm will jump past the optimal point, jump back, and then continue to jump around until the learning rate causes it to settle down. Think of it this way: If we are within 0.1 units of the optimal point and the learning rate is 0.5 then the gradient descent method will overshoot (onto the uphill side) by 0.4 units on the next step. After that it will have to turn around and march back toward the optimal point. When it gets close again it will likely overshoot by a bit and then have to travel back again. This repeats itself forever unless you have an adaptive learning rate that slows down the step size as the iterations progress.

4 Gradient Descent for Linear Regression

In linear regression we are aiming at using our data to estimate the parameters $\beta_0, \beta_1, \ldots, \beta_p$ so that the sum of the squared errors is minimized. We can do this very efficiently with calculus or with linear algebra (as we'll see soon), but for the sake of example let's do this with gradient descent.

4.1 Simple Linear Regression

Recall that one particular objective function for single-variable regression is the **mean squared error** (MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)^2$$

where (x_i, y_i) is an observation from a dataset, β_0 is the intercept of the model function, β_1 is the slope of the model function, and n is the number of observations in the dataset. After a little careful calculus, the gradient of the MSE function with respect to the parameters is

$$\begin{aligned}\nabla MSE &= \left(\frac{\partial MSE}{\partial \beta_0}, \frac{\partial MSE}{\partial \beta_1}\right) \\ &= \left(\frac{-2}{n}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i), \frac{-2}{n}\sum_{i=1}^{n}[(y_i - \beta_0 - \beta_1 x_i)x_i]\right) \\ &= \frac{-2}{n}\left(\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i), \sum_{i=1}^{n}[(y_i - \beta_0 - \beta_1 x_i)x_i]\right).\end{aligned}$$

Even though this particular gradient vector is a bit complicated the process of doing the gradient descent search is still the same. You are standing at a point in the $\beta_0 - \beta_1$ plane, the gradient vectors tells you where *uphill* is on the MSE surface, so you travel in the opposite direction until you're at a minimum. One thing to keep in mind is that the MSE function is quadratic in β_0 and β_1 so we know that there is one unique global minimum. It is just a matter of finding it.

Let's put this into action.

4.2 Concrete Compressive Strength - Single Variable Regression Example

Write an `R` script that takes in a dataframe with two columns (x then y) and outputs the estimates for β_0 and β_1. Use the `Concrete_Data.csv` dataset to build a simple linear regression model predicting `Concrete_compressive_strength ~ Cement`. Check your answer against what `R`'s `lm` command tells you. Also create a plot that shows how the MSE behaves over successive iterations. Remember that we don't know the optimum value of the MSE ... we're just looking for it to settle down as the iterations proceed.

Only look at the following code if you are completely stuck!

Hide

Code

```
## (Intercept)      Cement
## 13.44252811  0.07958034
```

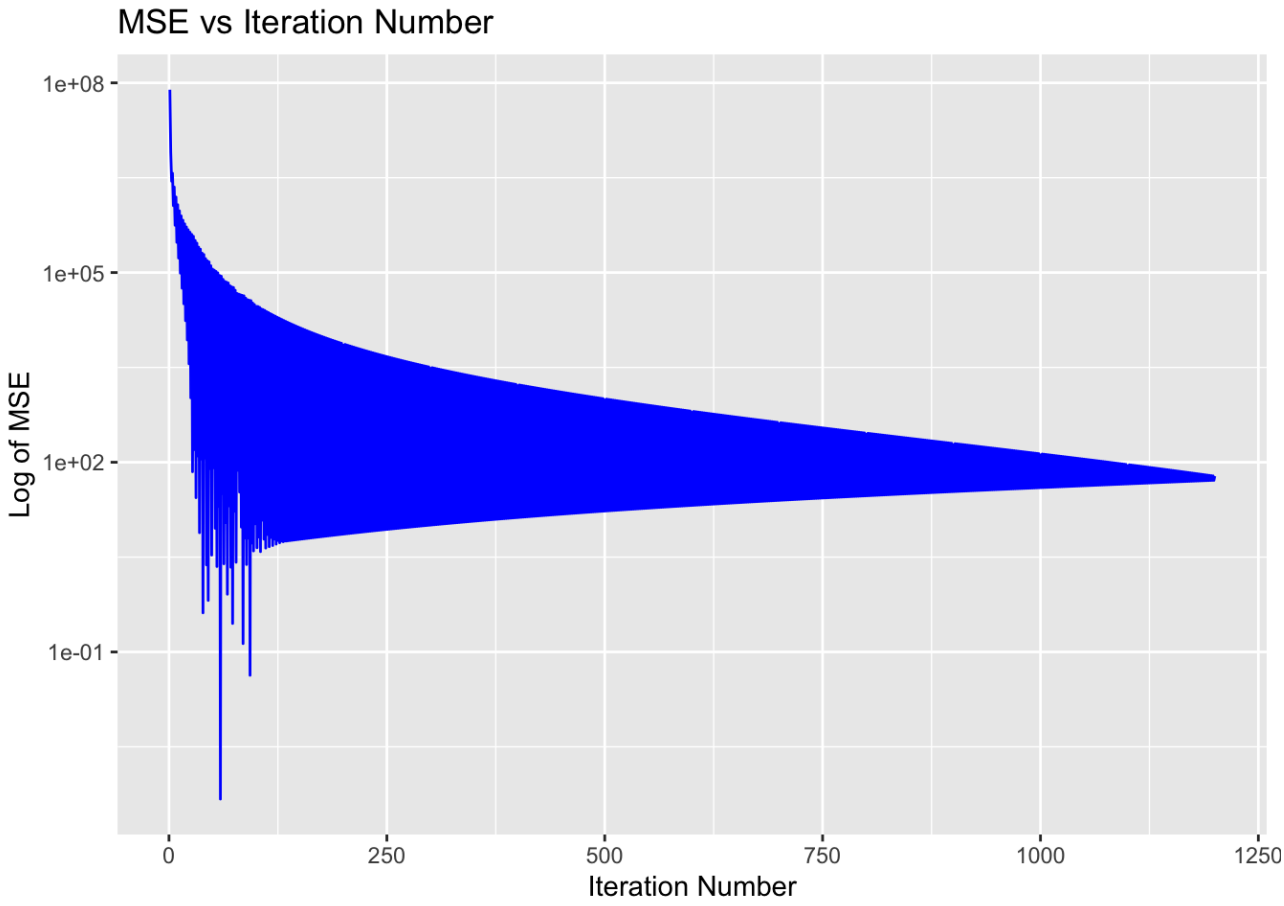
Hide

Code

```
## [1] 13.01919358  0.08194732
```

Hide

Code



4.3 Multiple Regression

Nothing changes when we have more than two parameters. If a function f is a function of p variables then $f = f(x_1, x_2, \ldots, x_p)$ and the gradient vector is

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{x_p} \right).$$

In the case of multiple regression the cost function is still the residual sum of squared errors

$$MSE = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi} \right)^2$$

where x_{ji} is the value of the i^{th} observation in the j^{th} predictor. From here the individual derivatives are

$$\begin{aligned}
\frac{\partial MSE}{\partial \beta_0} &= \frac{-2}{n}\sum_{i=1}^{n}\left(y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi}\right) \\
\frac{\partial MSE}{\partial \beta_1} &= \frac{-2}{n}\sum_{i=1}^{n}\left[\left(y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi}\right) x_{1i}\right] \\
&\vdots \qquad\qquad\qquad \vdots \\
\frac{\partial MSE}{\partial \beta_p} &= \frac{-2}{n}\sum_{i=1}^{n}\left[\left(y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi}\right) x_{pi}\right]
\end{aligned}$$

It is worth noting that the β_0 derivative can be seen as exactly the same as the others if we pad the dataset with a column of 1s in the first column. This is particularly helpful for coding the gradient descent algorithm for multiple regression.

Also note that we have greatly increased the dimensionality of the problem. There are $p + 1$ different parameters to find, so we are travelling *downhill* in a $p + 1$ dimension space. For the concrete data that means that we are working in a 9-dimensional space. Thankfully, our intuition from 2-dimensional problems carries over nicely and the same gradient descent algorithm still works.

4.4 Concrete Compressive Strength - Multiple Regression

Write an `R` script that takes in a dataframe with all of the columns (xs first and y last) and outputs the estimates for $\beta_0, \beta_1, ..., \beta_p$ where p is the number of predictors. Use the `Concrete_Data.csv` dataset to build a simple linear regression model predicting `Concrete_compressive_strength ~ .`. Check your answer against what `R`'s `lm` command tells you and make a plot of the values of the MSE as the iterations progress.

Only look at the following code if you are completely stuck!

Hide

Code

```
##        (Intercept)             Cement Blast_Furnace_Slag
##       -23.33121358         0.11980433         0.10386581
##            Fly_Ash              Water   Superplasticizer
##         0.08793432        -0.14991842         0.29222460
##   Coarse_Aggregate     Fine_Aggregate                Age
##         0.01808621         0.02019035         0.11422207
```

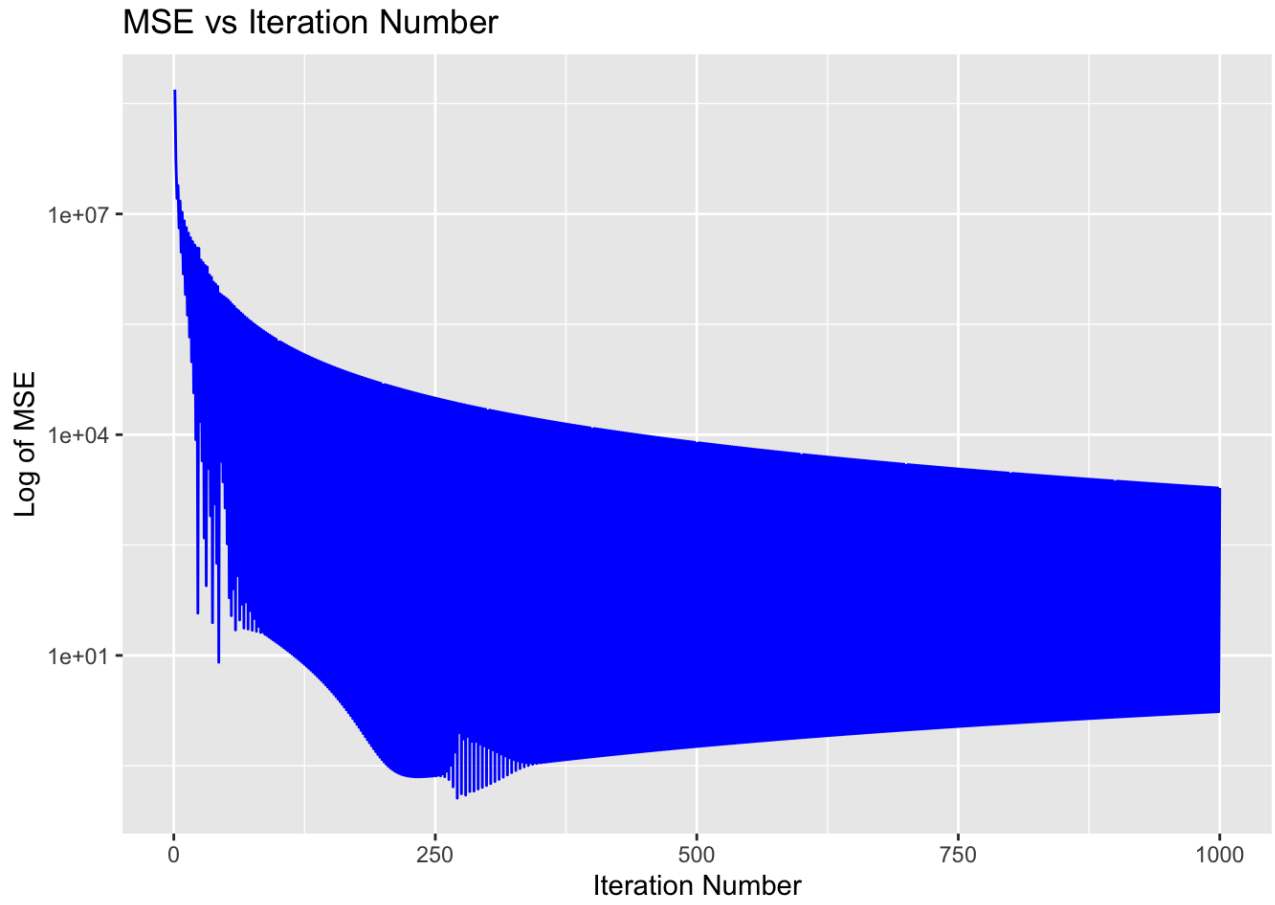
Hide

Code

```
## [1] -23.30006440   0.11934583   0.10353467   0.08743096  -0.14926
939
## [6]   0.29984296   0.01731492   0.01939378   0.11415739
```

Hide

Code



Notice in my solution that I needed to start the gradient descent algorithm pretty darn close to the optimal solution. This is a catch 22 – you need to know an approximation for the optimal coefficients in order for gradient descent to find the optimal coefficients!

5 Nonlinear Regression via Gradient Descent

There are MANY other objective functions that could be used to find the parameters in a regression setting. The *MSE* is the most natural geometrically and it lends itself to several very nice linear algebra based solution techniques (as we'll see in the future). However, we

can look at so called *penalized regression* where we instead seek to minimize a modified *MSE* function that makes it more challenging to minimize in some sense.

One example is the the **Ridge Regression** objective function

$$C(\beta_0, \beta_1, \ldots, \beta_p) = \frac{1}{n}\sum_{i=1}^{n}\left[\left(y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i} - \cdots \beta_p x_{pi}\right)^2\right] + \lambda\left(\beta_1^2 + \beta_2^2 + \cdots + \beta_p^2\right).$$

In this case we introduce a new parameter, λ, that controlls how large the sum of the squares of the coefficients can get. We call this a *penalized* method since if the coefficient values are large then it will move us away from the minimum of the objective function. Another way to say this is that we have incentivized the minimization process to keep the parameters small (notice that this doesn't apply to β_0 since we want to allow for a nontrivial shift in intercept). We are using the function name "C" as an alliteration for the word "cost", since this is just a new cost function.

The big advantage to this style of regression is that the optimization routine will shrink less significant parameters toward zero – hence we can use this as a variable selection technique: the parameters that end up very small relative to others correspond to predictors that can be removed from the modeling process.

To choose the λ parameter (also called the *shrinkage parameter* or the *penalty parameter*) we would have to test our various values of λ on different testing data sets. For now, however, let's just fix λ as a large number and see how gradient descent works on these problems.

The individual derivatives now are very similar to those for the *MSE* function in multiple regression, but now each derivative (other than β_0) has a new term based on the penalty in the objective function.

$$\begin{aligned}
\frac{\partial C}{\partial \beta_0} &= \frac{-2}{n}\sum_{i=1}^{n}\left(y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi}\right) \\
\frac{\partial C}{\partial \beta_1} &= \frac{-2}{n}\sum_{i=1}^{n}\left[\left(y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi}\right) x_{1i}\right] + 2\lambda\beta_1 \\
&\vdots \qquad\qquad\qquad \vdots \\
\frac{\partial C}{\partial \beta_p} &= \frac{-2}{n}\sum_{i=1}^{n}\left[\left(y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i} - \cdots - \beta_p x_{pi}\right) x_{pi}\right] + 2\lambda\beta_p
\end{aligned}$$

Remember that this type of regression is meant to be a variable selection technique. In the end of the gradient descent algorithm, once we find approximations for the coefficients, we look at the predictors that were decreased (proportionally) the most from the `lm` or non-penalized gradient descent method and elementate them as predictors.

5.1 Ridge Regression on the Concrete Data

Impelement Ridge Regression via gradient descent on the `Concrete_Data.csv` dataset. We want to find which variables might be good candidates for being eliminated based on their resulting relative size after the ridge regression optimization routine has finished. Compare this with typical variable selection techniques based on p-values (you will get different answers, but not substantially different). Also show a plot of the cost function as a function of the iteration number.

Only look at the following code if you are completely stuck!

The value that we'll use for λ (the shrinkage parameter) is arbitrarily chosen for these solutions as

Hide

Code

```
## [1] 1000
```

Hide

Code

```
##        (Intercept)             Cement Blast_Furnace_Slag
##       -23.33121358         0.11980433         0.10386581
##            Fly_Ash              Water   Superplasticizer
##         0.08793432        -0.14991842         0.29222460
##   Coarse_Aggregate     Fine_Aggregate                Age
##         0.01808621         0.02019035         0.11422207
```

Hide

Code

```
## [1] -23.29993378   0.10268048   0.07692534   0.06059690  -0.05425
276
## [6]   0.07327634   0.01225129   0.02039473   0.07871721
```

The following are relative percent changes compared to `R`'s `lm` solution:

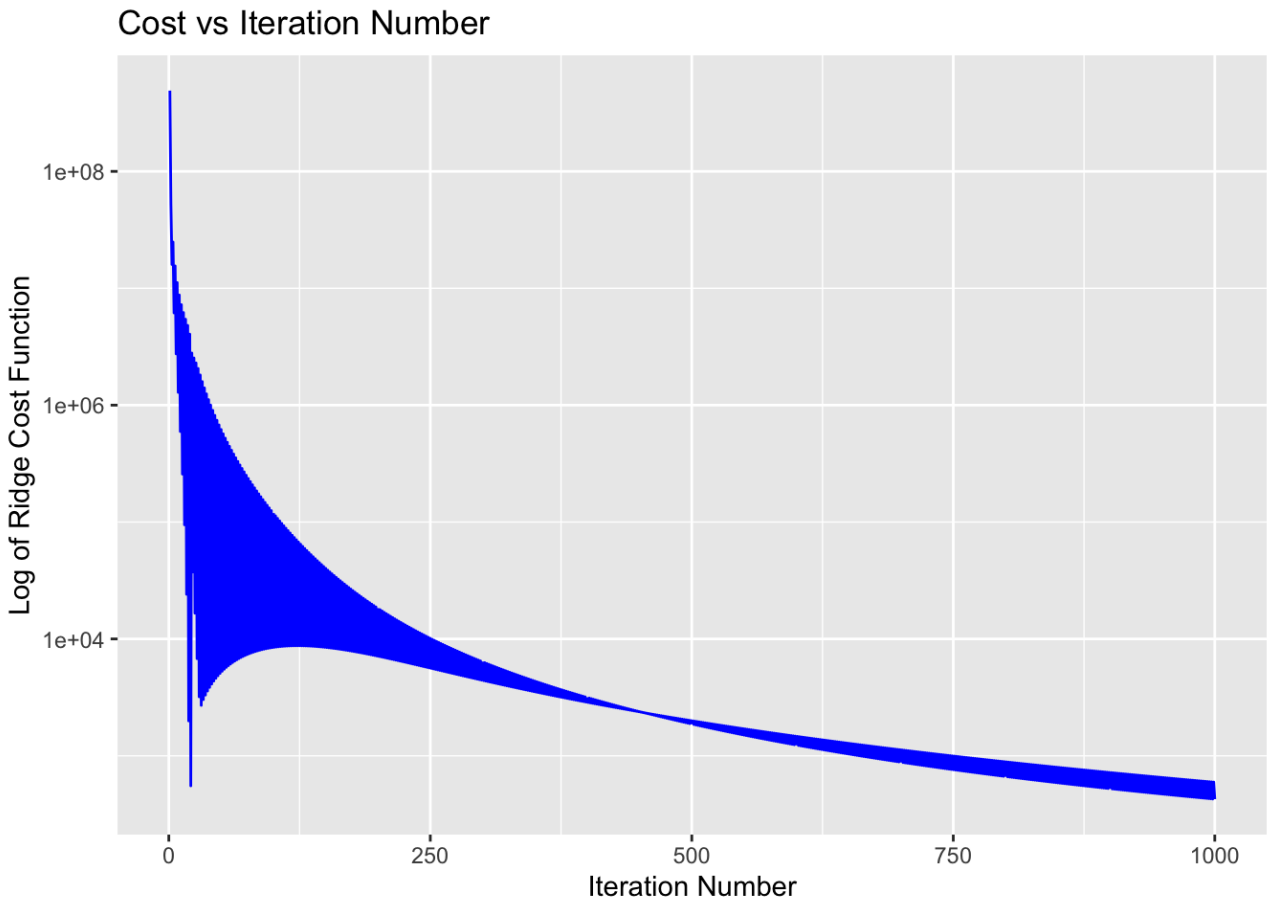
Hide

Code

```
##         (Intercept)              Cement Blast_Furnace_Slag
##          -0.1340685        -14.2931837        -25.9377628
##             Fly_Ash               Water   Superplasticizer
##         -31.0884587        -63.8118115        -74.9246503
##    Coarse_Aggregate     Fine_Aggregate                Age
##         -32.2617232          1.0122552        -31.0840613
```

Hide

Code



In my solution (with my value of λ) I find that `Superplasticiser` should be the first predictor on the chopping block.

6 Stochastic Gradient Descent

There is one problem with the gradient descent algorithm in Machine Learning: *The gradient needs to see all of the data every time it is computed.*

If the dataset is large, or if the gradient vector is particularly difficult to compute, then we'll find that the gradient descent algorithm can be really slow. One very modern fix for this problem is **not** to use all of the data at each step of the gradient descent algorithm, but

instead just to use a random subset of the observations. This should speed things up computationally! An additional advantage is that we drastically reduce the likelihood that the gradient descent algorithm sees high outliers or large leverage points.

The algorithm looks like this:

- Pick a feasible starting point
- Randomly select a sample of the observations (without replacement)
    - use the sample to estimate the gradient at your point
    - update the decision variables using the gradient descent algorithm
- Repeat

It should be relatively obvious now that we will expect pretty erratic behaviour from the algorithm, but if we use an adaptive learning rate the behaviour should settle down as the number of iterations proceeds.

6.1 Stochastic Gradient Descent - Single Variable Concrete Data

Let's go back to a single variable regression of `Concrete_compressive_strength ~ Cement`, but this time we'll implement it with stochastic gradient descent.

Only look at the following code if you are completely stuck!

Hide

Code

```
## (Intercept)      Cement
## 13.44252811  0.07958034
```

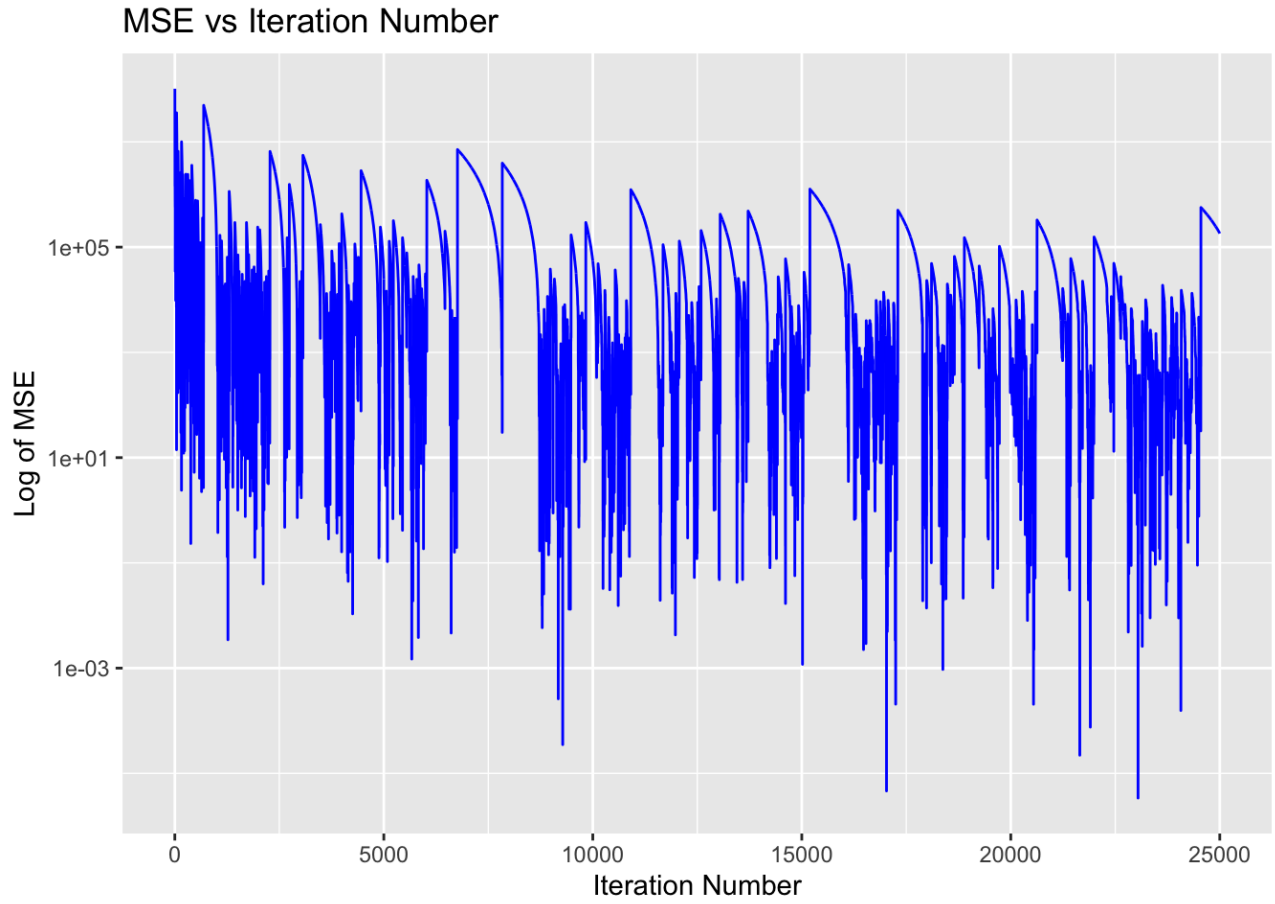
Hide

Code

```
## [1] 13.00748617  0.03390512
```

Hide

Code



6.2 Stochastic Gradient Descent - Multiple Regression on Concrete Data

Now implement stochastic gradient descent on the `Concrete_Data.csv` dataset to predict `Concrete_compressive_strength` from all of the other variables.

Only look at the following code if you are completely stuck!

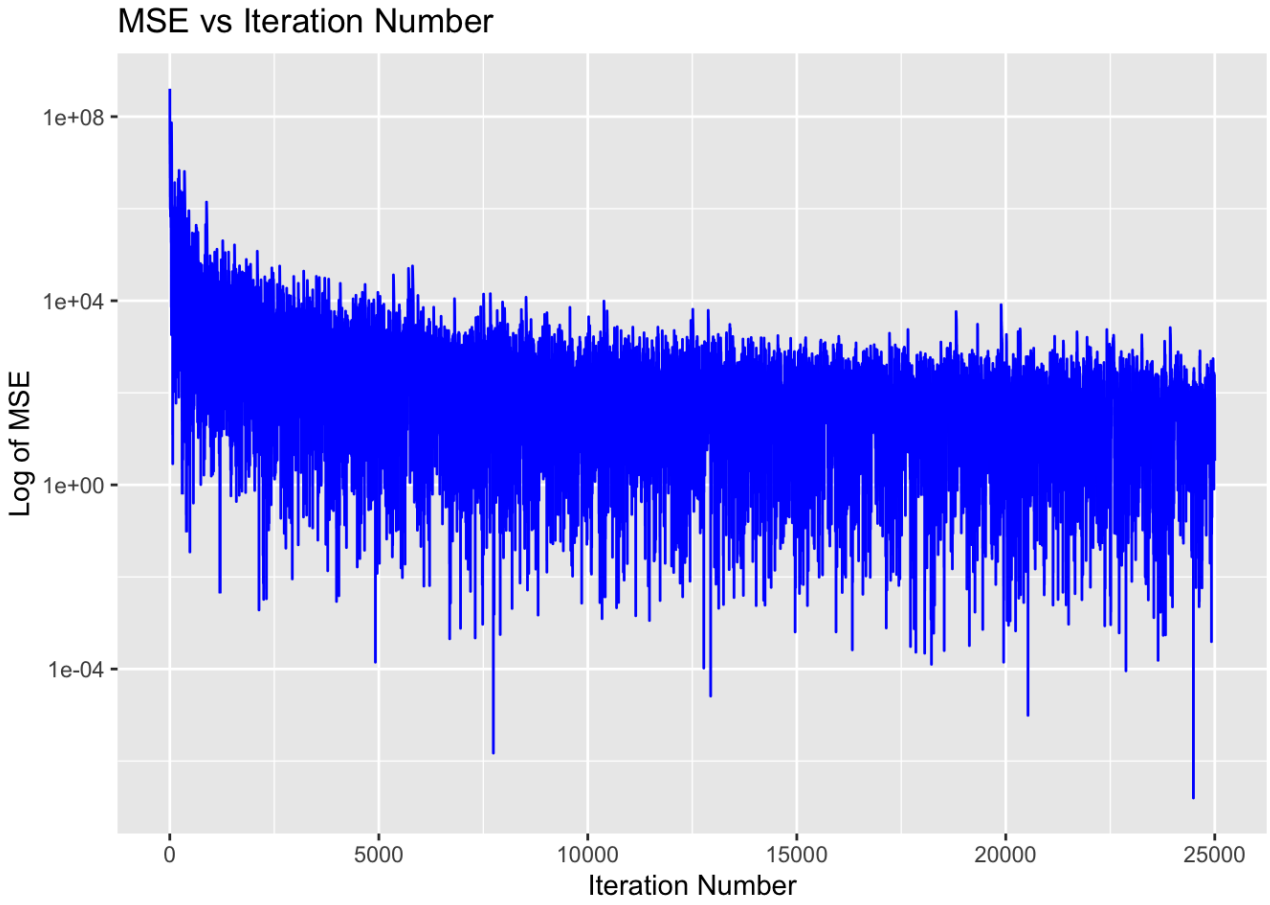
Hide
Code

```
##         (Intercept)              Cement Blast_Furnace_Slag
##       -23.33121358         0.11980433         0.10386581
##             Fly_Ash               Water   Superplasticizer
##          0.08793432        -0.14991842         0.29222460
##    Coarse_Aggregate     Fine_Aggregate                Age
##          0.01808621         0.02019035         0.11422207
```

Hide
Code

```
## [1] -23.30005615   0.11932105   0.10297328   0.08705106  -0.14631
525
## [6]   0.30392784   0.01797615   0.01974855   0.11416655
```

Hide
Code



7 In Class Challenge Task

In this task you are to work with your peers to build a model for the critical temperature (`critical_temp`) of a superconductor. The *critical temperature* is the temperature (in Kelvin) below which the resistance in the conductor drops to zero (https://en.wikipedia.org/wiki/Superconductivity (https://en.wikipedia.org/wiki/Superconductivity)).

Tasks:

1. Download the data from the Moodle page. The primary dataset is called `train.csv` .
2. Explore the data: Each row is a different superconductor. There are 21,263 total superconductors in this dataset. The extra dataset `unique_m.csv` shows the chemical makeup of the superconductors if you're interested (not really necessary here).

3. **Primary ML Task:** Use stochastic gradient descent to build a multiple regression model to predict the critical temperature. Start with coefficients chosen from a random uniform distribution on the interval $[-100, 100]$.
4. Display a table showing your final coefficient values alongside the coefficient values found from `R`'s `lm` command. (Hint: just `cbind` them side by side)

8 Homework

1. Implement all of the different algorithms described above. You should write all of your code without looking at my solutions. If you're very stuck (after many hours of coding) then you can look at my code. At that point, though, you need to read the code only to get the idea, then hide it and write your own code.

2. Linear regression is a bit of a boring problem to do gradient descent on since the MSE function is quadratic and is therefore gauranteed to have exactly one global minimum (always!). This isn't the case in many different optimization problems. Consider the slighly ridiculous two-variable function $f(x, y) = \sin(xy)e^{-x^2-y^2}e^{-x}$. This function has one global minimum and several local minima. There are several ways to handle this type of optimization. Implement each of the following modified algorithms and report the location of the global minimum predicted from that algorithm.
    a. Start gradient descent at many different randomly generated starting points. Find the decision that gives you the minimum of all of the minimums of the function.
    b. Start gradient descent at one randomly chosen location. As the gradient gets close to zero start a new gradient descent starting nearby and see if that new estimate is the same as or different than the first one. This sort of algorithm basically says: if I'm near a minimum then throw a ball in a random direction and see if that ball rolls back to me. If it does not roll back to us then there is another minimum nearby and we should then see where the ball rolls to. If it rolls to a lower minimum we should go to that location, throw a ball, and repeat.
    c. At every N^{th} iteration (where N is 100 or 500 or 1000 or something like that) make the learning rate increase back to its original size and then keep stepping forward – allowing the new learning rate to start decaying as before. This allows for large steps later in the process and should allow you to escape local minima.
3. It is often very helpful to pre-process your data before using an algorithm like gradient descent. The scale of the values in each predictor might cause some headache in the gradient. Think of it like this: if one predictor is measured in square meters and has a range of 3 to 10 and another predictor is measured in dollars with a range of $100,000 to $1,000,000 then changes in the dollar amount will heavily dominate the gradient calculation as compare to changes in the areas. One of the best

ways to get around this is to do a z transformation to every column so that each column has mean 0 and standard deviation 1

$$z = \frac{x - \bar{x}}{\sigma}$$

a. Using the concrete data, start by doing the z transformation on all of the columns (not including the reponse variable). Then use gradient descent to find the parameters $\beta_0, \beta_1, \ldots, \beta_8$ such that

$$\text{Concrete\_compressive\_strength} = \beta_0 + \beta_1 z_1 + \beta_2 z_2 + \cdots \beta_8 z_8$$

where z_1 is the z-transformed `Cement` column, z_2 is the z-transformed `Blast_Furnace_Slag`, etc. Run your gradient descent for 1000 iterations with a fixed (small) learning rate and compare your results to what `R`'s `lm` command says the coefficients should be.

b. Now rerun gradient descent with 1000 iterations and the exact same learning rate on the original data. Compare to what `R`'s `lm` command says these coefficients should be.

c. Compare your answers from part (a) and part (b). The easiest might be to look at the percent error in each of the parameters for the two models and then to compare the individual percent errors. What do you notice about the gradient descent algorithm when applied to pre-processed data as compared to raw data? What is the takeaway here?

Note: In this numerical experiment you need to make the comparison fair. The starting point for gradient descent can greatly affect your results. Therefore, for parts (a) and (b) you should take the starting point as `R`'s `lm` coefficients rounded to 1 decimal place. If we do this the same for both the pre-processed and the non-pre-processed data we are at least putting the algorithms on a level playing field.