Computational Complexity: From OLS to Gradient Descent

[(https://github.com/learn-co-curriculum/dsc-computational-complexity)](https://github.com/learn-co-curriculum/dsc-computational-complexity) [(https://github.com/learn-co-curriculum/dsc-computational-complexity/issues/new/choose)](https://github.com/learn-co-curriculum/dsc-computational-complexity/issues/new/choose)

Introduction

In this lesson, you'll be introduced to computational complexity. You'll learn about this idea in relationship with OLS regression and see how this may not be the most efficient algorithm to calculate the regression parameters when performing regression with large datasets. You'll set the stage for an optimization algorithm called "Gradient Descent" which will be covered in detail later.

Objectives

You will be able to:

- Describe computational complexity and how it is related to Big O notation
- Describe why OLS with matrix algebra would become problematic for large/complex data
- Explain how optimizing techniques such as gradient descent can solve complexity issues

Complexities in OLS

Recall the OLS formula for calculating the beta vector:

$$\beta = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

This formula looks very simple, elegant, and intuitive. It works perfectly fine for the case of simple linear regression due to a limited number of computed dimensions, but with datasets that are very large or **big data** sets, it becomes computationally very expensive as it can potentially involve a huge number of complex mathematical operations.

For this formula, we need to find $(\boldsymbol{X}^T\boldsymbol{X})$, and invert it as well, which makes it very expensive. Imagine the matrix $X_{(N\times M+1)}$ has $(M+1)$ columns where M is the number of predictors and N is the number of rows of observations. In machine learning, you will often find datasets with $M > 1000$ and $N > 1,000,000$. The $(\boldsymbol{X}^T\boldsymbol{X})$ matrix itself takes a while to calculate, then you have to invert an $M \times M$ matrix which adds more to the complexity - making it very expensive. You'll also come across situations where the input matrix grows so large that it cannot fit into your computer's memory.

The Big O notation

In computer science, Big O notation is used to describe how "fast" an algorithm grows, by comparing the number of operations within the algorithm. Big O notation helps you see the worst-case scenario for an algorithm. Typically, we are most concerned with the Big O time because we are interested in how slowly a given algorithm will possibly run at worst.

Example

Imagine you need to find a person you only know the name of. What's the most straightforward way of finding this person? Well, you could go through every single name in the phone book until you find your target. This is known as a simple search. If the phone book is not very long, with say, only 10 names, this is a fairly fast process. But what if there are 10,000 names in the phone book?

At best, your target's name is at the front of the list and you only need to need to check the first item. At worst, your target's name is at the very end of the phone book and you will need to have searched all 10,000 names. As the "dataset" (or the phone book) increases in size, the maximum time it takes to run a simple search also linearly increases.

Big O notation allows you to describe what the worst case is. The worst case is that you will have to search through all elements (n) in the phone book. You can describe the run-time as:

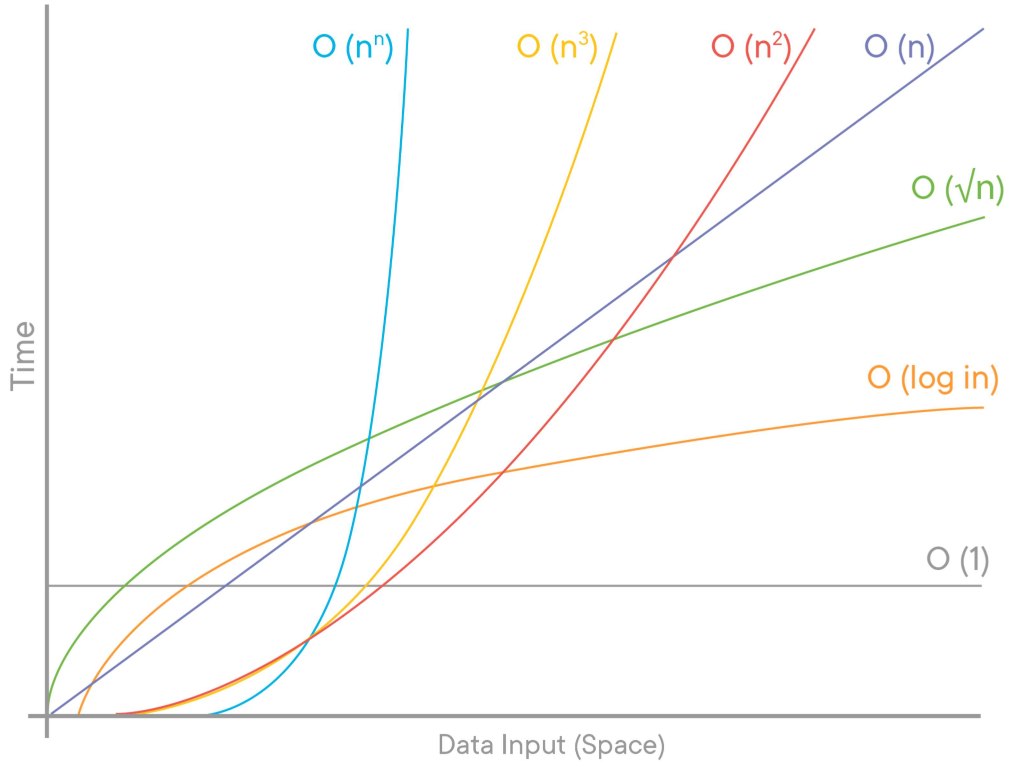
$O(n)$ **where n is the number of operations**

Because the maximum number of operations is equal to the maximum number of elements in our phone book, we say the Big O of a simple search is $O(n)$. **A simple search will never be slower than $O(n)$ time.**

Different algorithms have different run-times. That is, algorithms grow at different rates. The most common Big O run-times, from fastest to slowest, are:

- $O(\log n)$: aka $\log$ time
- $O(n)$: aka linear time
- $O(n^2)$
- $O(n^3)$

These rates, as well as some other rates, can be visualized in the following figure:



OLS and Big O notation

Inverting a matrix costs $O(n^3)$ for computation where n is the number of rows in X matrix, i.e., the observations. Here is an explanation of how to calculate Big O for OLS.

OLS linear regression is computed as $(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$.

If $\boldsymbol{X}$ is an $(\boldsymbol{n} \times \boldsymbol{k})$ matrix:

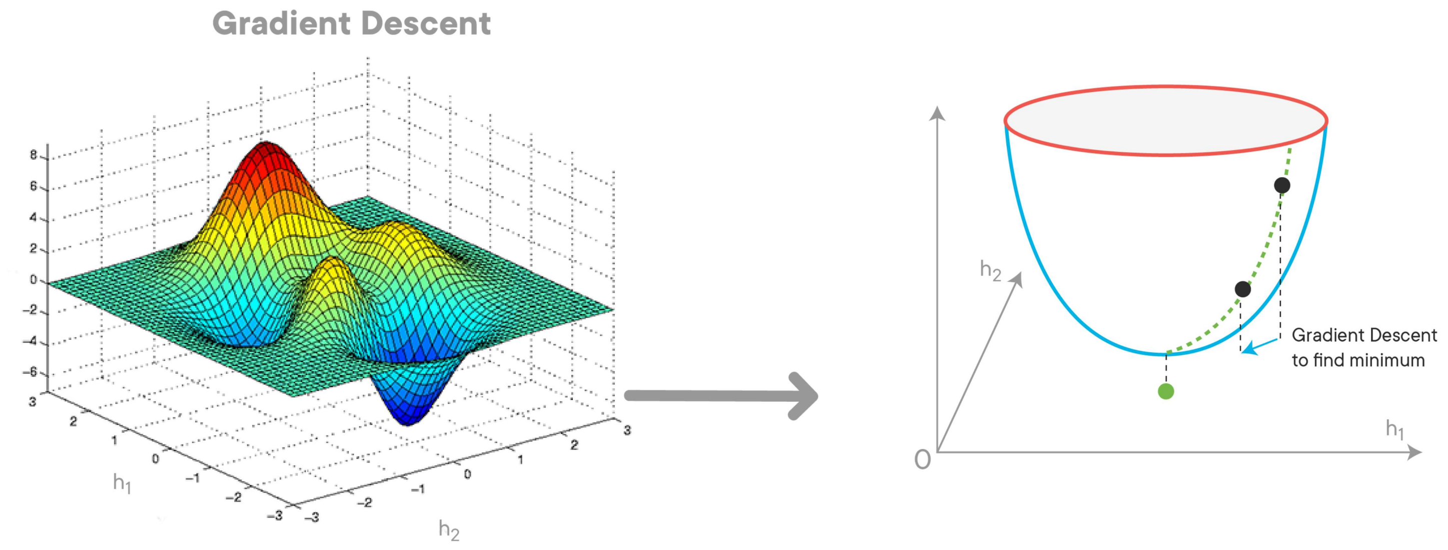
- $(\boldsymbol{X}^T\boldsymbol{X})$ takes $O(\boldsymbol{n} * \boldsymbol{k}^2)$ time and produces a $(\boldsymbol{k} \times \boldsymbol{k})$ matrix
- The matrix inversion of a (k x k) matrix takes $O(\boldsymbol{k}^3)$ time
- $(\boldsymbol{X}^T\boldsymbol{Y})$ takes $O(\boldsymbol{n} * \boldsymbol{k}^2)$ time and produces a $(\boldsymbol{k} \times \boldsymbol{k})$ matrix
- The final matrix multiplication of two $(\boldsymbol{k} \times \boldsymbol{k})$ matrices takes $O(\boldsymbol{k}^3)$ time

** So the Big O running time for OLS is $O(k^{2*(n+k)})$ - which is pretty expensive **

Moreover, if X is ill-conditioned (i.e. it isn't a square matrix), there will be computational errors in the estimation. Another common problem is overfitting and underfitting in estimation of regression coefficients.

So, this leads us to the gradient descent kind of optimization algorithm which can save us from this type of problem. The main reason why gradient descent is used for linear regression is the computational complexity: it's computationally cheaper (faster) to find the solution using the gradient descent in most cases.

Gradient Descent



> Gradient Descent is an iterative approach to minimize the model loss (error), used while training a machine learning model like linear regression. It is an optimization algorithm based on a convex function as shown in the figure above, that tweaks its parameters iteratively to minimize a given function to its local minimum.

In regression, it is used to find the values of model parameters (coefficients, or the β matrix) that minimize a cost function (like RMSE) as far as possible.

In order to fully understand how this works, you need to know what a gradient is and how is it calculated. And for this, you would need some Calculus. It may sound a bit intimidating at this stage, but don't worry. The next few sections will introduce you to the basics of calculus with gradients and derivatives.

Further Reading

- **Wiki: Computational complexity of mathematical operations** (https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations)
- **Simplified Big O notation** (https://medium.com/karuna-sehgal/a-simplified-explanation-of-the-big-o-notation-82523585e835)

Summary

In this lesson, you learned about the shortcomings and limitations of OLS and matrix inverses. You looked at the Big O notation to explain how calculating inverses and transposes for large matrix might make our analysis unstable and computationally very expensive. This lesson sets a stage for your next section on calculus and gradient descent. You will have a much better understanding of the gradient descent diagram shown above and how it all works by the end of next section.

How do you feel about this lesson?

Have specific feedback?

Tell us here! (https://github.com/learn-co-curriculum/dsc-computational-complexity/issues/new/choose)