I. Linear Model Recap & Extensions

Recall that our linear model is given by $\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$ and that our goal is to find values of $\boldsymbol{\beta}$ that best define the relationship between $\mathbf{Y}$ and $\mathbf{X}$.

- If we're being especially rigorous about it, we'd say that the values of $\boldsymbol{\beta}$ that best define the relationship between $\mathbf{Y}$ and $\mathbf{X}$ are the ones that minimize $\boldsymbol{\varepsilon}^2$.

We can show that the values of $\boldsymbol{\beta}_{OLS}$ are given by:

$$\hat{\boldsymbol{\beta}}_{OLS} = (\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y})$$

This can generalize if we want to do different types of regression. Consider weighted least squares regression. In this case, the values of $\boldsymbol{\beta}_{WLS}$ are given by:

$$\hat{\boldsymbol{\beta}}_{WLS} = (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{W}\mathbf{y})$$

If we want to fit a generalized linear model, we generally don't have the luxury of analytical solutions to find the optimal values of $\boldsymbol{\beta}$. So we use a method called iteratively reweighted least squares (IRLS) to optimize our values of $\boldsymbol{\beta}$. Glossing over most of the mathematical details here, we pick a starting point for our weights, calculate $\boldsymbol{\beta}_i$, update our weights, re-calculate $\boldsymbol{\beta}_{i+1}$, and so on until the difference between $\boldsymbol{\beta}_i$ and $\boldsymbol{\beta}_{i+1}$ is "small." Once we reach this pre-set stopping condition, we use that value for $\boldsymbol{\beta}$.

$$\hat{\boldsymbol{\beta}}_{i+1} = (\mathbf{X}^T\mathbf{W}_i\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{W}_i\mathbf{y})$$

ADDED FOR COMPLETENESS: There is the idea of a generalized inverse that allows us to fit a model even when the matrix $(\mathbf{X}^T\mathbf{X})$ does not have an inverse. In this case, the values of $\boldsymbol{\beta}_{GLS}$ (GLS indicates general least squares) are given by:

$$\hat{\boldsymbol{\beta}}_{GLS} = (\mathbf{X}^T\mathbf{X})^{-}(\mathbf{X}^T\mathbf{y})$$

The reason we don't discuss these is because the estimates are not unique. For any noninvertible matrix, we can come up with an infinite number of generalized inverses and thus infinitely many different values of $\hat{\boldsymbol{\beta}}_{GLS}$.

What we notice with each of these is that we're multiplying matrices together and inverting matrices. Being able to do this in a computationally efficient manner is incredibly important - especially when we have a large number of observations n and/or a large number of features p.

II. Computational Efficiency

We've spoken at length about computational efficiency and its relevance.

- What are some examples of times when we've discussed computational efficiency?

In model building, as above, we're multiplying and inverting many matrices. While in 2×2 cases this may seem efficient (although tedious by hand...), if we consider a real-world situation with a large number of observations n and/or a large number of features p, we can easily see how computational complexity can be a really important consideration.

We generally describe complexity of a particular algorithm as $O(\cdot)$ ("Big Oh of $\cdot$"), where the $\cdot$ changes depending on how complex the algorithm is.

- $O(1)$, or "Big Oh of 1" describes an algorithm that runs in exactly the same amount of time regardless of the input. (An example of this would be to print the first element of a list. Regardless of the size of the list, the time to print the first element will be the same.)

- $O(N)$, or "Big Oh of N" describes an algorithm that increases in time proportionally with the increase in the size of the input. For example, if you add one new observation to your data and your run-time increases by k units, adding c new observations should increase your run-time by ck units. (An example of this would be a single `for` loop.)
- $O(N^2)$, or "Big Oh of N-squared" describes an algorithm that increases in time proportionally with the square of the increase in the size of the input.. For example, if you add one new observation to your data and your run-time increases by k units, adding c new observations should increase your run-time by c^2k units. (An example of this would be nested `for` loops.)

Here are a few resources to help more explicitly define this concept:

- Big-O Cheat Sheet
- Blog Post about Big-O
- Wikipedia Article on Big-O Notation
- Intermediate Discussion of Algorithmic Complexity
- Mathematical Discussion of Algorithmic Complexity

How can you assess your computational complexity? This can be done in the following way:

```
import time

t0 = time.time()
{code_block}
t1 = time.time()

total = t1-t0
print total
```

III. Floating Point Arithmetic

Floating point arithmetic, or "computer math," as I like to call it, is important to understand. Because computers have certain limitations, the numbers we store may be imprecise. This can cause issues when numbers are very, very large or very, very small. (From the linked Web site above: "It's actually pretty simple. When you have a base 10 system (like ours), it can only express fractions that use a prime factor of the base.")

IV. Linear Algebra & Computational Challenges

Within the realm of linear algebra, there are many "tricks" that we consider to make our lives easier. These are often clever algorithms that simply reduce the number of computations needed to execute some command.

Formally, a matrix $\mathbf{A}$ can be decomposed into matrices $\mathbf{X}$ and $\mathbf{Y}$ if $\mathbf{A} = \mathbf{XY}$.

While this seems counterintuitive, there are particular benefits to decomposing matrices.

- When solving the system of equations $\mathbf{Ax} = \mathbf{b}$, applying the LU decomposition to $\mathbf{A}$ decomposes $\mathbf{A}$ into $\mathbf{L}$ and $\mathbf{U}$, where $\mathbf{L}$ is a lower-triangular matrix and $\mathbf{U}$ is an upper-triangular matrix. (This exists for many square matrices; a slight variant called the LUP decomposition exists for all square matrices.)
  - Computers can use the properties of triangular matrices to invert $\mathbf{LU}$ much more quickly than $\mathbf{A}$.
- The Cholesky decomposition of $\mathbf{A}$ into $\mathbf{VV^T}$ where $\mathbf{V} = \mathbf{LD}^{1/2}$, $\mathbf{L}$ is lower-triangular and $\mathbf{D}$ is a diagonal matrix, can be used for any square, symmetric, positive definite matrix $\mathbf{A}$.

  - A matrix $\mathbf{A}$ is positive definite if $\mathbf{xAx^T} > 0$ for all nonzero vectors $\mathbf{x}$. Positive definiteness ensures nice properties like invertibility of certain matrices.
  - While the Cholesky decomposition can only be applied to a subset of the matrices to which the LU decomposition can be applied, the Cholesky decomposition is roughly twice as computationally efficient.
- The spectral decomposition (also called the eigendecomposition) of a matrix $\mathbf{A}$ is where $\mathbf{A} = \mathbf{PDP}^{-1}$ with $\mathbf{D} = \text{diag}\{\lambda_1, \ldots, \lambda_n\}$ and $\mathbf{P}$ consisting of the eigenvectors corresponding to the eigenvalues in $\mathbf{D}$.
  - Suppose we want to find $\mathbf{A}^k$. (We'll do this on the board.)
  - Spectral decomposition will also be very important in principal component analysis.
  - This works for any diagonalizable (also called "diagonable") matrix, which means there are n independent eigenvectors.