CMSC 320

Introduction to Data Science

Akilesh Praveen

Prof. John Dickerson • Fall 2020 • University of Maryland

https://cmsc320.github.io

Last Revision: May 5, 2020

Table of Contents

---

1 Notes & Preface

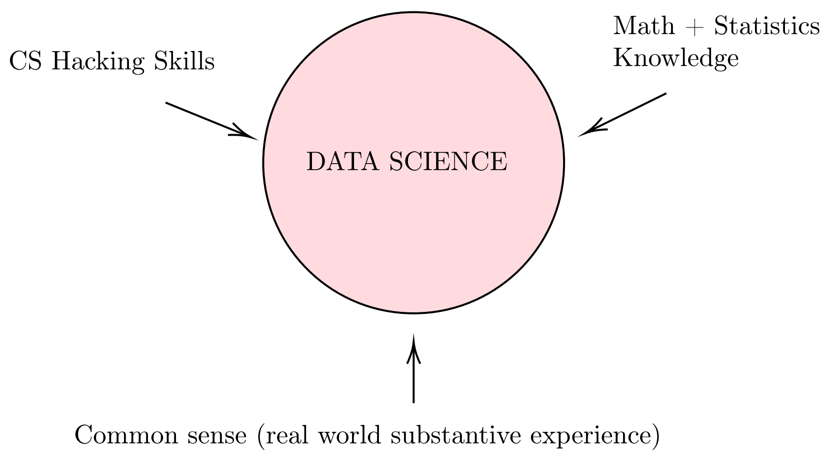
Course notes for CMSC320, under Prof. John Dickerson. Notes collected from previous and current lectures.

2 Lecture 1

What is Data Science?

Data Science is the application of computation and statistical techniques to address or gain insight. It's the intersection of statistics and Computer Science. Based on what I've learned thus far, learning to do data science is like learning how to use a TI-84 in statistics class. You're simply learning how to leverage programming tools in order to perform advanced, complex, and meaningful data-related operations.

It's the use of statistics and computer science in order to find real-world insights.



Topics

Here are the general topics that this class will cover.

- Processing data
- Visualizing data
- Understanding data
- Communicating data
- Extracting value from data

Tools

Here are some tools commonly employed by data scientists. We'll try to cover how to use most of them here.

- Python
- Scikit-Learn

- Docker
- PANDAS
- Spark
- TensorFlow

Conda

Conda is a package and environment manager for python that we can use with the command line. We can create multiple environments for us and install separate packages in each of them. This will be highly useful to us, as we sometimes want to consolidate the tools we use into separate environments.

3 Lecture 2

Definition:

Data Collection → The process of measuring and gathering information on targeted variables.

Literate Programming

The idea of **literate programming** is that you have the source code, an explanation of the source code, and the end result of running the code all in one file. Usually, this file is identified as a *notebook*. In other words, the syntax is no different from regular code, you just get a more organized way to show off tables, plots, and other outputs generated from your code.

Jupyter Notebook + Alternatives

Jupyter Notebook is a service that started off as `iPython`, but it's basically a web-based platform that we use for literate programming. Specifically, it supports Python-based literate programming. Most data scientists prefer it, and it can also apparently leverage big data tools, such as Apache Spark.

It saves files in `.ipynb` format, which most platforms (i.e. GitHub) have built in viewers for. Options to export in other readable formats are available. Basically, it's just Python with a bunch of bells and whistles on top to make the output of your code look pretty.

Apache Zeppelin is an alternative data analysis tool, but we will stick to Jupyter for our purposes. This is because Jupyter seems to be preferred in industry.

RStudio is the equivalent, for people who prefer to use the `R` programming language for data science.

This course will be centered around Jupyter Notebook.

List Comprehensions in Python

To make lists in Python, you can use loops or the `map()` function, but a *pythonic* way of doing this would be to use a list comprehension. Below is a simple example.

> **Example:** Make a list of all the squares of `{0,1,2,3,4,5,6,7,8,9}`
>
> **List Comprehension:**
> `squares = [i * i for i in range(10)]`

A good way of thinking about this is that it allows you to build sets like a mathematician. This is a common theme in data science, where we can find the intersection between a lot of math stuff and computer science stuff. It's good to know how lists are generated in a mathematical sense in Python for that reason. Here's an example where we translate mathematical notation into a Python list comprehension.

> **Example:** Make a list of all odd natural numbers from 0 to 999
>
> **Math Notation:**
> $E = \{x \mid x \text{ in } \mathbb{N} \wedge x \text{ is odd } \wedge x < 1000\ \}$
>
> **List Comprehension:**
> `E = [x for x in range(1000) if x % 2 != 0]`

Using Python3

We will use Python3. Since I used Python2 during my internship, I'm going to note some big changes to keep track of.

- Python3 is backwards incompatible. (Don't write in Python2!)
- Print has changed from a command to a function, so make sure to use proper function notation when invoking it.
- Division has changed. `1/2` no longer equals 0. `1/2 == 0.5` and floored division is now taken care of this way: `1//2 == 0`

Python vs. R for Data Scientists

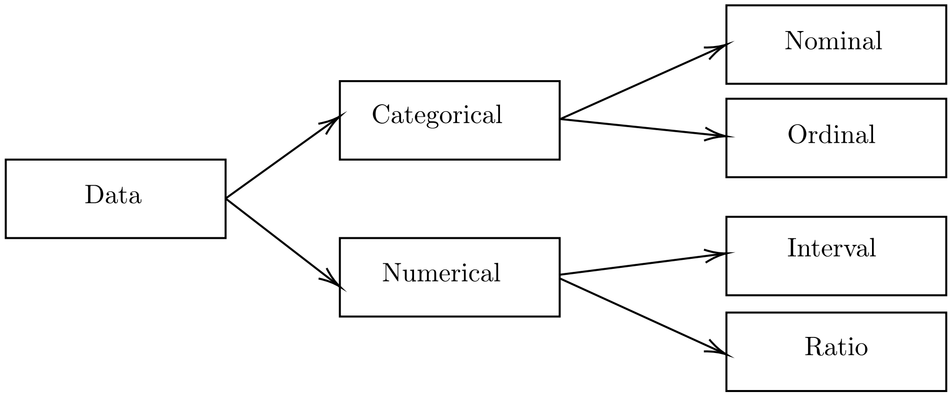
Some arguments for both sides in terms of what to use.

- Python is a 'full' programming langauge. Also, if you've got prior experience with Python paradigms or just programming in general, that's a big plus in terms of learning curve.
- R has more mature 'pure statistics' libraries, but Python is apparently catching up.
- In terms of **processing speed**, R is certainly faster. It was designed and optimized for statistics processing.
- Python is preferable for machine learning operations, which is pretty big right now.

My personal choice will be to use Python as much as I can when I'm studying this course. Since it's more prominent in the tech industry, I should be using it more anyway.

The Classic Statistical View of Data

There are **four** main types of data: Nominal, Ordinal, Interval, and Ratio data. They can each be classified under two main subgroups, Categorical and Numerical data. Here's a visualization.



Nominal Data

A type of categorical data, nominal data value have names and describe the state of things. For example, your marriage status is nominal data because you can either be *single*, *married*, or *separated*. Another example is the type of drink you're going to have. Will it be *Milk*, *Beer*, or *Juice*?

The key here is that there can be no quantitative values assigned to each of these categories, as that would allow us to do math with them and would defeat the purpose of these labels. These values **cannot be easily compared**, so they have no material value. *E.g. being single is not quantitatively better than being married (objectively), and vice versa.*

> **Example:** What is your marital status?
>
> - Married
> - Divorced (separated)
> - Single

Ordinal Data

Ordinal data represents values that have names that describe the state of things, but in this case, there **is** an ordering of those values. This is what sets it apart from nominal data.

> **Example:** What did you think of the movie?
>
> - Strongly liked
> - Liked
> - Indifferent
> - Disliked
> - Strongly Disliked

Given how subjective some of these things can be, the distinction between nominal and ordinal can be **blurry** at times. For example, going back to our nominal example, some people may think that being single is quantitatively better than being married.

Interval and Ratio Data

Interval and Ratio data are pretty similar, and both can be used to measure things that can be represented by either integers or real numbers.

Interval data scales with fixed but arbitrary values. That might sound silly, but a good example is **dates**. Below is an example of two data comparisons of interval data that seem arbitrary, but indeed hold integer value.

> **Example:** The following two operations are equal.
>
> ```
> 10/1/2019 - 9/1/2019
> 10/1/2018 - 9/1/2018
> ```

The measures don't look like integer values at first, but we can quantify them by marking them with days.

Here's what sets **Interval** data apart, however. You have **no method** of computing ratios or scales with it. For example, never mind that you can try computing `(9/1/2019` $\times$ `8/25/2015)`, the unit of the answer would be totally useless to us, and neither would the actual number, even if you went ahead with the operation.

Ratio data is, in essence, the same as interval data in that it is numerical, but the scale itself **has a true zero**. While dates don't necessarily have a true zero, we can say that money counts as ratio data. For example, having zero money means that you're at the absolute zero of that scale, whereas the absolute zero for dates is disputable. Are we saying we're starting at O A.D.? The Big Bang? Even earlier?

Differentiating between the two is usually a case-by-case basis thing, which is what I'm thinking is the best way to handle any conflicts I end up running into between ratio and interval data.

> **Example:** Interval data
>
> Temperature on the scale of Celsius or Fahrenheit is interval-type data because $0°$ is set to an arbitrarily fixed point. Also, we can't scale it properly- $30°F$ isn't twice as hot as $15°F$.
>
> **Example:** Ratio data
>
> Temperature on the Kelvin scale is ratio data. $0K$ is set at legitimate absolute zero, and $50K$ is truly twice as cold as $100K$.

Data Science at a Glance

Data science is basically manipulating and computing using data. As such, we need to shift our thinking from writing **imperative** code to manipulate **data structures** to creating **sequences and pipelines** to conduct operations on **data**. That stuff is covered more in 420 and 424, for reference.

More often than not, we have to take the data that we've found and make it easily understandable for humans. This is called Data Representation.

> **Definition:**
>
> **Data Representation** → The natural way to think about data, in a human way.

Here are some ways that we think about data in this class:

- **One Dimensional Arrays** → E.g. `<'red', 'blue', 'green'>` or `<0,3,4>`. We can use functions like map, fold and filter to manipulate these.
- **N-Dimensional Arrays** → Also known as **tensors**.
  - For example, a Tensor of dimensions `[6,4]` is just a 6×4 matrix.
  - Similarly, a Tensor of dimensions `[4,4,2]` is a 3D array.
  - Here, we can start to make use of **Linear Algebra** for further data manipulation. Some example operations that we can use to mess with tensors:
    - Matrix/Tensor Multiplications
    - Transpose
    - Vector Multiplication
    - Matrix Factorization (we will explore this later)
- **Sets** of objects, or **Key/Value Pairs**
- **Tables/Relations** → This goes into relational databases, which is the basis of `SQL`. We'll go into this later.
- **Hierarchies/Trees/Graphs** → This sort of spills over into data structures, but they've got some additional nuances included with them.
  - They tend to make use of 'path' queries
  - Graph and Tree Algorithms will be useful here, efficiency is key
  - Example: networks are represented this way, we'll cover that later in this class

4 Lecture 3

Acquiring Data

Here are some examples of how we can grab data from places. Pretty obvious, common sense stuff. We're going to explore all of these as we move forward.

- Direct download from online or loading it from local storage
- Generate the data locally via a simulation or equivalent program
- Query data from a database
- Query data from an API
- Scrape data from a website

When you pull from APIs, you're going to want to be using HTTP Requests.

RESTful APIs

This stands for REpresentational State Transfer APIs, and it's basically a standard that enforces that APIs do a few things. It says that they should support these basic operations:

- `GET` → Query a data entry
- `POST` → Create a new data entry
- `PUT` → Update an existing data entry
- `DELETE` → Delete an existing data entry

RESTful APIs are also supposed to be stateless. That is, with every API request, you send a token of who you are, and you get a current capture of the data at that time/edit the data.

A good example of a REST API is Github, where you can use REST API calls on your repositories.

There are other guiding principles and miscellaneous guidelines for RESTful APIs, which can all be found at `https:/restfulapi.net`

Aside: GRAPHQL

GraphQL → REST has been adopted by many developers and is widely regarded as the traditional way to send data over HTTP. GraphQL, on the other hand, is a revolutionary new player that's presented as a way to *replace* legacy REST APIs *(back4apps)*

Oauth

If you want to grant an app access to your identity without actually giving it your username and password, is there a way to do that? The answer is **yes**, because this is a common software engineering problem.

OAuth is the standard for *access delegation* used for internet users to grant websites access to their information on other sites. A pretty good example of this is Google's sign in page on other websites. How do you think other websites conduct sign in without knowing your password for your Google account?

GET Requests

Assume we used Python's `requests` module to query a server with a `GET` request.

First, we'd get either a `CSV`, `JSON`, or `HTML/XML/XHTML` file back, in response. This is the data that we have to sift through. *Note:* You might also get a domain-specific file, like an **rvt** file. You're always welcome to make your own filetype for storing data, but make sure it's actually documented somewhere.

Aside: Parsing CSVs and JSON

Never write your own `CSV` or `JSON` parsers. This is another example of reinventing the wheel. We'll use Python Libraries to do this more easily. *E.g. PANDAS*

More on Data Storage Formats

Definition:

Serialization → The process of converting objects into strings.
Deserialization → The process of converting strings back into objects.

`JSON` is a pretty common format for serializing objects. Plus, serializing objects makes it easier for humans to read and perform sanity-checks on. In Python, `JSON` is built with Strings, Lists, Dictionaries, and sometimes mixes of a few of those together.

Definition:

Document Object Model → A tree-based data storage method. For example, HTML is structured this way.

SAX

SAX is a lightweight way to process XML. It generates a stream of events as it parses an XML file. IT helps us pay attention to individual parts of an XML file without having to process through the rest of it.

Parsing HTML

Parsing HTML is the hardest to do in this case, as I've seen many times before in hackathon projects. Although HTML's specifications are pure, the real world examples of it are pretty nightmarish, thanks to how it interacts with JavaScript and loads dynamic content. All in all, it's fairly unreliable in terms of parsing it manually.

In this case, we're best off using the Python library `BeautifulSoup`. We can also make use of Python's Regex, which is similar enough to Ruby regex that we worked with in 330. A website like Rubular-`https://pythex.org` will be useful in this case.

By combining `BeautifulSoup`, Regular expressions, and `GET` requests, we can make the process fairly streamlined. This is usually what we'll be using to scrape websites. In order to scrape more dynamic websites, we'd probably have to make use of Selenium. Check my 320 folder to find an example of a simple webscraper with `BeautifulSoup`.

5 Lecture 4

Overview: Numpy, PANDAS, Relational Databases, Apache Spark

Available Technologies

Python has a bunch of 3rd party packages for scientific and numerical computation. Some examples are..

- **`Numpy and Scipy`** → Numerical and scientific function libraries.
- **`NUMBA`** → A Python compiler supporting 'Just in Time' compilation. That is, it supports compilation of code while code is running.
- **`ALGLIB`** → A cross-platform numerical analysis library
- **`PANDAS`** → An extensive data analysis tool with some neat built-in data structures
- **`PyGSL`** → GNU Scientific Library in Python
- **Scientific Python** → A collection of scientific computing modules for Python

These are a bunch of examples of what's available to developers right now, but we won't focus on all of it. Particular emphasis will be placed on `Numpy` and `PANDAS`.

NumPy Stack

The **NumPy** stack is the most commonly used out of all of these packages. It includes the following:

- `Numpy` - Works sort of like MatLab, just lets us handle a lot of number manipulation and mathematical operations
- `Matplotlib` - This is a plotting and graphing library
- `PANDAS` - This gives us a bunch of data structures and data analysis tools to play with/keep track of our data. (Usually, you'll want to import your data into a `PANDAS dataframe` or something.)
- `SciPy`
- `SymPy`
- `Jupyter` - This will be our medium for **literate programming**.

To see more about this stuff, search Google for the **NumPy Stack** and you'll find everything you need.

Misc About NumPy

Here are a few more notable things about `Numpy`:

- It contains the **n-dimensional array** object
- It contains 'sophisticated' functions that we can use
- It provides us with excellent tools to integrate `C++`, `C`, and even `FORTRAN`
- It has math capabilities that are highly useful to us (e.g. Linear Algebra, Fourier Transform, etc)
- `Numpy` also comes with a bunch of new DataTypes for us to use.

Aside: Numpy Arrays

Arrays in `Numpy` are different from regular lists in Python, so make sure your syntax is correct and you know the difference when you decide to use either one in practice.

Linear Algebra with NumPy

One of `NumPy`'s most common uses lies within its **Linear Algebra** module. It allows us to do regular LA stuff, like `.transpose()` and `.inverse()` to matrices stored as n-dim arrays. Here's an example.

```
# Note: remember, we have to use NumPy's n-dimensional array ↩
    object here

array([[1.0, 2.0],
       [3.0, 4.0]]).transpose()
```

SciPy

`SciPy` includes various tools and functions for solving common problems in **scientific computing**.

We won't use it much for now, but it's supposed to be good to know. Often you'll be able to find higher-level `Scipy` functions that will work around the need to call lower-level `Scipy` functions. It's got a lot of functionality built in, so make sure not to overlook it.

The Idea of Reproducibility

Starting from the same dataset, can we reproduce your analysis and get the same results? **This is the goal that we're trying to fulfill with our analysis-** we want our stuff to be reproducible! (Otherwise, what exactly does it even mean?)

Best Practices

Honestly, most of this stuff should be common sense.

- Use version control to keep track of code. (e.g. `git`)
- Use unit testing. (e.g. `unittest` module in python)
- Use libraries when you can. (don't reinvent the wheel!)

The Idea of Open Data

Some data should be widely available for everyone to use as they want, without restrictions from copyright, etc.

This is probably where all of our free data comes from, so this idea is super helpful to us as data scientists.

General Process

Here's the general process for data science- just so we have an idea of what's going on.

Data Acquisition

Acquire the data, then put it in a usable format.

→

Algorithm / Tool Development

If new tools are required to analyze your data, create them.

If new algorithms may be needed to analyze it, synthesize them.

→

Data Acquisition

Use the tools you've built / setup and the algorithms you've created or implemented to analyze the data.

After we do that, we still technically have some programming left. In this new era of literate programming, there's one more step of processing we have to do with our results in order to make them publicly presentable and meaningful.

Communication of Results

Use the principles of **literate programming** to provide results, plots, and publication regarding your data science experiment. This gives you the ability to explain your choices and clearly interpret your results.

It's emphasized a lot here to think like an **algorithm developer**, as you're going to need efficiency in the data analysis that you perform. However, you also need to think like an experiment-conducting **data scientist**. We don't usually get enough training as the latter, so hopefully this course should be an introduction to that sort of stuff.

6 Footnotes

Taken by Akilesh Praveen.