

# Data 100

## Lecture 9: Scraping Web Technologies

Slides by:

Joseph E. Gonzalez, Deb Nolan

[deborah.nolan@berkeley.edu](mailto:deborah.nolan@berkeley.edu)

[hellerstein@berkeley.edu](mailto:hellerstein@berkeley.edu)



Last Week ...

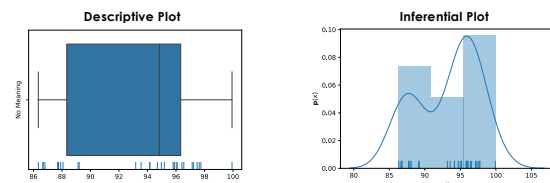
## Visualization

- Tools and Technologies
  - Matplotlib and seaborn
- Concepts
  - Length, color, and faceting
- Kinds of visualizations
  - Bar plots, histograms, rug plots, box plots, violin plot, scatter plots, and kernel density estimators
- Good vs bad visualizations

## Kernel Density Estimates and Smoothing

## Kernel Density Estimators

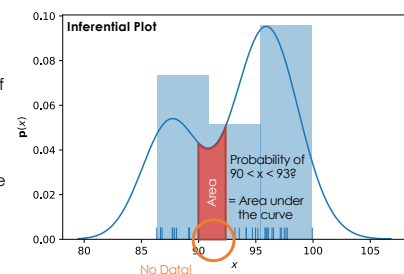
- Inferential statistics – estimate "shape" of the population
  - Draw conclusions beyond the data...



- Inferential statistics – estimate "shape" of the population
  - Draw conclusions beyond the data...

Suppose this data was constructed by a **random sample** of student grades?

What is the probability that the next student's grade will be between 90 and 93?



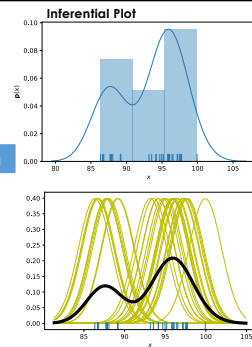
## Constructing KDEs

- Non-parametric Model
- "Size/complexity of the model depends on the data:

$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^n K_{\alpha}(x - x_i)$$

Gaussian Kernel: (Commonly used → Very smooth):

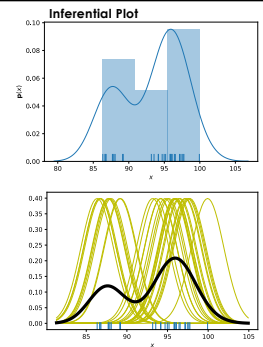
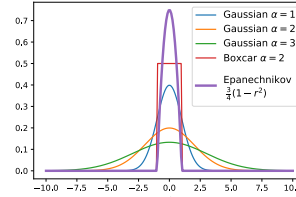
$$K_{\alpha}(r) = \frac{1}{\sqrt{2\pi}\alpha^2} \exp\left(-\frac{r^2}{2\alpha^2}\right)$$



$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^n K_{\alpha}(x - x_i)$$

Gaussian Kernel: (Commonly used → Very smooth):

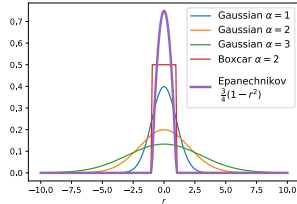
$$K_{\alpha}(r) = \frac{1}{\sqrt{2\pi}\alpha^2} \exp\left(-\frac{r^2}{2\alpha^2}\right)$$



$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^n K_{\alpha}(x - x_i)$$

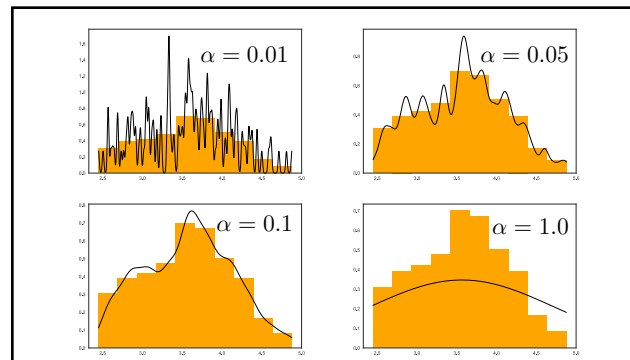
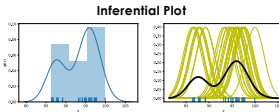
Gaussian Kernel: (Commonly used → Very smooth):

$$K_{\alpha}(r) = \frac{1}{\sqrt{2\pi}\alpha^2} \exp\left(-\frac{r^2}{2\alpha^2}\right)$$

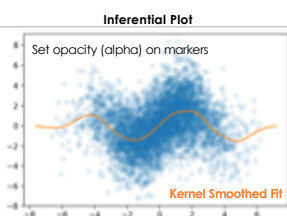
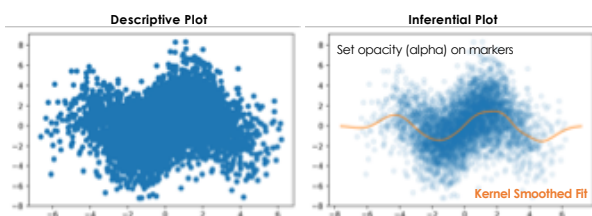


How do you pick the kernel and bandwidth?

- **Goal:** fit unseen data
- **Idea:** Cross Validation
  - Hide some data
  - Draw the curve
  - Check if curve "fits" hidden data ... more on this later



## Smoothing a Scatter Plot

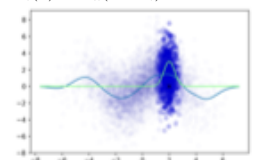
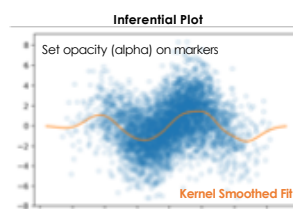


## Smoothing a Scatter Plot

- Weighted combination of all y values

$$\hat{y}(x) = \frac{1}{\sum_{i=1}^n w_i(x)} \sum_{i=1}^n w_i(x) y_i$$

$$w_i(x) = K_{\alpha}(x - x_i)$$



## Dealing with Big Data

- **Big n** (many rows)
  - Aggregation & Smoothing – compute summaries over groups/regions
    - Sliding windows, kernel density smoothing
  - Set transparency or use contour plots to avoid over-plotting
- **Big p** (many columns)
  - Faceting – Using additional columns to
    - Adjust shape, size, color of plot elements
    - Breaking data down by auxiliary dimensions (e.g., age, gender, region ...)
  - Create new hybrid columns that summarize multiple columns
    - **Example:** total sources of revenue instead of revenue by product

## What's Next ...

## This Week

- Today (Tuesday)
  - Web technologies -- getting data from the web
    - HTTP – Get and Post
    - REST APIs, Scraping
    - JSON and XML
- Thursday
  - Both Fernando and I are out → guest lecturer Sam Lau!!
  - String processing
    - Python String Library
    - Regular Expressions
    - Pandas String Manipulation

## Getting Data from the Web

Starting Simple with Pandas

## Pandas `read_html`

- Loads tables from web pages
  - Looks for `<table></table>`
  - Table needs to be **well formatted**
  - Returns a **list** of dataframes
- Can load directly from URL
  - Carefull! Data changes. Save a copy with your analysis
- You will often need to do additional transformations to prepare the data
- Demo!

## HTML, XML, and JSON

data formats of the web

## HTML/XML/JSON

- Most services will exchange data in HTML, XML, or JSON
- Why?
  - Descriptive
    - Can maintain meta-data
  - Extensible
    - Organization can change and maintain compatibility
  - Human readable
    - Useful for debugging and provides a common interface
  - Machine readable
    - A wide range of technologies for parsing

## JSON: JavaScript Object Notation

```

{
  "Prof": "Gonzalez",
  "Classes": [
    { "Name": "Data100", "Year": 2017, 2018 },
    { "Name": "Data100", "Year": 2017, 2018 }
  ],
  "Tenured": false
}

```

- Recursive datatype
  - Data inside of data
- **Value** is a:
  - A basic type:
    - String
    - Number
    - true/false
    - Null
  - Array of Values
  - A dictionary of key:Value pairs
- Demo Notebook

## XML and HTML

eXtensible Markup Language

```

1 <CATALOG>
2 <PLANT>
3 <COPRQ>Blodroot</COPRQ>
4 <BOTANICA>Sanguinaria canadensis</BOTANICAL>
5 <DOSE>4</DOSE>
6 <LIGHT>Mostly Shady</LIGHT>
7 <PRICE currency="USD">$2.44</PRICE>
8 <AVAILABILITY>$31399</AVAILABILITY>
9 </PLANT>
10 <PLANT>
11 <COPRQ>Columbine</COPRQ>
12 <BOTANICA>Aquilegia canadensis</BOTANICAL>
13 <DOSE>3</DOSE>
14 <LIGHT>Mostly Shady</LIGHT>
15 <PRICE currency="USD">$8.37</PRICE>
16 <AVAILABILITY>$30699</AVAILABILITY>
17 </PLANT>
18 <PLANT>
19 <COPRQ>Marsh Marigold</COPRQ>
20 <BOTANICA>Caltha palustris</BOTANICAL>
21 <DOSE>4</DOSE>
22 <LIGHT>Mostly Sunny</LIGHT>
23 <PRICE currency="CAD">$6.81</PRICE>
24 <AVAILABILITY>$51799</AVAILABILITY>
25 </PLANT>
26 </CATALOG>

```

XML is a standard for semantic, hierarchical representation of data

## Syntax : Element / Node

The basic unit of XML code is called an "element" or "node"

Each Node has a start tag and end tag

```

      ↗
    <zone>4</zone>
      ↘
Start tag      Content      End tag

```

## Syntax : Nesting

A node may contain other nodes (children) in addition to plain text content.

```

      ↗
    <plant>
      ↘
    <zone>4</zone>
      ↘
    <light>Mostly Shady</light>
      ↘
    </plant>

```

Content consists of two nodes

Indentation is not needed. It simply shows the nesting

## Syntax : Empty Nodes

Nodes may be empty

```
<plant>
  <zone></zone>
  <light/>
</plant>
```

These two nodes are empty  
Both formats are acceptable

## Syntax : Attributes

Nodes may have attributes (and attribute values)

```
<plant id='a'>
  <zone></zone>
  <light source="2" class="new"/>
</plant>
```

The attribute named type has a value of "a"

This empty node has two attributes: source and class

## Syntax : Comments

Comments can appear anywhere

```
<plant>
  <!-- elem with content -->
  <zone>4 <!-- a second comment --></zone>
  <light>Mostly Shady</light>
</plant>
```

Two comments

## Well-formed XML

- An element must have both an **open** and **closing** tag. However, if it is empty, then it can be of the form `<tagname/>`.
- Tags must be **properly nested**:
  - Bad!: `<plant><kind></plant></kind>`
- Tag names are case-sensitive
- No spaces are allowed between `<` and tag name.
- Tag names must begin with a letter and contain only alphanumeric characters.

## Well-formed XML:

- All **attributes** must appear in quotes in:
 

```
name = "value"
```
- Isolated markup characters must be specified via entity references. `<` is specified by `&lt;`; and `>` is specified by `&gt;`.
- All XML documents must have one *root node* that contains all the other nodes.

## xHTML: Extensible Hypertext Markup Language

- HTML is an XML-"like" structure → Pre-dated XML
  - HTML is often not well-formed, which makes it difficult to parse and locate content,
  - Special parsers "fix" the HTML to make it well-formed
    - Results in even worse HTML
- xHTML was introduced to bridge HTML and XML
  - Adopted by many webpages
  - Can be easily parsed and queried by XML tools

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
2 www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4 <head>
5   <meta http-equiv="Content-Type" content="text/html; charset=utf-8"
6   />
7   <title>Example Website</title>
8 </head>
9 <body>
10  <div id="people">
11    <div class="person" id="jgonzal">
12      <div class="name">Joey</div>
13      <div class="address">jgonzal@berkeley.edu</div>
14    </div>
15    <div class="person" id="fperez">
16      <div class="name">Fernando</div>
17      <div class="address">fperez@berkeley.edu</div>
18    </div>
19  </div>
20 </body>
21 </html>

```

Example of well formed XHTML

## DOM: Document Object Model



- Treat XML and HTML as a Tree
- Fits XML and well formed HTML
- Visual containment → children
- Manipulated dynamically using JavaScript
  - HTML DOM and actual DOM the browser shows may differ (substantially)
  - Parsing in Python → Selenium + Headless Chrome ... (out of scope)

## Tree terminology

- There is only one *root* (AKA *document node*) in the tree, and all other nodes are contained within it.
- We think of these other nodes as *descendants* of the root node.
- We use the language of a family tree to refer to relationships between nodes.
  - *parents, children, siblings, ancestors, descendants*
- The *terminal nodes* in a tree are also known as *leaf nodes*. Content always falls in a leaf node.

## HTML trees: a few additional “rules”

- Typically organized around `<div>` `</div>` elements
- Hyperlinks: `<a href="uri">Link Text</a>`
- The *id* attribute: unique key to identify an HTML node
  - Poorly written HTML → not always unique
- Older web forms will contain forms:

```

<form action="/submit_comment.php" method="post">
  <input type="text" name="comment" value="blank" />
  <input type="submit" value="Submit" />
</form>

```

See notebook for demo on working with forms ...

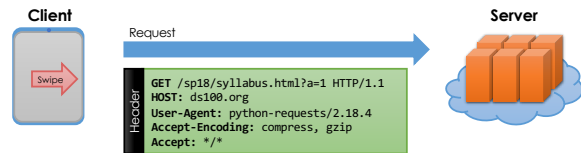
## HTTP – Hypertext Transfer Protocol



## HTTP Hypertext Transfer Protocol

- Created at CERN by Tim Berners-Lee in 1989 as part of the World Wide Web
- Started as a simple **request-response protocol** used by web servers and browsers to access hypertext
- Widely used exchange data and provides services:
  - Access webpage & submit forms
  - Common API to data and services across the internet
- Foundation of modern REST APIs ... (more on this soon)

## Request – Response Protocol



First line contains:

- `GET /sp18/syllabus.html?a=1 HTTP/1.1`
- a method, e.g., `GET` or `POST`
- a URL or path to the document
- the protocol and its version

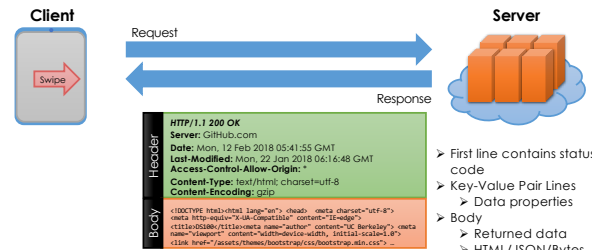
Remaining Header Lines

- Key-value pairs
- Specify a range of attributes

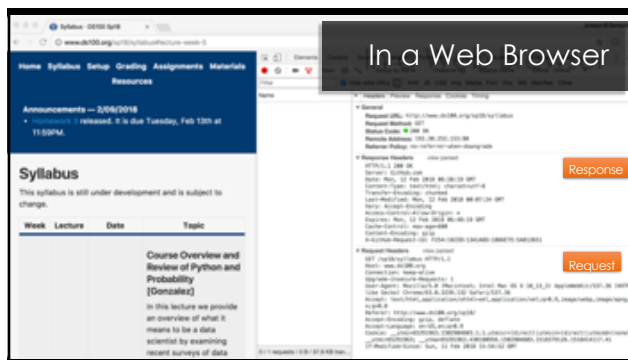
Optional Body

- send extra parameters & data

## Request – Response Protocol



- First line contains status code
- Key-Value Pair Lines
- Data properties
- Body
- Returned data
- HTML/JSON/Bytes



## Request Types (Main Types)

- **GET – get information**
  - Parameters passed in URI (limited to ~2000 characters)
    - `/app/user_info.json?username=mejoeyg&version=now`
    - Request body is typically ignored
  - Should not have side-effects (e.g., update user info)
  - Can be cached in on server, network, or in browser (bookmarks)
  - Related requests: HEAD, OPTIONS
- **POST – send information**
  - Parameters passed in URI and BODY
  - May and typically will have side-effects
  - Often used with web forms.
  - Related requests: PUT, DELETE

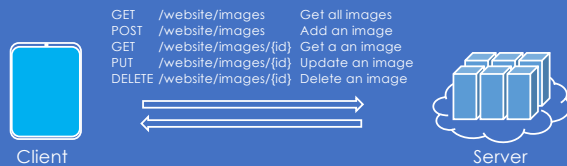
## Response Status Codes

- **100s Informational** – Communication continuing, more input expected from client or server
- **200 Success** – e.g., 200 – general success;
- **300s Redirection or Conditional Action** – requested URL is located somewhere else.
- **400s Client Error**
  - 404 indicates the document was not found
  - 403 indicates that the server understood the request but refuses to authorize it
- **500s Internal Server Error or Broken Request** – error on the server side

## Demo

basic\_http\_requests.ipynb

## REST APIs



## REST – Representation State Transfer

- A way of architecting widely accessible, efficient, and extensible web services
- Typically implemented on top of HTTP
- All client session state is maintained by the client:
  - request 1: GET /data/pages → responses first 33 pages
  - request 2: GET /data/pages?afterPage=33 → more pages ...
- REST APIs should be programmatically discoverable
  - In the example of request 1 above → response should indicate how to get the next batch of pages.

## REST Constraints

- **Client-Server:** both client and server should be able to evolve independently
- **Stateless:** The server does not store any of the clients session state → client passes state to server in each call
- **Cacheable:** system should clearly define what functionality can be cached (e.g., GET vs POST requests)
- **Uniform Interface:** provide a consistent interface for getting and updating data in a system
  - Accomplished through common resource identifiers (URIs)
  - Responses contain information used describe next operations

## Demo

TwitterAPI\_REST\_Example.ipynb



## Scraping Ethics

- Issues:
  - Violate terms of use for the service or data
  - Can cause substantial additional load on service
    - Many services are optimized for human user access patterns
    - Requests can be parallelized/distributed to saturate server
    - Each query may result in many database requests
- How to scrape ethically
  - Used documented REST APIs – read terms of service
  - Examine at robots.txt (e.g., <https://en.wikipedia.org/robots.txt>)
  - Throttle request rates (sleep)
- Avoid getting Berkeley (or your employer) blocked from Websites & Services