Your Cover Here

atlas book skeleton

Author Name

Table of Contents

Preface Title

This Is an A-Head

Congratulations on starting your new project! We’ve added some skeleton files for you, to help you get started, but you can delete any or all of them, as you like. In the file called chapter.html, we’ve added some placeholder content showing you how to markup and use some basic book elements, like notes, sidebars, and figures.

Tools For Social Justice in Python 1

By *Eric Schles*

Organization of the text

Introduction

- The intention of the text/thank you
- Strategies for getting stuff done
- prerequisite knowledge (and resource to learn said prerequisite knowledge)
- outline of text/overview of text

Chapter 1

working with data for non-profits and government

- web scraping
- PDF to csv
- data cleaning techniques
    - named entity resolution with machine learning
- optical character recognition - hard

Chapter 2

Analyzing data for government and non-profits

- finding optimal solutions to organizational issues:
    - choosing the optimal number of beds for a homeless shelter - hard

- Raising funds, finding bad guys, with targeted advertising:
    - raising funds - predictive marketing
    - finding bad guys - understanding your population with descriptive statistics
- doing social network analysis to catch bad guys and make connections
    - graph based algorithms - easy?
- Looking for money laundering in financial data - easy?
- analyzing audio records for red flags with time series analysis - easy?

Chapter 3

Visualizing data for analysts and to find patterns and creating interfaces

- web dev in flask, bootstrap, and angular.js - easy
- gis systems for free - easy
- c3,d3, and vincent - easy
- time series analysis in action - easy

Chapter 4

Searching across massive data sets

- facial recognition and facial comparison - building a cbir - easy
- searching across text in mass - easy
- multithreading applications for faster processing - easy
- using Hadoop and spark for search - easy

Chapter 5

Build the system I pitched to OAG - easy Finding patterns with disparate data

- bringing together closed and open data to build systems

Three example systems in this chapter Investa_gator/alert system
twilio system

- setting up a hotline
- diaster recovery/mass text message alerts

bring together everything in the book to build one giant automated attack system

Introduction

The intention of this text/thank you

The intention of this book is to explain the interested reader how to use computer science to solve social problems. To build systems and change minds and hearts. Often times those interested in social justice are not technologists and so they are unaware of what technology can do to make their work easier and more effective.

This book is divided into five chapters, each one touching on a specific theme. Each theme is intended to provide background and code on how to do one type of task. These tasks were culled from the many projects I have participated in, in the social justice/non-profit/government space. The intention of this guide is to teach you how to recreate the tools I have made but more so, to teach you, the developer, how to recreate the mindset I used to make these tools. I selfishly want to help the world, to make it a place that is more fair and balanced. A place where everyone is guaranteed freedom from opression or violence. A place where love wins over fear, not in every instance, but on average and increasingly so. In a way this means building systems where such guarantees are possible. Except the cogs of the system are people and social organizations, not machines. The intention of this book is to teach you how to construct the machinery of the morality. And to inject justice, freedom, voice to those most in need and in suffering.

It is no simple task that we embark on, but that also doesn't mean it has to be hard. Together, with your intention, and my knowledge, I believe we can make a difference together. I only hope that the lessons I have learned can be helpful in your work, whatever it may be. In computer science we spend so much time perfecting the ability to solve difficult, previously in tractable problems. We seek elegance of form in our solution. We try to build clean obvious solutions. Selfishly it is my hope that we can apply such systematic thinking, where possible, to the social problems of the world, and in doing so, form a better tomorrow for us all.

This text would not be possible without my mentors, friends, family, and collaborators. So to my parents Marc and Andrea, I say thank you for embuing me with a social compass, and for believing a life of service is the most fulfilling one. To my friends, I say thank you, you are far to numerous to name, but you know who you are. If not for you, I would have never made it through school or life, you are oxygen, and I hope you all know that. To my mentors, Ethan Mann, James Javor, Ellen Eurbanek, the late Samuel Schack, John Temple, and Andrew Raiseji you have my deepest thanks. If not for your guidance, and support I would have never found the direction that has led me to this point. It is my only hope that I can be half the mentor that each of you has been to me. And finally to my collaborators, I say thank you for your partnership and fraternity in this fight for a better tomorrow. I will name each of them throughout the text and so their names and organizations are merely listed here.

- Freddie Andrade wikitongues
- Daniel wikitongues
- Miguel Code in mexico - fix this (get spanish name)
- Marianne Belloti Exversion
- John Temple Manhattan District Attorney's Office Human Trafficking Response Unit
- Genna ECPATUSA
- Janai Smith ECPATUSA
- Geofrey Digital Ocean
- Phil Self Employed
- Lior Shahverdi Self Employed
- Alan NYU
- Tim Savage NYU
- Steve EPA
- Rob Spectre - Twilio

Strategies for getting stuff done

ToDo - write up the strategies for getting around obstacles in local government and add this to the introduction.

Prerequisite Knowledge (and resource to learn said prerequisite knowledge)

The intention of this text is to cover a lot of ground quickly. However, I want this text to be accessible to those who don't know everything. Here are some of my favorite guides to get all the prerequisites out of the way.
list of prerequisites:
For Beginners:

- **Codecademy**
- **Khan Academy Python**

Transition from Beginner to Intermediate

- **Learn to Code the Hard Way**
- **Python Open Book Project**

Intermediate

- **Data Structures and Algorithms in Python**

Intermediate and Advanced Topics:
More Python Resources:

- **General Set of Guides for Python**

Data Science Resources:

- **Set of Guides for Learning Data Science (Mostly Free)**
- **A list of Tools for Data Science**
- **A talk on named entity recognition** Web Dev:
- **A list of web dev tools and learning resources**
- **A curated list of Web Dev Guides**

Chapter Specific Prerequisites
Chapter 1:
ToDo
Chapter 2:
ToDo
Chapter 3:
ToDo
Chapter 4:
ToDo
Chapter 5:
ToDo
References:

- **Thomas Levine's personal Blog**
- **Aaron Schumacher's personal Blog**
- **Data Science 101**
- **R-Bloggers**
- **Hardcore currated list**
- **James Powell's Blog**

A quick note on installation

A lot of the tools we'll be using throughout this book will rely on pip, so we'll download and install that now. From time to time we'll need python packages that don't have working pip repos (at the time of this writing) and so alternatives will be presented for those libraries.

To get pip head over to **get-pip.py** simply save the file as get-pip.py. The run sudo python get-pip.py (on ubuntu or mac) or run python get-pip.py with a administrator rights (on windows).

Github for this book

All the source code for this text can be found on the books associated github site. Here you'll find a list of what code belongs to what sections:

Chapter1:

Working with GETs and POSTs - `making_requests.py && test_server.py`

Tools For Social Justice in Python 2

By *Eric Schles*

Organization of the text

Introduction

- The intention of the text/thank you
- Strategies for getting stuff done
- prerequisite knowledge (and resource to learn said prerequisite knowledge)
- outline of text/overview of text

Chapter 1

working with data for non-profits and government

- web scraping
- PDF to csv
- data cleaning techniques
    - named entity resolution with machine learning
- optical character recognition - hard

Chapter 2

Analyzing data for government and non-profits

- finding optimal solutions to organizational issues:
    - choosing the optimal number of beds for a homeless shelter - hard

- Raising funds, finding bad guys, with targeted advertising:
    - raising funds - predictive marketing
    - finding bad guys - understanding your population with descriptive statistics
- doing social network analysis to catch bad guys and make connections
    - graph based algorithms - easy?
- Looking for money laundering in financial data - easy?
- analyzing audio records for red flags with time series analysis - easy?

Chapter 3

Visualizing data for analysts and to find patterns and creating interfaces

- web dev in flask, bootstrap, and angular.js - easy
- gis systems for free - easy
- c3,d3, and vincent - easy
- time series analysis in action - easy

Chapter 4

Searching across massive data sets

- facial recognition and facial comparison - building a cbir - easy
- searching across text in mass - easy
- multithreading applications for faster processing - easy
- using Hadoop and spark for search - easy

Chapter 5

Build the system I pitched to OAG - easy Finding patterns with disparate data

- bringing together closed and open data to build systems

Three example systems in this chapter Investa_gator/alert system
twilio system

- setting up a hotline
- diaster recovery/mass text message alerts

bring together everything in the book to build one giant automated attack system

Introduction

The intention of this text/thank you

The intention of this book is to explain the interested reader how to use computer science to solve social problems. To build systems and change minds and hearts. Often times those interested in social justice are not technologists and so they are unaware of what technology can do to make their work easier and more effective.

This book is divided into five chapters, each one touching on a specific theme. Each theme is intended to provide background and code on how to do one type of task. These tasks were culled from the many projects I have participated in, in the social justice/non-profit/government space. The intention of this guide is to teach you how to recreate the tools I have made but more so, to teach you, the developer, how to recreate the mindset I used to make these tools. I selfishly want to help the world, to make it a place that is more fair and balanced. A place where everyone is guaranteed freedom from opression or violence. A place where love wins over fear, not in every instance, but on average and increasingly so. In a way this means building systems where such guarantees are possible. Except the cogs of the system are people and social organizations, not machines. The intention of this book is to teach you how to construct the machinery of the morality. And to inject justice, freedom, voice to those most in need and in suffering.

It is no simple task that we embark on, but that also doesn't mean it has to be hard. Together, with your intention, and my knowledge, I believe we can make a difference together. I only hope that the lessons I have learned can be helpful in your work, whatever it may be. In computer science we spend so much time perfecting the ability to solve difficult, previously in tractable problems. We seek elegance of form in our solution. We try to build clean obvious solutions. Selfishly it is my hope that we can apply such systematic thinking, where possible, to the social problems of the world, and in doing so, form a better tomorrow for us all.

This text would not be possible without my mentors, friends, family, and collaborators. So to my parents Marc and Andrea, I say thank you for embuing me with a social compass, and for believing a life of service is the most fulfilling one. To my friends, I say thank you, you are far to numerous to name, but you know who you are. If not for you, I would have never made it through school or life, you are oxygen, and I hope you all know that. To my mentors, Ethan Mann, James Javor, Ellen Eurbanek, the late Samuel Schack, John Temple, and Andrew Raiseji you have my deepest thanks. If not for your guidance, and support I would have never found the direction that has led me to this point. It is my only hope that I can be half the mentor that each of you has been to me. And

finally to my collaborators, I say thank you for your partnership and fraternity in this fight for a better tomorrow. I will name each of them throughout the text and so their names and organizations are merely listed here.

- Freddie Andrade wikitongues
- Daniel wikitongues
- Miguel Code in mexico - fix this (get spanish name)
- Marianne Belloti Exversion
- John Temple Manhattan District Attorney's Office Human Trafficking Response Unit
- Genna ECPATUSA
- Janai Smith ECPATUSA
- Geofrey Digital Ocean
- Phil Self Employed
- Lior Shahverdi Self Employed
- Alan NYU
- Tim Savage NYU
- Steve EPA
- Rob Spectre - Twilio
- Jon Gottfried - Major League Hacking
- Ben Singleton - NYPD
- Bryan Britten - Manhattan DAs office
- Tara Byrne - ???
- Noelle - heatseeknyc
- Ziba Cramner - Demand Abolition
- Dhakir - Demand Abolition
- Delaney - Demand Abolition
- Andreas Mueller - NYU
- Darius - ???

And now I need to devote a paragraph to this person. 5 years ago I was just a graduate student hoping to work on human trafficking. I got to do a small project with the department of homeland security and I thought that was going to be it. However thanks to you, Brandon Diamond, I was able to share that work on the stage of New York Tech Meetup, just a month after I finished it. Thanks to you I was able to become a part of the social justice community and face the fear of speaking to thousands of people, judging you. You've always been a good friend, but when you recommended me to speak in front of that community, you changed my ability to have an impact on the world. That's not something even most good friends can do. I owe you more than these words. I owe you my undying gratitude and hope we will be friends for a long time to come. I love you, man. Thank you for helping me do what I've always selfishly wanted to do. For bringing my work forward. For making me better, smarter, stronger and more capable. I learn something new from you every time we speak, not something you can usually say about someone you've known since before high school.

And finally to James Powell, without your help I'd never be able to speak to as many audiances as I have. You have given me limitless speaking opportunities and chances to share my story and the story of the voiceless - those who are trafficked. Thank you for everything you've done for me.

Strategies for getting stuff done

ToDo - write up the strategies for getting around obstacles in local government and add this to the introduction.

Prerequisite Knowledge (and resource to learn said prerequisite knowledge)

The intention of this text is to cover a lot of ground quickly. However, I want this text to be accessible to those who don't know everything. Here are some of my favorite guides to get all the prerequisites out of the way.

list of prerequisites:
For Beginners:

- **Codecademy**
- **Khan Academy Python**

Transition from Beginner to Intermediate

- **Learn to Code the Hard Way**
- **Python Open Book Project**

Intermediate

- **Data Structures and Algorithms in Python**

Intermediate and Advanced Topics:
More Python Resources:

- **General Set of Guides for Python**

Data Science Resources:

- **Set of Guides for Learning Data Science (Mostly Free)**
- **A list of Tools for Data Science**
- **A talk on named entity recognition** Web Dev:

- **A list of web dev tools and learning resources**
- **A curated list of Web Dev Guides**

Chapter Specific Prerequisites
Chapter 1:
ToDo
Chapter 2:
ToDo
Chapter 3:
ToDo
Chapter 4:
ToDo
Chapter 5:
ToDo
References:

- **Thomas Levine's personal Blog**
- **Aaron Schumacher's personal Blog**
- **Data Science 101**
- **R-Bloggers**
- **Hardcore currated list**
- **James Powell's Blog**

A quick note on installation

A lot of the tools we'll be using throughout this book will rely on pip, so we'll download and install that now. From time to time we'll need python packages that don't have working pip repos (at the time of this writing) and so alternatives will be presented for those libraries.

To get pip head over to **get-pip.py** simply save the file as get-pip.py. The run sudo python get-pip.py (on ubuntu or mac) or run python get-pip.py with a administrator rights (on windows).

Github for this book

All the source code for this text can be found on the books associated github site. Here you'll find a list of what code belongs to what sections:

Chapter1:

Working with GETs and POSTs - `making_requests.py && test_server.py`

Chapter 1 - Working With Data For Non-Profits And Government

3

The biggest problem facing government agencies and non-profits today is data. These problems stem from cleaning, storing, maintaining and sometimes even accessing this data. Lots of entities have been slow to the technology revolution and often they don't have basic functionality like search. Much of the data government deals with is either trapped in emails, pdfs, word documents, or pictures. This means these documents are not searchable and aren't easily capable of becoming searchable. Of course, many, many solutions exist to this problem, but often times getting such a solution installed is a herculean task, taking many many years. Therefore writing your own solution is usually going to be faster. This type of task addresses our first strategy to being effective in government, hack around obstacles.

For whatever reason, I've found that government administrators are much more comfortable with having python on their machines than a given open source technology. My guess is this is a mix of fear and misunderstanding of open source technology in general, but I can't be sure exactly why. In any event, I've found myself waiting months or even sometimes being told a flat out 'No!' to installing specific open source projects. And therefore, I've had to come up with my own solutions to many well solved problems. This certainly isn't the case for every government institution or non-profit, but from what I understand talking to other technical colleagues across local government, it seems to be more the rule than the exception.

The intention of this chapter is to give you methods of hacking around such artifical obstacles as access to data that is already publicly available, search across data, poorly formed data, and data trapped in images. We'll address these obstacles with the following tools:

- Web Scraping
- Turning PDFs into CSVs

- Data Cleaning Techniques
    - Named Entity Resolution with Machine Learning
- Optical Character Recognition

Web Scraping

Web scraping is the ability to programmatically store and process data from API's, websites, RSS feeds, and other content that exists on the internet. The steps usually involve two things - download the relevant content locally and then parse that content, extracting the relevant information.

There are many great blog posts, tutorials, and tools for doing web scraping. Here we'll touch on my three of my favorite:

- **Python's requests Library**
- **Python's lxml library**
- **Python's Selenium library** & **Phantom.js**

Installation

Mac OSX or Ubuntu

sudo pip install -U requests lxml selenium

Windows

with administrator rights:
pip install -U requests lxml selenium

Getting started with requests

The requests library is extremely powerful and very high level. It can be used for a whole bunch of things. Before we talk about how we use it, let's talk about some reasons why we'd use it. The common reasons I've seen for using requests is there is some data served somewhere on the internet that people at your organization care about. For my work fighting slavery, we've often wanted to store the content of a website we think is encouraging or creating human trafficking. We want to do this for a few reasons, but the most important one is to gather complete and robust evidence because often times this data can be deleted, lost or destroyed. If that happens then a vital piece of an investigation may be lost. Another common reason I've seen is for collecting statistics that another website posts, but that isn't publicly available in another format easily. The other big reason only happens in large government agencies. This is the case when a certain process publishes regular reports from a database, but you can't find anyone who has access to the database, but you need the data.

There's one common theme here - there is data that is publicly or semi-publicly available that you'd like to be able to manipulate or store locally, and this data is updated overtime.

The requests library allows you to do some very important things, we'll only talk about two of the methods in the library - GET and POST. It is assumed that you already know object oriented programming, so if you don't please go check out the resources I listed above so that you know how classes, objects, and data structures work.

Working With GETs And POSTs

Note: For this section of the text please refer to `making_requests.py` and `test_server.py` in chapter1 in the github repo.

A GET request is one that asks the server for some data. A POST request is one that sends some data to the server. They are easily to do in the requests library and have specific uses.

The canonical GET request is something like visiting a web page - the html that is rendered by your browser was by a GET request (more often than not).

Here's how that method might look:

```
import requests
r = requests.get("https://www.google.com")
```

This request returns an object - which we've named r, short for response. This is a classical pattern called request, response and you'll find it in other places, like javascript MEAN stack applications. There are a number of properties that the response object returns. The most important ones are:

- text - stores the html of the page as a string
- url - stores the url of the request
- content - stores the content of the request, in case we are dealing with something other than html like a pdf
- raw - the same thing as content more or less, but there is a difference in practice, although I'm not sure what exactly. Best to try content and if you don't get back everything you wanted, try raw.
- headers - the headers of the request.

You'd access this information in the following way:

```
import requests
r = requests.get("https://www.google.com")
print r.text
print r.url
print r.content
print r.raw
print r.headers
```

A POST request is one that sends data to the server. Since I don't want to have people sending the same fake data to some website, I wrote my own test server for this example and the remainder of the section.

So you'll need to get the github repo - `https://github.com/EricSchles/tools_for_social_justice` which you can get by typing:

`git clone https://github.com/EricSchles/tools_for_social_justice.git`

then navigate over to `tools_for_social_justice/code/chapter1/`. From there you'll be able to run the server with `python test_server.py` and all the examples with `make_requests.py`.

So go ahead and run the server and then the following code will work:

```
payload = {"username":"eric","pass":"1234"}
r = requests.post("http://localhost:5000/form",data=payload)
```

This request will post the following data - `"eric"` and `"1234"` to the form, which is pretty neat! Often times you'll need to log into certain government websites, before you can access their data and this will allow you to do just that. Other uses include automatically filling out forms for surveys - say you're unit needs some piece of software and the agency happens to be sending around a suggestion form, you can write the data so that it's clear that you really want this one specific thing and then pass it around to all your coworkers to run this script. Just make sure you get your bosses approval first! But your coworkers will love you for this! I've spent way too much time filling out internal surveys just to drop the hint that our unit needs one specific thing.

I do want to note, that not all websites will allow you to pass credentials and then scrape their content, sometimes a web page is generated dynamically, sometimes a website will pass back a cookie, tracking your state while accessing the content of the website. We'll address each of these cases as well, but this is the simplest form of submitting data to a website.

An important thing to note is, the data here - called payload is sent after the html page is loaded, by the server. And so it makes sense that a dictionary is passed - the keys correspond to the names of the fields in the html page, the values, the actual data being passed to the form.

When making use of an api that requires credentials, we'll need to pass a different keyword - auth (instead of data). This will send the credentials along, before the html page is loaded.

```
data = ('admin','1234')
r = requests.get("http://localhost:5000/secret-page",auth=data)
```

Some important differences to note - 1st we pass a tuple - the credentials listed in order. 2nd we use the `auth` keyword to pass the data. 3rd we make use of a get request, because we want to get some data from the server, rather than sending some form data to the server.

The next example is that of sending files via a form to the server. This can be extremely powerful, if you know you have some internal web service and 500 files that need to be processed. Perhaps classically this service only processed one file at a time. Rather than having to track down the maintainer of this service, if they are even still around, it's far easier to have python just send all the files across.

This simple example illustrates how to send a file via post request.

```
files = {"file": open("report.csv","r")}
r = requests.post("http://localhost:5000/get_csv",files=files)
```

Notice that this is a dictionary, where the key corresponds to the name of the input form and the value corresponds to the file in question. For pdfs and microsoft documents you can use `open([filename],"rb")`. This stands for read binary.

Parsing HTML

Now that we know how to get and send data programmatically, we are ready to start being selective about what we download and keep or write a web crawler. Web crawlers are extremely powerful for mapping every page of a website and infact, I've done just this to forensically download and store websites over time, to be used as evidence of human trafficking. Other uses exist in the investigative world, like navigating the deep web.

```
import requests
import lxml.html
from unidecode import unidecode
from sys import argv

website = argv[1]
r = requests.get(website)
```

```
html = lxml.html.fromstring(unidecode(r.text))
print html.xpath("//a/@href")
```

In this simple example we see the power of web parsing, in its simplest form - grabbing all the links from a web page. Once we have a web page in memory, we transform the ascii characters into something python can understand, via unidecode, and then loading the html into the lxml tree data structure. Once we've parsed all the tags via lxml's fromstring method, we are free to make use of xpath - a micro language that will allow you to quickly and efficiently traverse a tree to query for certain peices of information, in this case all the hyper links in an html page.

While xpath as a language is far too deep in terms of length to go into in detail, we'll cover a few features here:

`//` - searches the whole tree for the following tag, from the root of the tree `/` - search from the relative node that is referenced so `//a/p` will search after all the a tags, and `/a` searches only the top level of nodes `/a/@href` - searches for a tags with the href attribute `/a/p[@id="hello"]` - searches for all a tags with `id="hello"`

If the xpath language isn't familiar to you I'd recommend the following guides:

- **w3schools**
- **mozilla guide**

Crawling the web

One of the great things of python is how readable and compact you can make the code. The following piece of code is a fully capable web crawler (albeit without any bells or whistles). A slightly more robust version of this (complete with a database) lives on the github repo for the book **here**

```
import requests #sudo pip install requests
import lxml.html #sudo pip install lxml.html
from unidecode import unidecode #sudo pip install unidecode

def links_grab(url):
    r = requests.get(url)
    html = lxml.html.fromstring(unidecode(r.text))
    return html.xpath("//a/@href") + [url] #ensures the url is stored in the final

def crawl(base_url,start_depth=6):
    return crawler([base_url],base_url,start_depth)

def crawler(urls, base_url, depth):
    urls = list(set(urls))
    domain_name = base_url.split("//")[1].split("/")[0]
    url_list = []
    for url in urls:
        if domain_name in url:
            url_list += links_grab(url)
    url_list = list(set(url_list)) #dedup list
    url_list = [uri for uri in url_list if uri.startswith("http")]
    if depth > 1:
        url_list += crawler(url_list, base_url, depth-1)
    urls += url_list
    urls = list(set(urls))
    num_urls = len(urls)
    return url_list
```

So let's break down the code. The `links_grab` function should look familiar, since we made use of it in the last example. Notice the small difference - the return statement is adding two lists together, which is why [url] is stored as a list.

Next we'll look at the crawler function.

The first line - `urls = list(set(urls))` deduplicates the urls as they are passed in, this ensures no url will be scraped twice, which would be redundant.

The next important line is - `if domain_name in url:` - this ensures that we are only grabbing content from one domain at a time. I include this because otherwise our list of urls will grow exponetially fast and by the fact that we aren't trying to necessarily re-implement the google crawler. Typically in investigative work, we only care about grabbing all the html for a single domain, rather than multiple. Of course, there will be some circumstances when you want all the domains linked to a website, if that is the case, simply remove this line. You can of course add further customization - simply by ignoring certain domain names. Below is a modified crawler method that ignores popular websites you are unlikely to care about when trying to store neifarious websites:

```
def crawler(urls, base_url, depth):
    domains = ["www.google.com","www.yahoo.com","www.linkedin.com","www.github.com"]
    urls = list(set(urls))
    url_list = []
    for url in urls:
        if not any([domain in url for domain in domains]):
            url_list += links_grab(url)
    url_list = list(set(url_list)) #dedup list
    url_list = [uri for uri in url_list if uri.startswith("http")]
    if depth > 1:
        url_list += crawler(url_list, base_url, depth-1)
    urls += url_list
    urls = list(set(urls))
    num_urls = len(urls)
    return url_list
```

Here we simply create a new list of domains and if any of them are found, we don't scrape those urls. The assumption is that you don't start from one of the urls on the no-scrape list.

Once we've scraped all our links we dedup again with `url_list = list(set(url_list)) #dedup list` and then make sure all our urls are reachable via their uri `url_list = [uri for uri in url_list if uri.startswith("http")]`. Many href's refer to relative paths for websites. Since this is just an introductory example, I don't include the code to resolve this here, however in the github repo, I have all the necessary added steps to resolve relative urls. I have to warn you however, this makes things very slow. You'd be surprised how many urls you can get from a single scrape.

Notice the next lines `if depth > 1:` and `url_list += crawler(url_list,base_url,depth-1)`, this is the recursive step and sort of the heart of the engine. Since we are moving through the urls recursively, scraping new links from the current step, our list of urls will get big fast. We can think of the structure of a single scraping like a b-tree, where the number of edges at a given level corresponds with the number of links on a page, and each node being the actual html page, storing said edges. This structure shows just how exponentially large our b-tree grows, often very big very fast. It would be an interesting study to look at how the number of links grow from page to page on average for different types of websites, allowing us to understand an average bounding on the time complexity of a type of scrape. But enough about the how, let's talk about the why.

Often times we need to scrape a set of pages for a few reasons:

1) To get information trapped from some database that you are supposed to have access to, but don't, mostly because the person who runs said database is being difficult and doesn't want to give you a direct feed. However the information is of course, published publicly in a far less machine readable format.

This is an example of hacking around obstacles, something you'll need to do often if you decide to work for the government or really any non-profit. I'm not sure why folks are stubbornly unhelpful in these roles, but usually they are.

2) To grab information for an investigation. Often times on the sex trafficking world traffickers will post to a website like backpage.com but have their own websites that you can link to. The content on these smaller websites will often change, sometimes very regularly, and sometimes not at all. It is important to an investigation to know all the people that are featured on a given website of this kind, and so being able to readily store such information is of paramount importance.

3) To do market research. An unfortunate part of the non-profit space is grant writing. This takes up a lot of time for many NGOs and non-profits, so knowing what everyone else in the space is saying with real-time accuracy is often a big deal. Being able to scrape the other folks in the space becomes an important part of the grant writing process - knowing not only what they are saying, but what they are not saying.

4) Scraping other government entities. If you thought your IT department was bad, wait till you meet the IT folks in other government agencies, these are typically the least helpful people possible. If your IT staffer decides he/she doesn't like you and doesn't want to help, at least you can escalate things to superiors so that they have to do it, if other people in other IT departments don't like you, there is no path forward, ever. But often when working on a problem with a social justice theme, you'll need the information that other government agencies are using. Thanks to open data standards that are being forced on many public agencies, for the first time, you'll have the ability to actually circumvent this difficult people, so that you can actually do your job! Often times, you'll be able to make the scraping happen, so you can get your work done.

Turning PDFs into CSVs

This next section really addresses this last point from the previous section - working with government data from other entities. As I've already said, many government institutions will have to release a bunch of their data to the public, in an attempt to create a more open and transparent government. Of course, this has lead to a ton of problems. Partially because many government instutions aren't really equiped to do that, from a technical stand point. And so, often what you get, is folks just publish static pdf documents to the web and think they are being open and transparent. One of the most important things open governance needs is the ability to make this data machine readable. However, this is likely a far off dream, so we are going to hack around it.

The uses of machine readable data across government are limitless, but here are a few my favorite ideas -

1. Creating citizen based analysis to show however government could be more effective:
    - Maps showing you were all the safest parks are based on crime data
    - High level financial analysis showing where and government dollars are actually spent, as well as how they could be spent better

1. Creating citizen based feedback loops showing government how they could better govern:

  ◦ Active voting from quarter to quarter, based on the performance of a specific agency, which partially determines funding level

Both of these could be accomplished via an API, number (1) from an api-GET request and number (2) from an api-POST request. Thats it.

In lew of such a fully functional API and having to work with PDFs isn't the end of the world. In this section we'll cover how to parse PDFs and give you stragies for getting the information you want.

The first strategy involves finding static text that doesn't change from document to document, this is typically useful for quarterly reports that are generated from a database. Because parts of the text don't change, you can use these as relative offsets to only process the parts of the document you care about. I'll refer to these pieces of unchanging text as invariants or text invariants.

Once we have the invariants figured out, we'll need to understand the structure of the data or text we want. This is typically a table, or a chart. Ofcourse, from time to time we also want some text, but this method generalizes nicely to that case as well.

Installation

Pandas - `sudo pip install pandas` or `sudo apt-get install python-pandas` #ubuntu only Poppler-utils:

- On ubuntu - **http://poppler.freedesktop.org/**
- On Windows - **http://blog.alivate.com.au/poppler-windows/**

Note for windows you'll need to set the environment variables or pdftotext. For information on how to do this please see **this guide**

The following piece of code can be found in the chapter1 section of the github:

```
from subprocess import call
from sys import argv
import pandas as pd
def transform(filename):
    call(["pdftotext","-layout",filename])
    return filename.split(".")[0] + ".txt"

def segment(contents):
    relevant = []
    start = False
    for line in contents:
        if "PROSECUTIONS CONVICTIONS" in line:
            start = True
        if "The above statistics are estimates only, given the lack" in line:
            start = False
        if start:
            relevant.append(line)
    return relevant

def parse(relevant):
    tmp = {}
    df = pd.DataFrame()
    for line in relevant:
        split_up = line.split(" ")
        # a row - 2008 5,212 (312) 2,983 (104) 30,961 26
        split_up = [elem for elem in split_up if elem != '']

        if len(split_up) == 7:
            tmp["year"] = split_up[0]
            tmp["prosecutions"] =split_up[1]
            tmp["convictions"] = split_up[3]
            tmp["victims identified"] = split_up[5]
            tmp["new or ammended legistaltion"] = split_up[6]
            #print tmp
            df = df.append(tmp,ignore_index=True)
    return df

if __name__ == '__main__':
    txt_file = transform(argv[1])
    text = open(txt_file,"r").read().decode("ascii","ignore")
    contents = text.split("\n")
    relevant = segment(contents)
    df = parse(relevant)
    df.to_csv("results.csv")
```

This code parses the pdf - trafficking_report.pdf which can also be found in the github. This is a very long report, but what we care about is one table in particular (by way of example). This table can be found on page 45 of the report and it details arrest details about human traffickers internationally.

Let’s walk through each of the methods, which are defined in the order they are called.

```
def transform(filename):
    call(["pdftotext","-layout",filename])
    return filename.split(".")[0] + ".txt"
```

The transform method calls pdftotext, a command line utility that transforms a pdf into a .txt document. Notice the use of the layout flag which preserves the formatting as much as possible from our pdf file, this is crucial to ensuring our .txt document will be easily parsable.

```
def segment(contents):
    relevant = []
```

```
        start = False
        for line in contents:
            if "PROSECUTIONS CONVICTIONS" in line:
                start = True
            if "The above statistics are estimates only, given the lack" in line:
                start = False
            if start:
                relevant.append(line)
        return relevant
```

In the segmentation step we find the text invariants that start and end the section of the document we wish to parse. Here the start invariant is `"PROSECUTIONS CONVICTIONS"` and the end invariant is `"The above statistics are estimates only, given the lack"`. What's returned is a simple dictionary of the relevant lines of text from the transformed .txt document.

```
    def parse(relevant):
        tmp = {}
        df = pd.DataFrame()
        for line in relevant:
            split_up = line.split(" ")
            # a row - 2008 5,212 (312) 2,983 (104) 30,961 26
            split_up = [elem for elem in split_up if elem != '']

            if len(split_up) == 7:
                tmp["year"] = split_up[0]
                tmp["prosecutions"] =split_up[1]
                tmp["convictions"] = split_up[3]
                tmp["victims identified"] = split_up[5]
                tmp["new or ammended legistaltion"] = split_up[6]
                #print tmp
                df = df.append(tmp,ignore_index=True)
        return df
```

The final section is the parse method. Here we create a dictionary which will be appended to the pandas DataFrame that we'll make use of. The reason for storing things in a pandas dataframe is because of the ease of transformation to other persistent file stores such as CSV, EXCEL, a database connection, and others. Notice that inside the loop we split up the line by whitespace, since this is a table, we should expect tabular data to appear in the same position on different lines. Also notice that we expect the size of each split up line to be of length 7. Not that this does not capture a few of the lines in the original pdf, it is left as an exercise to handle these minor cases. Length is not the best metric for ensuring you are processing the correct information, but the intention of this code is to be as readible as possible, not necessarily as sophisticated as possible. Other ways you can check to ensure your scraping the correct text is with regex, checking for certain character invariants that should appear on every line, or by using advanced machine learning techniques which we will talk about in the next section. Finally, the tmp dictionary is assigned each of the values from the table and appended to the dataframe. Notice that we only need to create tmp dictionary once and then simply overwrite it's contents on each loop through the relevant content. Then we simply return the dataframe to main. Here a single line is used to send the dataframe to a csv:

```
df.to_csv("results.csv")
```

And we are done!

There are other, more elegant ways of parsing pdfs, like those found in **this chapter** of automate the boring stuff. Unfortunately, using a library like PyPDF2 won't work in all cases. So while my method is certainly not elegant, it is robust and will work 99% of the time.

Data Cleaning techniques

Now that we know how to handle PDFs under the best possibel scenario, when there are no Optical Character Recognition errors, its time to deal with slightly less than ideal situations, IE when there are mistakes in your data. In the typical large company setting data isn't entered the same way consistently, which is an annoying problem, however for that we have things like named entity recognition, which we'll get into. What I'm talking about is when your data is plain and wrong. Here's an example of what I mean:

A row is supposed to say:

For check number ending in 7519:

But it actually says:

F0e c$#ck namb ending in 7%@!:

How the heck are you supposed to handle this situation?! This is what actually happened to me while working for the manhattan DAs human trafficking response unit. We'd get a ton of documents from subpoena compliance, but we could only afford an absolutely terrible OCR solution and so the .txt documents hand BOAT loads of errors just like the above. And here's the truly frustrating thing, they were never consistent. The OCR solution would mess things up, often differently, for the same words. And so you'd never get consistent results. This is really bad for two important reasons:

1. Because it means my invariant scheme falls apart :((((<- the many chins of sadness rained down upon me the day I realized this.
2. Because it means your data will be inconsistent and incorrect in your CSV :(((((<- even more chins of sadness, I had to stress eat to cope with the level of depression and thus became even fatter.

So, how do we get around this? Unfortunately there is no one size fits all answer, but there are lots of things you can do:

Let’s say you know that there are certain words that will never appear in a line with something you do care about, but this data is being collected, I.E. you have extra data. Then you could do something like this:

```
def parse(relevant):
    tmp = {}
    df = pd.DataFrame()
    to_avoid = [
        "Hard to see",
        "weird",
        "Something else you want to avoid"
    ]

    for line in relevant:
        if any([elem in line for elem in to_avoid]):
            continue

        split_up = line.split(" ")
        # a row - 2008 5,212 (312) 2,983 (104) 30,961 26
        split_up = [elem for elem in split_up if elem != '']

        if len(split_up) == 7:
            tmp["year"] = split_up[0]
            tmp["prosecutions"] =split_up[1]
            tmp["convictions"] = split_up[3]
            tmp["victims identified"] = split_up[5]
            tmp["new or ammended legistaltion"] = split_up[6]
            #print tmp
            df = df.append(tmp,ignore_index=True)
    return df
```

With this you can keep a list of all the terms you want to avoid and simply not process any line that has this term you know will cause problems. So now we can handle the case when we have too much data, what happens when we miss some data?

For that we’ll need to account for all the variations of a given starting or closing phrase:

```
def segment(contents):
    relevant = []
    start = False
    starting = [
        "PROSECUTIONS CONVICTIONS",
        "PROSECUTIONS",
        "CONVICTIONS",
        "CONVICTION$",
        "C0NV1CT!ONS"
        ]
    ending = [
        "The above statistics are estimates only, given the lack",
        ", given the lack",
        "The above statistics are estimates",
        "statistics are estimates"
        ]

    for line in contents:
        if any([elem in line for elem in starting]):
            start = True
        if any([elem in line for elem in ending]):
            start = False
        if start:
            relevant.append(line)
    return relevant
```

Here we can see employing a similar trick. Every time we encounter a new type of OCR error, we handle it in specific. However, this likely doesn't generalize well, here we enter named entity recognition to generalize out this technique further, making it more flexible and effective.

Named Entity Recognition

So far we've covered a somewhat hacky way of converting things that should be the same to things that are treated the same, but we didn't actually transform anything. Named entity recognition is the practice of taking techniques from machine learning and other data processing techniques, and using them to convert text that looks different but has the same semantic meaning. The most common example of this is disambiguating addresses. There are hundreds of ways to write a single address, which you wouldn't think, but account for the fact that people misspell things and this becomes very plausible. And so you need a way to make sure all of the different ways you can write 1234 Fake st., New York, NY, 10013 are all the same thing. Of course, the first thing to do is decide on a canonical representation and then make sure everything that should be that thing, appears the same way.

Since named entity recognition is actually a very, very diverse set of techniques, we'll just touch on a few of each. You can see a more complete list of these techniques in **this great lecture by Benjamin Bengfort**.

Installation

- **Dedupe** - sudo pip install dedupe

- Potential dependency for Dedupe: sudo pip install zope.interface
- **FuzzyWuzzy** - sudo pip install fuzzywuzzy
- **Distance** - sudo pip install Distance
- **PyBloom** - sudo pip install pybloom
- **NLTK** - sudo pip install -U nltk; python -c "import nltk; nltk.download()"
- **scikit-learn** - too long for one liner here's **a guide**
- **Jellyfish** - sudo pip install jellyfish

Together the above packages represent throwing the preverbal kitchen sink at the problem. Yes, a lot of people have spent time and energy making sure that the preprocessing of data was easy. Sadly, this topic is still daunting for first timers. Mostly because there are so many tools and a lot of it comes down to a matter of taste. This is in part because there are so many ways to do data transformations and different people will be more or less comfortable with different techniques depending on their background. They'll all agree on some set of techniques, but the ways to get there are many.

It isn't essential that you pick a favorite, or that you use any of the above packages. In fact, all of the transformations you could want to do can with the techniques I already showed you in the last section. However, that doesn't mean this will be the most performant or accurate way to do things, and that's what these techniques are for. So you don't have to reinvent the wheel.

First we'll talk conceptually in this section, about the techniques you can use. And then we'll play with each of the libraries, showing very basic examples. I don't want to be exhaustive about how to do everything or walk you through all of the examples each of these libraries provides, because this isn't a book solely on machine learning, it's a book about getting stuff done. And so if you feel like you'd be better served simply knowing these libraries exist and getting your hands dirty, I'd encourage you to do so.

Also, it's worth noting that much of this section is lifted from the Benjamin Bengfort lecture listed above.

So let's get some definitions going:

- Deduplication - Cluster records that correspond to the same real world thing
- Canonicalization - Creating one representation for each real world entity
- Record Linkage - Match records from one record store to another (typically a CSV or a database table)
- Referencing - Match records to a look up table that has already been processed
- Normalization - make everything look the same (no capitalization, no extra spaces, etc.)

As you can see these are all pretty standard tasks. But really they are just ways about talking about one thing, how do we tell when something is the same? And what does being the same mean, really?

The simplest form of checking for similarity is via various distance metrics, in written language. So how do we define distance? It's clearly not topological, at least not in the strict mathematical sense. We can't do an L1 or L2 norm. We can think of numbers in two space or three space. Or can we? Even if we could, it probably wouldn't make much sense.

Instead, let's prepose that we look at things like the length of two words, the beginning of two words, the number of characters that are common to both words, how similar two words sound when pronounced and their colocation in reference to other words throughout a text. These will form the basis for our understanding of 'distance' in the land of words.

Edit Distance

First we'll look at edit distance - the number of characters you'd have to change in order to get from one word to another. If the edit distance is small, say one character, then someone probably had a typo, and the words are likely the same. Let's see a way we can check the edit distance in python:

```
import distance
print distance.levenshtein("Hello there","hello there")
```

The result of this computation is a long where a distance of 0, means the two words are the same. Any other distance will be a positive number. So if one character is different, the distance between the two strings will be 1, a difference of two characters two, and so on.

Edit distance and other factors

The next distance we'll look at is Jaro-Winkler. Like levenshtein's distance metric, Jaro-Winkler looks at the edit distance between two words, however it does things in a more sophisticated way. Rather than only considering the number of characters you'd need to change in order to get from one string to another, it considers the order of the characters that are the same, the number of spaces you'd need to move those characters so they'd be in the same order, and the number of characters that are different. For short words, Jaro-Winkler is ideal and will usually outperform levenshtein distance. However, it won't be best for all possible cases, so its important to be careful. For instance, in long words, with lots of typos Jaro-Winkler will likely not do very well. It should be noted that Jaro-Winkler is great for record linkage.

```
import jellyfish
print jellyfish.jaro_winkler(u'Hello there new friends',u'hello there new friend')
```

Notice a few things here - (1) we need to use unicode strings (mildly annoying), (2) jaro-winkler returns a number between 0 and 1 (a float).

Looking at stemming

Another way of checking if two words are semantically the same is by looking at substrings.

The following code is just an example of how one might go about doing this, a trie is likely a far more compact and interesting way to look at substrings. It is of my own design (as far as I know), and is extremely simple-minded:

```
def str_comp(str1,str2):
    score = 0
    words1 = str1.split(" ")
    words2 = str2.split(" ")
    if len(words1) < len(words2):
        for ind,word in enumerate(words1):
            score += word_comp(word,words2[ind])
    else:
        for ind,word in enumerate(words2):
            score += word_comp(word,words1[ind])
    return score

def word_comp(str1,str2):
    subword1 = [str1[:pos] for pos in xrange(1,len(str1)+1)]
    subword2 = [str2[:pos] for pos in xrange(1,len(str2)+1)]
    score = 0
    if len(subword1) < len(subword2):
        for ind,sub in enumerate(subword1):
            if sub == subword2[ind]:
                score += 1
        return score/float(len(subword1)*2)
    else:
        for ind,sub in enumerate(subword2):
            if sub == subword1[ind]:
                score += 1
        return score/float(2*len(subword2))
```

While this is clearly not the best possible way to do compare two words based on substring it gets at the point - the more the beginnings of the words are the same the higher the overall score, because if a word has the first five characters in common this will mean the score is greater than if only the first three characters are in common. However, this assumes that some of the first few characters are in common or the words can't possibly be the same or similar in meaning. This represents a problem for words that don't come from the same root word, but have similar meanings and therefore it is intended to only illustrate the idea, rather than being rigorous.

Looking at Colocation

In the previous sections we looked at the words themselves. What if two words refer to the same thing but are described with different nouns? Like perhaps if a person has a nick name. Looking at the most frequent words around a given word can help us that. For this analysis we'll make use of something called a n-gram. The best way to understand an N-gram, is to see it.

We will use the same sentence through out:

`Hi, my name is Eric.`

A 1-gram of that sentence would be:

`[('Hi,'), ('my'), ('name'), ('is'), ('Eric.')]`

A 2-gram of that sentence would be:

`[('Hi,', 'my'), ('my', 'name'), ('name', 'is'), ('is', 'Eric.')]`

A 3-gram of that sentence would be:

`[('Hi,', 'my', 'name'), ('my', 'name', 'is'), ('name', 'is', 'Eric.')]`

As you can see the number in front of gram determines how many elements each split we have. Also, we only increment the element by one in the sentence. This creates small phrases, which make up the elements of the n-gram.

Here's the code I used to generate the above sequences:

```
def ngram(sentence,n):
    input_list = [elem for elem in sentence.split(" ") if elem != '']
    return zip(*[input_list[i:] for i in xrange(n)])
```

The zip function will zip n many lists together into one list, where the elements of each list will become tuples. A small piece of syntactic sugar you may not be familiar with is the * in front of the list comprehension.

To understand the difference here let's look at an example:

```
#basic example
def thing(*x): print x
>>> thing([[elem] for elem in xrange(5)])
([[0], [1], [2], [3], [4]],)
>>> thing(*[[elem] for elem in xrange(5)])
([0], [1], [2], [3], [4])
```

```
#with zip
>>> zip([[elem] for elem in xrange(5)])
[([0],), ([1],), ([2],), ([3],), ([4],)]
>>> zip(*[[elem] for elem in xrange(5)])
[(0, 1, 2, 3, 4)]
```

As you can see, the * being passed into a function simply empties the elements of the list comprehension one by one, which is exactly what we want for our zip function.

Using our ngram function we can do the following:

```
def similarity_analysis(doc_one,doc_two):
    ngrams_one = [ngram(doc_one,elem) for elem in xrange(1,4)]
    ngrams_two = [ngram(doc_two,elem) for elem in xrange(1,4)]

    #longer body of text should be looped through
    if len(ngrams_one) < len(ngrams_two):
        ngrams_one,ngrams_two = ngrams_two, ngrams_one
    word_choice_count = 0
    phrase_choice_count = 0
    for elem in ngrams_one[0]:
        if elem in ngrams_two[0]:
            word_choice_count += 1
    word_choice_similarity = float(word_choice_count)/len(ngrams_one[0])

    phrases_one = ngrams_one[1] + ngrams_one[2]
    phrases_two = ngrams_two[1] + ngrams_two[2]
    for elem in phrases_one:
        if elem in phrases_two:
            phrase_choice_count += 1
    phrase_choice_similarity = float(phrase_choice_count)/len(phrases_one)
    return word_choice_similarity, phrase_choice_similarity
```

Rather than working at the word level checking for similarity, ngrams are good for checking similarity at the document level. Another document similarity algorithm is Term Frequency Inverse Document Frequency(TFIDF). Using TfIdf we can determine how important a given word is for a document. So how does this metric work?

Simply put the term frequency is the number of times the word appears in a document, divided by the total number of words in that document, and the inverse document frequency is the log of the number of documents in the corpus divided by the number of documents where the specific term appears.

Implementing this is obviously simple for a small amount of text, however as the amount of text grows, its important that our implementation be efficient. Fortunately sci-kit learn has implemented an efficient implementation of this metric:

```
from sklearn.feature_extraction.text import TfidfVectorizer

def TfIdf(document_list):
    vectorizer = TfidfVectorizer(stop_words="english")
    X = vectorizer.fit_transform(document_list)
    return vectorizer,X

corpus = [
    "Hello there, my name is Eric, will you by my friend?",
    "Hi there, I'm olaf, and I like warm hugs",
    "Oh hey there, my name is billop, and my mother didnt want to name me bill or phillip, so
    "Hello, my name is the rock, and can you smell what I'm cooking?  It's a pie, for your fa

vectorizer,result = TfIdf(corpus)
print dict(zip(vectorizer.get_feature_names(),vectorizer.idf_))
```

Using this metric we get a nice ranking of the important words across the corpus. We can use this to determine what are the most important words in the corpus. And when comparing two corpora, if the words that are ranked highest and lowest are similar, we can guess that the corpora might be the same. Or at least similar enough that we may believe they are written by the same person. While it is easy to see how distance metrics can be used to resolve individual words as referring to the same thing, it may be harder to understand the need for resolution at the document level. If large pieces of texts are very similar, we may believe that the documents are the same and that some words may have merely been mis-spelled.

So here is some code that could be used to do some more sophisticated entity resolution:

To Do

Here we make use of TfIdf, n-gram analysis, and a few distance metrics. As you can see, applying the two meta similarity scores and then looking for a cutoff value can save time, although it isn't perfect.

Chapter 2 - Modeling

Analyzing data for government and non-profits

- finding optimal solutions to organizational issues:
    - What is the optimal number of volunteers for an event or set of events
- Raising funds, finding bad guys, with targeted advertising:
    - raising funds - predictive marketing
    - finding bad guys - understanding your population with descriptive statistics

- doing social network analysis to catch bad guys and make connections
    - graph based algorithms - easy?
- Looking for money laundering in financial data - easy?
- analyzing audio records for red flags with time series analysis - easy?

Finding the optimum

Mathematics and statistics is one of the most under utilized tools within the government and non-profit world. This is because folks typically doing social justice avoided math like the plague in high school and college - which is fine. But math can be leveraged to make a world of difference. One of the largest organizational issues in the non-profit space is optimizing different resources. One of the ones no one likes to talk about is the number of volunteers. Have you ever volunteered with a non-profit, for a one off event? How much time did you actually spend doing stuff? I've been volunteering for many years to various organizations and there has been one common thread - there simply isn't enough work. Often volunteers are either under or over utilized (and usually not in a balanced way). Part of the reason for this is it's hard to do event planning (because most people don't show up) and it's even harder to discern how capable different individuals will be at different tasks.

Here we present a worked example, motivating finding the optimal solution:

Say we have a volunteering event for habitat for humanity and we want to build a house. Now assume we have a local college close by and we know they will be sending volunteers - we also have some contractors on hand to help build the home - how do we determine the optimal number of volunteers and the optimal number of contractors to have on hand?

Assume we have statistics on how productive student workers are on average and for simplicity - assume that students are around the same level of productive, thus even though there will be some variance in skill, none of the college students took masonry or built houses during their summers in high school. We should also assume most contractors are of a similar level of skill (definitely a simplying assumption). Despite all our assumptions we can generalize this example out to several sets of individuals with different levels of skill and still find an optimal solution (it just means we'll have more variables in our equations).

Say we have past data on habitat projects: (assume all homes have been normalized in some way in terms of size)

```
number of contractors, number of volunteers, number of hours to complete
7                     , 25                    , 12
8                    , 24                    , 9
14                    , 30                     , 8
```

```
15                    , 53                    , 14
13                    , 45                     , 13
15                    , 75                    , 17
```

We can think of this data as a function that takes in two integers and returns an float (I chose numbers for simplicity).

So how might this function look:

`f(number of contractors, number of volunteers) = number of hours to complete`

In code we might express this function as follows:

```
def calculuate_hours_to_complete(number_of_contractors, number_of_volunteers):
    return g(number_of_contractors,number_of_volunteers)
```

Note here that the actual mathematical function for this data is unknown - of course we could use statistics to approximate a functional form (which will learn how to do later in the chapter). For now we take it as given that a mathematical form for the function g exists, and we don't care what it is. Right now, we actually have enough information to dig into the data, so let's do that!

It looks there are a few things worth looking at here - the least number of hours to build the home was not done with the most number of contractors! Remember the house size was normalized so we can't chalk that up to size. This implies one of a few things:

1) There is an optimal number of volunteers

2) There is a difference in skill between the contractors and volunteers at each event

If you remember above - we simplified away the possiblity of 2 (more or less) leaving option (1)! So we can do some analysis here! So we notice that when the number of contractors is 8, and the number of volunteers is 30 it takes the least amount of time to complete a project. Thus we can claim the discrete form of this function takes on it's minimum value with those worker combinations. Of course, this may not be a complete data set, so its possible this isn't the absolute minimum number of overall workers you need, but it does tell us something - there are minimums - therefore this function whatever it's form is not strictly increasing! What this means for us is actually really awesome, we can do better without necessarily just throwing more workers at the problem - which means we can conserve resources. Maybe this means we could do twice as many projects, assuming we get more volunteers than we expected on a given day - maybe it means we need to spend less money on contractors!

So how did we do this wonderful analysis - we used a rate of change, we looked at how the number of hours changed when different input combinations were used and discerned an optimal point - this kind of analysis is common in calculus, applied to productivity and more generally in economics.

Next we'll look at a more simplistic example - with a larger amount of data. Here we'll assume only one type of volunteer (which certainly happens) and we'll use a generative functional form to figure out how productive these volunteers will be at building a house. From there we'll take the derivative and set it equal to zero to find the minimum of the function. By minimizing the functional form we can project out what the optimal number of volunteers will be for a project, even if we haven't seen that many volunteers before (assuming our volunteers are homogeneous). Of course, we could use heterogenous volunteers like the example above, but that would require multivariate optimization, which while fun, takes a bit too far afield from the point - you can use mathematics to do important work and making non-profits more effective.

So with out further ado, our code, the functional form, and the result. From there we'll look into how to generate an approximate functional form from our data and then take a derivative.

Before we do the derivative we'll need the ability to calculate the derivative. To do this we'll leverage the work of some wonderful people. To understand how it works check out: **hack the derivative @ pygotham** or you can just take my word for it :)

We'll need to install a very small library:

sudo pip install hackthederivative

```
import pandas as pd
from hackthederivative import complex_step_finite_diff as derivative
#https://pygotham.org/2015/talks/115/hack-the-derivative/

def frange(start,end,step=0.1):
    listing = [start]
    cur = start
    while cur < end:
        cur += step
        listing.append(cur)
    return listing

#(x-5)^2 - 3.5*(x-5) + 8
def calc_hours(volunteers):
    return (volunteers-5)**2 - 3.5*(volunteers - 5) + 8

df = pd.DataFrame()

for i in frange(0,10.0):
    tmp = {}
    tmp["value"] = i
    tmp["derivative"] = derivative(calc_hours,i)
    df = df.append(tmp,ignore_index=True)
df.to_csv("result.csv")
```

If you run the follow script you'll get back a csv with a list of values and their corresponding values under derivative. Next all we do is look for where the derivative is zero which happens somewhere between 6.7 and 6.8 so we'll say 6.75 and call it a day. So 6.75 is the optimal number of volunteers - the time we'll be minimized here to build a house! And notice all we had to do was eye-ball the solution. This kind of automated analysis is exactly the intention of this book - helping decision makers make the best decision possible so they can help the most people possible, without understanding everything that goes on.

Getting back to the analysis - since we can't exactly have .75 of a person, we'll round up to 7 people, and assume one person can take an extra long break OR everyone can take a slightly longer break and reach the optimal level of productivity :)

Now that we know how to work with a function - let's figure out how to use statistics to approximate a functional form. I will note this is a deeply complex part of mathematics, statistics, and computer science. We will be doing a very basic version of this. If you are interested in this topic I highly recommend checking out **this set of numerical courses taught and GW** or any advanced econometrics/statistics book.

We'll make use of the statsmodels package to approximate a non-linear function

```
#http://www.walkingrandomly.com/?p=5215 - this code was influenced by this
import numpy as np
from scipy.optimize import curve_fit
import random


xdata = np.array([5,6,7,3,4,8,9,12,17,4,14,9,25,12,24,15])
ydata = np.array([12,11,11,30,27,9,8,8,14,30,10,8,10,10,10,10])

def func(x,p1,p2):
    return p2*x**2+ p1*x

popt, pcov = curve_fit(func, xdata,ydata,p0=(7.0,9.0))
p1 = popt[0]
p2 = popt[1]
residuals = ydata - func(xdata,p1,p2)
f_residuals = sum(residuals**2)
print f_residuals
```

While this isn't a perfect example it motivates the point: we can approximate functions without know an exact functional form. However, we usually need to know something about our functional form - at least if we are going to use the above methods. There are some more advanced techniques that allow you to approximate a functional form, which we'll look at next. But for now, this is good enough. Okay, so let's remind ourselves of why we are doing this:

We want to know one simple thing: What is the optimal number of volunteers to build a house?

Now we are on the journey of using less and less data to learn something about our population of volunteers and when they work together best. Of course, it should be noted that the more assumptions we make the less we can trust our data, so we need to be careful to not trust our predictions too strongly, but more or less see them as a general guideline for how to move forward. If we are very good about our data collection - then of course we can readily trust our predictions and thus truly do something meaningful with our analysis. Some organizations like teaching mentorship programs, various house building organizations, and other organizations are in deep need of optimization and making use of this scheme could be helpful.

The next thing to do is stop guessing a functional form and actually come up with them. For this we will leverage a wonderful technique called a genetic algorithm. Genetic algorithms take simple functions and "evolve" them together via "mating". Over the "generations" clear solutions emerge (assuming you write good tests) and a clear functional form will be discovered. So the way generation works is iteration (via a while loop) and the mating happens via some combining function, typically either addition, subtraction or multiplication. Division can also be used, but I haven't seen it work particularly well. Of course, that could just be a failed on my part.

In order to get a complete understand of Genetic Algorithms I'd look at the following articles:

- **Great Introduction to Genetic Algorithms generally**
- **A great first example in Pyevolve**
- **A second example in Pyevolve**
- **A list of more advanced examples in Pyevolve**

Here we'll make use of the idea from **this post about genetic algorithms**. If you want to learn more about the details of GAs I suggest checking it out!

```
#Code comes from here: http://acodersmusings.blogspot.com/2009/07/curve-fitting-wit

from pyevolve import G1DList, GSimpleGA, Selectors, Scaling, DBAdapters
from random import seed, randint, random

def eval_polynomial(x, *coefficients):
    result = 0
    for exponent, coeff in enumerate(coefficients):
        result += coeff*x**exponent
    return result
```

```
def generate_fitness_function(sample_points):
    def fitness_function(chromosome):
        score = 0
        for ind,point in enumerate(sample_points):
            delta = abs(eval_polynomial(point[0], *chromosome) - point[1])
            score += delta
        score = -score
        return score
    return fitness_function

if __name__ == "__main__":
    # Generate a random polynomial, and generate sample points from it
    seed()

    #randomly initialize source polynomial
    source_polynomial = []
    for i in xrange(randint(1, 5)):
        source_polynomial.append(randint(-20,20))

    xdata = [5,6,7,3,4,8,9,12,17,4,14,9,25,12,24,15]
    ydata = [12,11,11,30,27,9,8,8,14,30,10,8,10,10,10,10]
    sample_points = zip(xdata,ydata)

    # Create the population
    genome = G1DList.G1DList(5)
    genome.evaluator.set(generate_fitness_function(sample_points))
    genome.setParams(rangemin=-50, rangemax=50)

    # Set up the engine
    ga = GSimpleGA.GSimpleGA(genome)
    ga.setPopulationSize(1000)
    ga.selector.set(Selectors.GRouletteWheel)

    # Change the scaling method
    pop = ga.getPopulation()
    pop.scaleMethod.set(Scaling.SigmaTruncScaling)

    # Start the algorithm, and print the results.
    ga.evolve(freq_stats=10)
    print(ga.bestIndividual())
    print("Source polynomial: " + repr(source_polynomial))
    print("Sample points: " + repr(sample_points))
```

The only real difference is that we are making use of predetermined sample data, rather than randomly generated sample data. This informs a useful extention given we usually have some sample data in the real world. So how do we interpret these results? Well if you read the first reference (above) you'd know that a fitness function is better when the score is higher. After running this algorithm a few times, I found the fitness score to be consistently pretty high, 8187568.661734 for example, gives an order of magnitude for how well we are doing.

So how do we take the results from this genetic algorithm and use that to do prediction? Well, if we looked at our `eval_polynomial` method and compare that with our result, we'd see the functional form. Essentially you just need to write down the solution from a run, to get an approximate functional form. Going deeper; to understand a solution we consider the "space" of possible functions. Our functional form restricts us to polynomial functions. So we know we'll always have a function of the form:

ToDo - convert this to ascii, sympy, or laTex

```
for i,j in xrange(n),xrange(m):
     summation(i*x**j)
```

Since we are searching a space for an optimal point in the space of functions, and since we intialize to a random function in the space, we won't get consistent solutions. This is because there are local maxima that our algorithm will get caught in before finding the global optimum. Of course, if we ran our algorithm for a sufficient number of iterations, we would always find the global optimum. However, finding such a function is usually unnecessary, at least for discrete and practical problems. This is because this function will be optimal for the entire space, which translates to having potentially millions of entries. If we have a very large set of data, finding an optimal solution may be of value, since we can be reasonably sure that our optimal point will actually reflect the decision problem of finding optimal volunteers, then we care. But we are working with like 10 data points? Totally not necessary. Here we are essentially using math to make a more educated guess. Which is fine, but it means that even if we end up with a local optimum, we are probably still doing just fine.

Here I'll include an example run of the algorithm, just to give us some context for finding a functional form:

```
eric@eric:~/Documents/tools_for_social_justice/code/chapter2$ python genetic_algori
Gen. 0 (0.00%): Max/Min/Avg Fitness(Raw) [52266340.08(-185812.00)/3066310.08(-493858
Gen. 10 (10.00%): Max/Min/Avg Fitness(Raw) [10702832.30(-11718.00)/0.00(-48719034.00
Gen. 20 (20.00%): Max/Min/Avg Fitness(Raw) [11484906.31(-5476.00)/0.00(-47442798.00)
Gen. 30 (30.00%): Max/Min/Avg Fitness(Raw) [9864874.22(-4394.00)/0.00(-46827932.00),
Gen. 40 (40.00%): Max/Min/Avg Fitness(Raw) [10133820.40(-1076.00)/0.00(-44872938.00)
Gen. 50 (50.00%): Max/Min/Avg Fitness(Raw) [7752485.98(-1058.00)/0.00(-37838204.00),
Gen. 60 (60.00%): Max/Min/Avg Fitness(Raw) [8451736.72(-912.00)/0.00(-39242148.00)/
Gen. 70 (70.00%): Max/Min/Avg Fitness(Raw) [8298231.31(-460.00)/0.00(-41262430.00)/
Gen. 80 (80.00%): Max/Min/Avg Fitness(Raw) [6134100.21(-144.00)/0.00(-42180212.00)/
Gen. 90 (90.00%): Max/Min/Avg Fitness(Raw) [5813074.61(-142.00)/0.00(-37781102.00)/
Gen. 100 (100.00%): Max/Min/Avg Fitness(Raw) [5944982.72(-90.00)/0.00(-45967430.00),
Total time elapsed: 15.229 seconds.
- GenomeBase
```

```
        Score:            -90.000000
        Fitness:         5944982.720309

        Params:         {'rangemax': 50, 'rangemin': -50}

        Slot [Evaluator] (Count: 1)
            Name: fitness_function - Weight: 0.50
        Slot [Initializator] (Count: 1)
            Name: G1DListInitializatorInteger - Weight: 0.50
            Doc:  Integer initialization function of G1DList

       This initializator accepts the *rangemin* and *rangemax* genome parameters.


        Slot [Mutator] (Count: 1)
            Name: G1DListMutatorSwap - Weight: 0.50
            Doc:  The mutator of G1DList, Swap Mutator

       .. note:: this mutator is :term:`Data Type Independent`


        Slot [Crossover] (Count: 1)
            Name: G1DListCrossoverSinglePoint - Weight: 0.50
            Doc:  The crossover of G1DList, Single Point

       .. warning:: You can't use this crossover method for lists with just one element.



    - G1DList
        List size:     5
        List:         [8, 0, 0, 0, 0]


    Source polynomial: [-2, -12]
    Sample points: [(5, 12), (6, 11), (7, 11), (3, 30), (4, 27), (8, 9), (9, 8), (12, 8), (17, 14
```

ToDo - explain interpretation of results. ToDo visualize polynomial: **http://www.arachnoid.com/sage/polynomial.html**

From here we can take our functional form and apply it to our data. From there we can apply our derivative and find an optimal number of volunteers. I'll leave it as an exercise to finish out this analysis since at this point you should have all the pieces to do this. Of course, if you get stuck or just need to get something done **check out this explanation for the completed example** ToDo - write the completed example

Raising funds, finding bad guys, with targeted advertising

- raising funds - predictive marketing
- finding bad guys - understanding your population with descriptive statistics

The goal of the last section was really to motivate mathematics as useful for social justice. Here we'll look at some more concrete examples of how one might use statistcs or machine learning - with a time tested practical approach - marketing with data science. Central to the nonprofit landscape is raising funds. The best way to do that is to understand who is most likely to give to your cause and then let them know that their money would be greatly appreciated. The ideas we'll discuss here were used extensively in the last two presidential campaigns to raise funds and are already being used by many non-profits. But not every non-profit in the world can afford the fancy marketing firm with the extremely well educated team of data scientists. So the intention here is more of a DIY approach to machine learning based marketing.

Before moving onto the techniques it's worth mentioning the second application. Finding bad guys. A very powerful/kind of creepy innovation of the last ten years or so has been the ability to take a set of actions users take on the internet, and market to them directly. Essentially this is a classification problem - where the typical set up is, what kind of person would like to by our product? And then you create sections of the population that are more and less likely to want to buy your product based on the behavior they take on the internet, their demographic information, and anything else you can figure out through the internet. But really what this is doing is trying to reverse engineer a set of actions you'd take given certain circumstances OR a set of preferences you have regarding a set of topics. We'll use this mindset to answer a very different question - how do you find pedafiles on the internet? The techniques presented for this problem and solution in particular open up a range of moral questions:

1) Can we structure investigations to find someone likely to commit a crime, before they may have done so?

2) Can we legally do these kinds of investigations?

Obviously for a less severe crime these questions become harder to answer, and everyone may not agree this is that bad a crime. But I come from a unique prospective (in this regard), I've seen first hand what pedafila can lead to - years of psychological and emotional damage, which can send someone spiraling down the path to becoming a sex slave. An area I have devoted my lifes work to ending. And so, for me, it's worth it to have the knowledge and ability to stop this horrible crime from happening. It's worth it to ask the hard questions about liberty and freedom in the larger context of this kind of suffering. I'm not saying we necessarily should be always doing this type of analysis, and I'm also not saying that particulars don't matter, using these types of techniques for law enforcement should be done with extreme care. But I am saying that it's worth it to have the conversation and for a very small set of cases, do this type of investigation.

So without further ado, let's learn how to be marketing experts!

Below is some reference material on terms often used in the industry:

- **microsoft - intro to data mining**
    - The most important concept in the above reference is lift charts, but the whole things is pretty decent explanation
- **wikipedia - data mining**
    - there is a whole section on business applications
- **MIT - data mining class**

Between those three references, you'll understand all the basics about data mining you'll ever need. Now we are ready to formulate the canonical problem in data driven marketing:

How do we improve the success of a marketing campaign?

To understand how to solve said problem, we'll need to understand first the structure of a marketing campaign and then we'll be able to look into some solutions.

Typically a marketing campaign is done via distribution of some media; either video, text, pictures or some combination of the three and sent through distribution channels; social networks, email, the mail system, commercials, ads on websites, spam texting, or cold calls or word of mouth. We can treat each campaign as an experiment, where we want to understand how spread of information affects how much money we bring in. For some people, more material is going to mean they are more likely to give, for others it means they'll be less likely to give. For some people it honestly depends on the cause. And for other people it won't matter, they'll never give money. The goal of a marketing campaign is to figure out who our potential high value targets are - the folks who will give consistently and more than the average doner, and make them feel as comfortable as possible. However there are lot of strategies. You don't necessarily need to suck up to a bunch of rich people, nor do you need your messaging to pander to the 1%. A message that speaks to a wide range of donors, that maximizes the amount of money they want to give, may be more effective than going after a small proportion of folks who many be more fickle. Of course, you need to look at the costs of running a campaign when considering all of this.

Now that we understand the mechanisms in play, let's go over how to structure our experiment and begin to understand how we might make improvements over existing campaigns.

Let's assume the simplest example - one mechanism for distribution and one media with one versions of the content.

So we'll do email - since the metrics are intuitive and we'll say that it's just a written email with some hyper links to web addresses we control.

So what are our metrics?

1) Open rate - What percentage of people openned emails?

2) Click through rate - what percentage of people clicked on our hyper links?

3) Conversion rate - what percentage of people read our email, went to our website, and gave money or volunteered for an event?

More or less what we capture with this process is the stages of conversion. Of course, what we want is to maximize the 3rd metric, our conversion rate however, understanding the first two give of a sense of who we might be able to convert. Folks who just delete the email are going to be dead ends, at least for this campaign. Maybe they ended up on our mailing list for some specific project and only care to be contacted about that, or perhaps they just wanted to talk to someone on the street for a few minutes but don't actually care about getting involved.

There are a number of subsequent metrics to each of these metrics:

- What percentage of people openned emails?
    - What time of day was the email openned?
    - What was the subject line?
        - Was a formal or casual subject line used?
        - Was the person's name used in the subject line?
            - First name or last name used?
    - What was the email address that sent the email?
- what percentage of people clicked on our hyper links?
    - How long did they wait between openning the email and clicking the link?
    - which link did they click?
        - where was the link on the page?
        - What color was the link?
        - Was it an html link or a plain text hyper link?
        - Was it underlined or not?
    - What wording did the link have?
        - was it one word or multiple words?
        - (assuming you have a few content types) What content type was the link? (a typical example is: volunteering or donating money)
- what percentage of people read our email, went to our website, and gave money or volunteered for an event?

  - What percentage of those people had given money in the past?
    - What percentage of those people have family or friends who have volunteered in the past?
    - What percentage of those people who gave money have given money to similar causes in the past?
    - What percentage of those people who gave money have given money to any charity in the past?
  - what percentage of those people had volunteered for an event in the past?
    - What percentage of those people have family or friends who have given money in the past?
    - What percentage of those people have volunteered for similar organizations in the past?
    - What percentage of those people have volunteered for anything charity in the past?

Other important questions:

```
* What area does this person live in?
    * What is the average income of the area?
    * Has this issue been a problem for this area?
    * How much presence does the non-profit have in the area?
        * How much work has been done in the area?
        * How many past marketing campaigns have been done in the area?
        * How many people receieve services from the non-profit in the area?
    * Has the person ever volunteered with the non-profit?
        * How often does this person volunteer?
        * How many friends does the person have who volunteer with the non-profit?
    * Demographic questions:
        * What ethnicity is the person?
        * Are they male or female?
        * How old are they?
        * How many years of school?
        * Currently in school?
            * Currently in high school?
            * Currently in undergrad?
            * Currently from in grad school?
        * Graduated from undergrad?
            * Related program of study? (This question is essentially asking if they aquired s
        * Graduated from a graduate program?
            * Related program of study?
```

There are problem a few important questions I missed but, this set of questions gives you a sense for the type of data you should be collecting and leveraging. Once we have all this information, we store the results for each person in a table, each question becomes a variable and answers the question. This information should be boolean or numerical in nature. We could use text, but that will complicate the analysis and make things more difficult in the context of prediction. Therefore text based answers are discouraged for spreadsheets.

From here we take in all the measurable variables - to make an attempt at predicting the three variables we really care about:

1) Open rate - What percentage of people openned emails?

2) Click through rate - what percentage of people clicked on our hyper links?

3) Conversion rate - what percentage of people read our email, went to our website, and gave money or volunteered for an event?

Using each of those variables that are measurable, before hand, we are able to construct a statistical model involving each of the variables. Then we use marketing campaigns we've run in the past to fight a mathematical model. This mathematical model will give use a few things we care about:

1) It will let us test how important each of the variables is for prediction. This way we know what variables we can safely ignore for collection, which will save us both money and time. Of course, we may still want to collect at least some of these variables as it may be the case that this variable matters, depending on the data collection that is done.

2) It will give us a model for predicting where to put our efforts for maximal pay off. Once we have a reasonable sense of our model we can begin to use it to ask questions, carrying out marketing experiments without sending a single email. Once we find a set of parameters that maximize our chance of success (given the model), we carry out a single marketing campaign, gathering statistics, and thus we are able to not only carry out an optimal solution this time, but we are able to assess further and refine our model. This is sort of a learning based approach to marketing. By learning our population over time, we are able to gain valuable insights to our population of potential volunteers and donors, allowing us to make optimal decisions, raise the necessary funds, and help the maximum number of people.

To do this modeling we'll make use of statsmodels.

installation: sudo pip install statsmodels

Statsmodels includes a linear regression method (which will be our main work horse). Since we don't want to actually want to run a marketing campaign (at leat not in the context of this book), I'll generate the data randomly. Below is my generate data python file:

```
# Data to generate:
# * What percentage of people openned emails?
#     * What time of day was the email openned?
#     * What was the subject line?
#         * Was a formal or casual subject line used?
#         * Was the person's name used in the subject line?
#             * First name or last name used?
#     * What was the email address that sent the email?

import random
import pandas as pd

df = pd.DataFrame()
for i in xrange(100):
    record = {}
    #What percentage of people openned emails
    record["open_rate"] = random.random() + 0.15
    if record["open_rate"] > 1.0:
        record["open_rate"] = 1.0

    #What time of day was the email openned?
    #{"morning":1,"midday":2,"evening":3}
    record["time_of_day"] = random.randint(1,3)

    #What was the subject line?
    #{"Hello there!":1,"Oh hi":2}
    record["subject_line"] = random.randint(1,2)

    #Was a formal or casual subject line used?
    #{"formal":1,"casual":2}
    record["formal_casual_subject"] = random.randint(1,2)

    #Was the person's name used in the subject line?
    #{"persons name":1,"no name present":2}
    record["name_present"] = random.randint(1,2)

    #First name or last name used?
    #{"first_name":1,"last_name":2}
    record["first_last_name"] = random.randint(1,2)

    #What was the email address that sent the email?
    #{"hello@non_profit.org":1,"ericschles@non_profit.org":2}
    record["email_address"] = random.randint(1,2)
    df = df.append(record,ignore_index=True)

df.to_csv("marketing_data.csv")
```

Notice that we don't use strings with any of the variables. Instead we make use of a concept called dummy variables. Here we map a string to a specific number. You might not think this is useful, but in fact it proves to be a very powerful analytic technique and a great work around :)

We'll be making use of much of the code from **this example** to build our model.

Here is our analysis:

```
import statsmodels.api as sm
import pandas as pd
from patsy import dmatrices

df = pd.DataFrame().from_csv("marketing_data.csv")

y, X = dmatrices('open_rate ~ email_address + first_last_name + formal_casual_subjec

lm = sm.OLS(y,X)
result = lm.fit()
print result.summary()
```

Notice how easy it is to write down a model:

`open_rate ~ email_address + first_last_name + formal_casual_subject + name_present + subject_line + time_of_day`

This basically says `open_rate` is related to all the other variables. So we formulate this as a hypothesis that we use linear regression to test against. Below is the results:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:              open_rate   R-squared:                       0.104
Model:                            OLS   Adj. R-squared:                  0.046
Method:                 Least Squares   F-statistic:                     1.795
Date:                Thu, 10 Sep 2015   Prob (F-statistic):              0.109
Time:                        17:47:57   Log-Likelihood:                0.81087
No. Observations:                 100   AIC:                             12.38
Df Residuals:                      93   BIC:                             30.61
Df Model:                           6
Covariance Type:            nonrobust
==========================================================================================
                          coef    std err          t      P>|t|      [95.0% Conf.
------------------------------------------------------------------------------------------
Intercept               0.3844      0.180      2.134      0.035         0.027
email_address          -0.0548      0.051     -1.072      0.286        -0.156
first_last_name         0.0750      0.052      1.442      0.153        -0.028
formal_casual_subject   0.0946      0.051      1.872      0.064        -0.006
name_present           -0.0567      0.051     -1.123      0.264        -0.157
subject_line            0.1016      0.053      1.908      0.059        -0.004
time_of_day             0.0069      0.032      0.219      0.827        -0.056
==============================================================================
Omnibus:                       11.141   Durbin-Watson:                   2.026
Prob(Omnibus):                  0.004   Jarque-Bera (JB):                4.775
Skew:                          -0.277   Prob(JB):                       0.0919
Kurtosis:                       2.084   Cond. No.                         30.0
==============================================================================
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

One quick note before we go into the interpretation - none of our values will or should work for this answer since our data was randomly generated. Naturally there should be non correlation.

We can safely ignore some of this, here are the important things:

R-squared - this is the overall measure of fit of the model. A good model with fit with 95% accuracy.
F-Statistics - tests the hypothesis - Are all the variables jointly useless? The p-values for each of the variables.

R-squared is bounded between 0-1.0 where 1.0 is a perfect fit and 0 is an extremely poor fit.

To understand the F-statistic we must first understand the hypothesis: A typical null hypothesis is the coefficients of all variables are jointly 0. In other words, none of the variables are statistically significant.

In this case, the Prob(F-statistic) is 0.109 which is not less than the critical 0.05 level (95% confidence) so we fail to reject the null hypothesis. There the variables are jointly not statistically significant, in other words, none of these variables matter. Remember, we expected this because we used randomly generated data. So it's not a big deal.

Finally, we can test each variable individually to look for significance:

```
                          coef    std err          t      P>|t|      [95.0% Conf. Int.]
-----------------------------------------------------------------------------------------
Intercept               0.3844      0.180      2.134      0.035         0.027     0.742
email_address          -0.0548      0.051     -1.072      0.286        -0.156     0.047
first_last_name         0.0750      0.052      1.442      0.153        -0.028     0.178
formal_casual_subject   0.0946      0.051      1.872      0.064        -0.006     0.195
name_present           -0.0567      0.051     -1.123      0.264        -0.157     0.044
subject_line            0.1016      0.053      1.908      0.059        -0.004     0.207
time_of_day             0.0069      0.032      0.219      0.827        -0.056     0.070
```

Notice none of the values p values meets the criteria for statistical significance (all of them are above the 0.05 mark) and so we reject each of the variables.

So this gives a complete picture of whether not we can trust any of variables we are measuring. Assuming any of our variables are statistically significant we can use the coef column to write down a model and then begin to ask mathematical questions. In this case our model would look like this (in python code):

```
def model(email_Address,first_last_name,formal_casual_subject,name_present,subject_line,time_c
    return 0.3844 -0.0548*email_address + 0.75*first_last_name + 0.0946*formal_casual_subject
```

Then we'd need only run experiments and then our model would actually *tell* us what our open rate would be! How nuts is that?! Of course, this model would probably only work for a short amount of time AND this model in particular of course would not work (because we used bad data).

We could expand this example to cover all the cases - tv, email, mail, social media. Each has it's own set of questions, but the method of finding solutions is pretty much the same as above. An important caveat - you'll need to account for the cross over effect between multiple mediums and account for that in your model explicitly via specific questions. Other than that, the situation is the same as the one we've described. Happy marketing!

Now it's time to move onto our next marketing related application

4 Finding Bad Guys.

This application will actually be fairly related to our last application, except we'll only focus how to market to them, and then understand our population with descriptive statistics rather than try to do anything predictive. We only care about catching bad guys, not predicting how effective we'll be - because taking even one murder, pedafile, human trafficker, etc. off of the street is worth it.

There are a few ways to find bad guys. However, typically you want to figure out where they hang out (on the internet), and then observe that they've committed a crime. Then you arrest. We'll look at the specific example of child sex trafficking and attempt to catch both purchasers of underaged sex as well as the suppliers.

Catching a Buyer of Sex

Catching a buyer of sex is actually pretty easy. All you need to do is put up a fake website, post a fake ad to a number of sex buying sites and then wait. This is very similar to the marketing campaign because you are posting content to a medium. You can refine your advertisement to make sure you maximize who you are getting by doing the analysis defined above.

So what do you need to identify a person uniquely?

Well there are typically two pieces of information that come in handy - their IP address (which helps you get a location) and their phone number. First let's look at how to capture IP addresses, by putting up a free website. We'll use Heroku for the website, however you can use whatever you want. Please do note that the way you capture IP addresses will change depending on what server you use and the security protocols they have in place.

The complete code can be found **here**

Here is our app.py file: (yes the name matters)

```
from flask import Flask, render_template, request
from flask.ext.sqlalchemy import SQLAlchemy
```

```
import os
import datetime
app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = os.environ['DATABASE_URL']
db = SQLAlchemy(app)

#http://blog.y3xz.com/blog/2012/08/16/flask-and-postgresql-on-heroku

class Logger(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    ip_address = db.Column(db.String(400))
    timestamp = db.Column(db.DateTime, default=datetime.datetime.now)

    def __init__(self,ip_address):
        self.ip_address = ip_address

    def __repr__(self):
        return '<ip_addr %r>' % self.ip_address


@app.route("/index")
@app.route("/")
def index():
    if request.headers.getlist("X-Forwarded-For"):
           ip = request.headers.getlist("X-Forwarded-For")[0]
    elif request.access_route:
        ip = request.access_route
    else:
           ip = request.remote_addr
    log = Logger(ip)
    db.session.add(log)
    db.session.commit()
    return render_template("index.html")

if __name__ == '__main__':
    app.run(debug=True)
```

The first thing to understand here is how the IP logging happens. That occurs here:

```
if request.headers.getlist("X-Forwarded-For"):
        ip = request.headers.getlist("X-Forwarded-For")[0]
elif request.access_route:
    ip = request.access_route
else:
    ip = request.remote_addr
```

First let’s understand `request.remote_addr` - this is the simplest way to get an incoming IP address - unfortunately heroku typically blocks the true IP address for security reasons. On local web servers, for testing, or on certain web servers `request.remote_addr`will store the address of anyone visiting the website. Also, the `request.access_route` does more or less the same thing as `remote_addr`. The most effective way of getting the IP address is typically by reading the headers directly. For more information on how to do this and a complete explanation check out this **stackoverflow post**.

Here is our Procfile:

```
web: gunicorn app:app
```

Here is our requirements.txt:

```
Flask==0.10.1
gunicorn==19.1.1
Flask-SQLAlchemy==2.0
itsdangerous==0.24
Jinja2==2.7.3
Werkzeug==0.9.6
wsgiref==0.1.2
MarkupSafe==0.23
psycopg2==2.5.4
```

Make sure to have a templates and static folder. In your templates folder add a index.html file. It can be whatever you want, since all we care about is capturing the IP address, but here is the index.html I used:

```
<!doctype html>
<html>
<head>
</head>
<body>
<p>Hello there</p>
</body>
</html>
```

Now that we have all the files we'll need let's go ahead a set up a github repository. If you don't already have git or github installed/ don't have a github account, please check out there **installation guides**

Once you have git installed initialize the git repo by following this set of commands:

Open a terminal - cd to the top level directory of your code for this project (if it lives in a folder with other projects make a new one) and then type the following commands after changing into the folder with only the code for this project:

```
git init
git add -A
git commit -a -m "first commit"
git remote add origin git@github.com:EricSchles/[name of top level project folder].g
git push -u origin master
```

Where [name of top level project folder] is the name of the project folder that will appear on github. It is good practice to make sure your names are consistent locally and on github. (However they need not be).

Next we'll create a heroku account (assuming you don't have one already):

Here are the instructions for getting started with python. **The next thing to do is set our database which you can do here**.

Now that we know how to capture IP addresses, let's work on capturing phone numbers. It is good practice to include phone numbers both on the website AND in the advertisement.

The complete code can be found here

Nothing will change in the Procfile

Here is my requirements.txt file:

```
Flask==0.10.1
gunicorn==19.1.1
Flask-SQLAlchemy==2.0
itsdangerous==0.24
Jinja2==2.7.3
Werkzeug==0.9.6
wsgiref==0.1.2
MarkupSafe==0.23
psycopg2==2.5.4
twilio==4.4.0
```

below is app.py:

```
from flask import Flask, request, redirect
import twilio.twiml
from flask.ext.sqlalchemy import SQLAlchemy
import os
import datetime

app = Flask(__name__)
#app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:////database.db"
app.config["SQLALCHEMY_DATABASE_URI"] = os.environ['DATABASE_URL']
db = SQLAlchemy(app)

# Try adding your own number to this li
class Logger(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    caller = db.Column(db.String(400))
    timestamp = db.Column(db.DateTime, default=datetime.datetime.now)

    def __init__(self,caller):
        self.caller = caller

    def __repr__(self):
        return '<ip_addr %r>' % self.ip_address

@app.route("/", methods=['GET', 'POST'])
def hello_monkey():
    # Get the caller's phone number from the incoming Twilio request
    from_number = request.values.get('From', None)
    resp = twilio.twiml.Response()
    resp.say("Hello Monkey")
    print from_number
    if from_number:
        call = Logger(from_number)
        db.session.add(call)
        db.session.commit()
    return str(resp)

if __name__ == "__main__":
    app.run(debug=True)
```

To install this, you can use the same procedure that we used above, for the last app.

Notice that we more or less steal this example from **the twilio basic example**. Please feel free to change `"Hello Monkey"` to whatever message you would like. Once we have the phone numbers, they are stored in our logger table in our database. Then we can go through an investigate each of the numbers, doing link analysis against other numbers we may have on file from trafficking cases or from other unrelated crimes. Of course, once a case is over, the case is sealed. Of course, just because a case is sealed, doesn't mean you can't get it unsealed. (Assuming you work in law enforcement).

The final step is to add your website, once you are sure it works to your twilio account. To do that, you simply need to create a twilio account and then head over to:

https://www.twilio.com/user/account/phone-numbers/incoming

From there find the Request URL field and then copy paste the name of your url from heroku and you are all done.

Notice that we don't buy a domain, since the website is only up to act as a datastore. Of course, you could include this in your other website if you are short on dynos and don't want to pay for more resources. But I personally think it's a good idea to keep them seperate.

Now that we have phone numbers and IP addresses (in theory), we can begin to find names and locations. We can use a number of reverse look up services (I won't mention any by name here) to find a name from a phone number. If we are in law enforcement, then we can use subpoena power and other look up powers to find out people's personal information from phone numbers. From there we might be interested in making arrests, but first we might want to understand our population with descriptive statistics.

A few that are typically useful are:

- The average, as a snapshot, as well as over time
- The median, as a snapshot, as well as over time
- The variance, from month to month
- A random variable that models the population well

All of this together will enform a way of tracking our population and thus understanding when we've made real gains. Moving from the abstract to a discrete example. The Manhattan DAs office started a unit called Crime Strategies that was devoted to stopping or reducing gang related activity, specifically homocide (among other things). This unit made use of a set of descriptive statistics to track how much progress it was making over time. By looking at the data and making a calculated, learned approach, you can have a real sense of how well you are doing. The above statistics may seem trivial or simple, but when applied to areas where no such statistics are generated, we can say whether or not things are getting better, and then change tactics if necessary to ensure that we are doing better. Of course, making sure you are using the right metrics will always matter. Fortunately, in the law enforcement domain, the metrics are usually simple - how many people died by homocide? For instance. We'll continue with our example of tracking pedafiles, but I wanted it to be clear that this type of thinking does get used in the real world and has a massive impact, when used correctly.

So what metrics are worth tracking:

First off, we'll take moving metrics, where we look over the course of a month. So moving averages, moving medians, variances dependent on time frame, and random variables will be fit to our data over time as well, looking at different time scales. One of the reasons to do this, is because this sort of crime doesn't happen in a vaccum. Instead, it happens in the context of policy change, law enforcement budget increases and decreases, changes in public sentiment, and in different economic conditions. Therefore, things need to be continually remeasured over time. Of course, they won't always change too much, but it's usually a good idea to make sure.

Some questions worth asking:

Metrics about the data: 1) How has the number of IP addresses that visited the website changed? (Answered with moving average and moving median)

2) How has the number of people who called the number changed? (Answered with a moving average and moving median)

3) How much does the number of IP addresses vary over a day, over a week, over a month? (Answered with variance)

4) How random should we expect the number of IP addresses captured to be? (answered by knowing the random variable)

5) Are there seaonality affects? (answered by looking at moving average)

Metrics about the population: 1) What is the average income of people looking for underaged commercial sex?

2) What is the earning power of the top 15% of individuals?

3) What is the average education level of the people looking for underaged commercial sex?

4) What is the most common industry of people looking for underaged commercial sex?

5) What is the least common industry of people lookign for underaged commercial sex?

6) What is the gender of these people, by percentage?

7) What is the average age of these people?

8) What is the distribution of ages? What distribution do these ages most closely follow?

9) What are the differences between the top 10 most frequent individuals and the average individual looking to buy sex?

The way you'd get at number 9 is by looking for repeated phone numbers of the course of a year and then doing some analysis between them and a collection of average people from your population.

5 Doing social network analysis to catch bad guys and make connections

We've already sort of seen an example of this in the last section, well sort of. We've seen how to do the data collection, but not the social network analysis. The way social network analysis works, is we build connections between entities based on a combination of soft and hard links. A hard link is something we can verify a relationship with. For instance, we can usually create a hard link between a person's name and phone number. IE, we have proof. A soft link is something we aren't sure about, but may imply a relationship. For instance, say two posters on a sex buying website use a similar writing style. This is enough to say there might be a connection, but it isn't enough hard evidence to say there is a connection for sure. How you might measure writing style is something you might remember from chapter 1 when we went over named entity recognition.

When building a set of social network graphs, for say, human trafficking all one needs to do is take in and label various entities and then draw connections between them. Entities of interest include - people and organizations. Sometimes its useful to represent individuals intermediately by the phone numbers they tend call from, this way we can draw direct connections between phone number conversations.

So how might we structure a social network analysis, in code?

Back to our analysis! Say we have sets of phone numbers, with the number on the left representing the caller and the phone number on the right representing who they called, like so:

```
source    , target
5167774444, 5164142222
5167774444, 5162829999
```

Here of course we have an abridged version of such a list. However it's enough to make the point. Here's how we could represent this set of connections in code:

```
import pandas as pd
import d3py
import networkx as nx
import matplotlib.pyplot as plt

df = pd.DataFrame().from_csv("sna_example1.csv",index_col=False)
G = nx.from_pandas_dataframe(df,"source","target")

nx.draw(G, with_labels=True)
plt.savefig("sna_example1.png")
```

If you run the above code, you'll see a print out of three nodes and two edges, connected by the source number. While this doesn't seem very exciting (and it isn't). What would happen if we combined two such graphs? Let's investigate!

Now we'll have two csv's:

First - (same as before)

```
source    , target
5167774444, 5164142222
5167774444, 5162829999
```

Second - (new one)

```
source    , target
4872223232, 5167774444
4872223232, 5162829999
```

Here's the code to visualize both of these excel documents at once:

```
import pandas as pd
import d3py
import networkx as nx
import matplotlib.pyplot as plt

df1 = pd.DataFrame().from_csv("sna_example1.csv",index_col=False)
df2 = pd.DataFrame().from_csv("sna_example2.csv",index_col=False)

G1 = nx.from_pandas_dataframe(df1,"source","target")
G2 = nx.from_pandas_dataframe(df2,"source","target")

G = nx.compose(G1,G2)
```

```
nx.draw(G, with_labels=True)
plt.savefig("sna_example2.png")
```

If you run the above code and view `sna_example2.png` you'll notice that a few connections were made betwee the two graphs. While this is easy to see with the above two source files, the generalization should be obvious - with thousands of numbers and many excel documents, we are able to draw out connections very very fast.

The last of the simple examples we'll cover involes the generalization of the above examples. We won't include anymore excel documents, but instead re-run example two, in the most generalized way possible.

```
from glob import glob
import pandas as pd
import d3py
import networkx as nx
import matplotlib.pyplot as plt

dfs = []
for CSV in glob("sna_example*.csv"):
    dfs.append(pd.DataFrame().from_csv(CSV,index_col=False))

#we assume column names are consistent across all csv's, please ensure this is true!
graphs = []
for df in dfs:
    graphs.append(nx.from_pandas_dataframe(df,"source","target"))

G = nx.compose_all(graphs)
nx.draw(G, with_labels=True)
plt.savefig("sna_example3.png")
```

Notice that we used phone numbers and csvs. But this generalizes easily to labelling any sort of connection set between two values in different columns. We also could have used a database connection or anything that can be fed into a pandas dataframe. I'll simply leave **a reference to connection types here**. Before going onto connections a note - In general for more information on how connections should be structured, you'll need to engage in actual collaboration with a stack holder. This is because what constitutes a connection for different types of crime or investigations can be vague. In order to understand how one might structure a set of connections we'll look at how to make connections when trying to find instances of human trafficking.

Before moving forward with this exposition, it is worth noting that law enforcement has subpoena power - that means they can request everything about you. So not everything we are going to talk about is something your average re-searcher working on their own will be able to get their hands on. But, that doesn't mean you can't build tools and then give them to law enforcement :)

- Connecting people by phone number - Essentially the analysis that we did before. Being able to say who knows each other, based on who has called who, is extremely powerful and can save you a ton of time from an investigation stand point. Get the data, and let the computer do all the work for you, finding connections.
- Finding victims automatically - One of the things I've managed to come up with, is a way to potentially find victims of human trafficking - You do this by comparing pictures of missing persons with pictures from ads on popular comerical sex buying websites like backpage.com. If the faces are a match, or reasonably close, then you have enough probable cause to start an investigation (because the person in one of the pictures is missing)
- Comparing twitter accounts - let's say you know someone is a trafficker (or are reasonably sure) and you happen to see your target talking to another person about the victim, possibly bragging about how they beat them up. This person is also likely a trafficker. For whatever reason, they really like to brag in public spaces to one another. Using text analytics and creating labels for bragging about violence you can establish a lot about a person, based on feeds.
- Comparing facebook accounts - the same analysis can typically be carried out for a facebook account as a twitter account.

There are certainly more ways to draw connections, unfortunately they require strictly closed information sources and therefore cannot be disclosed. Hopefully, this gives you a feel for how you might engage in creating a graph! Some place this analysis could be taken - using disperate data sources, you could create weighted graphs. A way to structure this - how many different ways are two individuals connected, via a simple frequency count. So if you find that a piece of information indicates a connection, that would be a frequency of one. If you find multiple sources of information (that aren't all derived from the same source) showing connection between the two individuals, then you can claim the connection is stronger.

From there you can use the page rank algorithm, for instance:

```
calculated_page_rank = nx.pagerank(graph, weight='weight')

#most important words in ascending order of importance
key_individuals = sorted(calculated_page_rank, key=calculated_page_rank.get, reverse
```

Using this listing, we can see who the most important members of the network are, automatically. If there is a hierarchy in a group and we have a representative set of information, then using this analysis, we'll be able to discern who they are, and how much power they have in the group.

This is merely an example of a graph based algorithm that can used to do social network analysis. There are far more, and sadly they are beyond the scope of this text.

Chapter 3 - Data Visualization

So far we've seen how to process data and make use of statistics/machine learning to automate analysis and get stuff done. Unfortunately for most non-technical folks working in government and non-profit land, this is really not useful. There are a few reasons behind this:

1. You're going to leave - The government and/or non-profit don't pay very well and eventually you'll need to actually make a living, paying down student debt, raising a family, covering medical expenses, going to the dentist and a whole other laundry list of things you won't be able to afford. Everyone knows you are leaving, because you won't be the first technologist who's ever tried to make a difference and you won't be the last. But sadly, they really can't afford you and never will be able to.
2. They don't actually understand what you've built - very, very few non-profits or government folks understand or embrace technology. Which is the reason you can have such a large impact. But it also means they don't really understand what you are doing, why you are doing it or how you are doing it. So you need to do a lot of hand holding throughout the process

For these reasons, building clean, pretty interfaces is almost twice as important as actually making sure the technology works. This is because adoption of technology is very, very low in the government/ non-profit space. If they don't understand what you've made intuitively (aka without really understanding) they won't use it. No matter how much money the organization as a whole paid for the software, service or technology.

So we are going to talk about design estetic, work flow, and a few other things.

Design Process

Website built for the social justice space need to be as simple as possible. You can do a max of three things per page, but really you should only do 1 thing.

And each tool should never really do more than 1 to two types of tasks. It's also important to understand, that no matter how useful a tool or set of tools might be, there is a finite amount of brain space for actually using tools, no matter how much easier the next thing might make someones life. So, you need to make sure that whatever you build is as useful as possible, even if it's technically very boring.

In fact, the tool that has had the highest adoption rate since I started working in government is the PDF to CSV tool I touched on in chapter one. All the machine learning stuff was put into production, but it took 2 years to find the right client - a national organization of folks who came from fortune 500 companies, to understand and make use of those analyses. So, the point, keep it simple, otherwise no one will use it.

So how should design of websites like this look? I don't know. And neither does anyone else. There are certain rules I've picked up, but in all honesty, you'll need to do an agile iterative design, before you have any idea about what you are going to build or how it's going to look. If you can add bootstrap, great! But don't do it until after you've iterated on your design with all the members of your team.

So how do you do agile in the government or non-profit space?

1. Find a person who is excited about technology - this is going to be your alpha tester. He or she will be your companion on the journey to creating something useful. For me, this was a young woman named Rachel. Over the past year we've worked together extremely closely. It hasn't always been fun, but its almost always been productive. The way our interactions typically work is, I get a requirement from my boss or I pitch an idea. I explain the idea to Rachel, she gets way too excited and thinks I'll finish it tomorrow, and then I get to work. Typically later that day I'll have a prototype with a basic flask app and a button. The button will give her the output from the task and she'll give feedback about the look of the output, the interface and the workflow. I'll take her notes and iterate on the design. Usually we'll do this three to five times.
2. Show someone else - Usually by this point, a week or two later, I'll have something that other people will understand and be able to use. Typically all the design work has already happened with Rachel, since she knows the rest of the team very well. However, making sure the workflow is intuitive to other folks is important, because it gets their buy in. Its important to get as much of the teams buy-in as possible. This ensures that more people will adopt the tool. Because remember, just because you build, it doesn't mean they'll use it. Even if its what they asked for.
3. Get by in from management - Its a sad truth, but no matter how good a tool is or how much time its going to save, if your boss doesn't care, it

won't get put into production. So make sure you have his or her approval. It's also important that they know about the project from the beginning. So although you have someone else who is working on it, make sure they are onboard with what you are doing.

4. Get ITs buy in - Unfortunately my person situation didn't really allow for this. So I recommend making friends with IT early on in your time. Your going to need them on your side if you want to get stuff into production, at least in many government institutions. Unfortunately, most of the time ITs (preference) is to say 'no' to new projects. So if you can get ally's here, it will make a world of difference. Assuming you can convince them to be on your side and actually do stuff that's meaningful, you should run it by them after you tell your boss, but before making the request to put it into production. If possible show them a prototype.

So really steps (3) and (4) happen after you've already started the iteration process with your alpha tester, but before you've completed it. So maybe after the second iteration.

Design Examples

Now that we've touched on the design process, let's talk about the design itself.

Installation

flask - sudo pip install flask

bootstrap - installation of bootstrap is slightly more involved than previous software. You'll need to either get a bunch of CDNs (content delivery networks) or download the (minified) source directly.

This simple example shows you how to include the Bootstrap CDNs in a simple HTML page:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap

    <!-- Optional theme -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap

    <!-- Latest compiled and minified JavaScript -->
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></scrip
    <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"
</head>
<body>

</body>
</html>
```

it's also worth noting that we can install bootstrap/jquery via npm or bower. With the following commands:

bower:

- `bower install bootstrap`
- `bower install jquery` npm:
- `npm install bootstrap`
- `npm install jquery`

angular.js - we'll also need either a **CDN** or we'll need to install angular locally with the command `npm install angular`. It's worth noting that npm installs packages to a directory rather than globally. You can do the following if you'd prefer to have certain packages globally:

`npm config set prefix /usr/local` `npm install -g [package name]` `which [package name] #testing everything worked`

Building our first flask app

Flask is one of the simplest and most useful web frameworks in the world. This is not only because you can create servers with the absolute minimum number of explicit lines of code, but because it has a ton of extensions that make it robust when necessary, but flexible enough to just get stuff done.

The following example is done entirely from the python shell:

```
>>> from flask import Flask
>>> app = Flask(__name__)
>>> @app.route("/",methods=["GET","POST"])
... def index():
...     return "Hello there"
...
>>> app.run()
```

The same app can be found **here**

As you can see, writing a very minimal flask app is extremely easy. And the good news is it continues to be easy for larger apps. So how does a flask app work?

First there is the flask object - `Flask` - this object creates the context for the running website. You can add routes to the app context and then run those routes. A route represents a url path and the action(s) the server will take when a browser visits the page, or more technically when a request is made to the endpoint. Notice that route takes one parameter by default and lots of optional parameters. The most common one is methods, which tells the server what kinds of requests the end point should handle. Typically get requests are for getting html pages or other kinds of data from the server, whereas post requests are for sending data to the server, as is the case with forms. (You may remember this type of discussion from chapter 1).

Let's do something slightly more complex now. Doing so will require a few files and some folders. Here is our file structure:

web*app/ run.py app/ _init*.py server.py templates/ index.html

It may seem artificial to seperate our application into all these different pieces, but this structure will be very useful once we start using blueprints - a pattern that lets us split up pieces of the web application into different application contexts. This allows for faster development and better organizational structure at scale. It also leads us well to understanding the structure of the django framework, which we'll get into soon!

run.py:

```
from app import app

app.run(debug=True)
```

init.py

```
from flask import Flask

app = Flask(__name__)

from app import server
```

Basically, **init**.py handles the instantiation of our application context and run.py handles actually running the server.

The server.py file is where all our routes are going to live for now. This application will still be very simple - its simply a form that takes in a name and email address and then displays the pair in a list below the form. While this may be a trival example it illustrates two major tasks in web development - getting data from the server and sending data to the server. Another important aspect of this application, is the data is persistent, because we make use of a simple database.

So let's take a look at server.py:

```
from app import app
from flask import render_template,request,jsonify,redirect,url_for
from flask.ext.sqlalchemy import SQLAlchemy

###Models

app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///database.db"
db = SQLAlchemy(app)

class Store(db.Model):
    __tablename__ = 'store'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(400),nullable=False)
    email = db.Column(db.String(400),nullable=False)

    def __init__(self,name,email):
        self.name = name
        self.email = email

    def __repr__(self):
        return  "<store %r>" % self.name

###Controller

@app.route("/",methods=["GET","POST"])
def index():
    return render_template("index.html",data=Store.query.all())

@app.route("/add_data",methods=["POST"])
def add():
    print "posted data to the server"
    name = request.form.get("name")
    email = request.form.get("email")
    store = Store(name,email)
    db.session.add(store)
    db.session.commit()
    print store
    return redirect(url_for("index"))
```

Here we've split up our concerns into model and controller - this idea of splitting up concerns in this way comes from the model,view,controller pattern. The view logic lives in the html file, which we will see in a few moments. The model concerns model our data, telling our database what kind of data should be stored, as well as how that data should be represented. We express storage via:

```
__tablename__ = 'store'
```

```
id = db.Column(db.Integer, primary_key=True)
name = db.Column(db.String(400),nullable=False)
email = db.Column(db.String(400),nullable=False)

def __init__(self,name,email):
    self.name = name
    self.email = email
```

We model our data as sqlalchemy's general purpose Object Relational Model. The individual fields are modeled via db.Column. We then inform the database what to do on instantiate of our record object - with the **init** method. Notice that we don't need to pass an id, it is generated automatically. The id is good practice and is useful for indexing data, allowing for faster queries, and database tunning generally.

Now we can explore the other part of our database class:

```
def __repr__(self):
    return  "<store %r>" % self.name
```

Here we make use of the **repr** method which will determine the representation of individual objects of our database, assuming we want high level information about the objects. We can also access individual fields from our objects in the following way:

testing_database.py:

```
from app.server import Store

store = Store.query.all()[0] #get any old object.
print store.name
print store.email
```

Notice that we can access the individual fields, and so, **repr** is more of a notational convenience more than anything else. This way we can know what object we are dealing with, without having to deal with accessors.

Now let's look at the controller:

```
@app.route("/",methods=["GET","POST"])
def index():
    return render_template("index.html",data=Store.query.all())

@app.route("/add_data",methods=["POST"])
def add():
    print "posted data to the server"
    name = request.form.get("name")
    email = request.form.get("email")
    store = Store(name,email)
    db.session.add(store)
    db.session.commit()
    print store
    return redirect(url_for("index"))
```

Here we don't have a ton of new stuff. Notice that we only expose a `POST` method for our add_route, which will be used for submitting our form data. Notice that I name the route the same as the method, this is good practice, as we will see, for `url_for`, which resolves method names to their url.

At this point, it makes sense to bring in the html to understand the interaction between the view and the controller:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>

<form method="post" action="">
<label>Name:</label><input type=text name=name>
<label>Email:</label><input type=text name=email>
<input type=submit>
</form>

<ul>

</ul>

</body>
</html>
```

So as you can see, the form action calls the add_data route, via url_for. Notice the use of openning - . This is what tells the jinga template engine to execute different python and flask methods. Typically it's for calling routes, rendering data, or resolving urls. Notice also that the name attribute is present in both of the input fields and these correspond to the `request.form.get`, which grabs the necessary input from the form. Finally, we can look back at our controller:

```
store = Store(name,email)
db.session.add(store)
db.session.commit()
```

These three lines are what create our Store object, with our name and email and then save these values to our database. Notice there was no specific connection made here, nor any database specific code. Meaning, we can safely swap databases by changing a single line of code:

`app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///database.db"`

The two recommended databases are postgres and sqlite - postgres for production and sqlite for local testing, however SQLAlchemy supports a plethora of datastores.

Next, let's look at this part of the html:

```
<ul>

</ul>
```

Here we see a dynamically generated list, that grows with the size of our data list. Notice that we can access specific data from our datum, since data is a list of type Store. The is used to evaluate individual datum. Notice also that we need an `endfor` statement, ending our for-loop.

This html corresponds to the index method:

```
@app.route("/",methods=["GET","POST"])
def index():
    return render_template("index.html",data=Store.query.all())
```

Notice that data is passed as an optional parameter to the `render_template` function, which is why the list is called data in the html. Whatever we name the optional parameter, will be the name of that parameter on the front end. The render_template function accepts all native python data structures - variables, lists, and dictionaries. In this case, we pass a list, which is all the objects in the Store table. Notice that we chose to pass all, but we could also have filtered on certain parameters. To filter a query with sqlalchemy objects we simply need to do something like the following:

`print [obj for obj in Store.query.filter(Store.name == 'Eric')]`

As you can see, the filter expects anything that would satisfy a WHERE clause in SQL. Most of the time I just work with basic booleans like equality, less than or greater than, but other more complex query terms are possible.

C3.js, D3.js and Vincent

We now have all the necessary rigging to work with C3.js or D3.js, vincent, or any other data visualization package we could want. Let's start with C3.js since it is very easy.

C3.js and flask

Now that we understand how flask works, we are ready to jump into C3.js! With C3 we can make beautiful dynamic charts and graphs with just a few lines of code. There isn't too much new in the server below, since that we are generating random lists of integers to be visualized.

server.py:

```
from flask import Flask,render_template
import json
from random import randint
app = Flask(__name__)

@app.route("/",methods=["GET","POST"])
def index():
    data1 = ['data1']
    data2 = ['data2']
    for i in xrange(0,randint(0,15)):
        data1.append(randint(0,150))
    for i in xrange(0,randint(2,25)):
        data2.append(randint(7,450))
    return render_template("line_chart.html",data1=json.dumps(data1),data2=json.dump


app.run(debug=True)
```

Here is where all the magic happens, specifically within the script tags:
line_chart.html:

```
<html>
    <head>
        <title>Data Viz example</title>

        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/c3/0.4.1
        <script src="https://cdnjs.cloudflare.com/ajax/libs/c3/0.4.10/c3.min.js"></s
        <script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.6/d3.min.js"></sc
    </head>
    <body>

    <div id="chart"></div>

    <script>
        var data1 = JSON.parse();
        var data2 = JSON.parse();
        var data3 = JSON.parse();
        var chart = c3.generate({
            bindto: '#chart',
            data: {
                columns: [
                    data1,
                    data2
                ]
            }
        });
    </script>

</body>
</html>
```

Notice that we need to create a div tag with an id that is the same as the bindto parameter in our `c3.generate` function. In this case we chose to name our id chart. Notice the use of the # infront of chart. Notice also that we create a variable called chart for this chart that will be generated. Notice also that adding or data to our graph is extremely easy, we just need to include our columns. Since the data is being loaded from the server, it might not be clear what these columns of data look like. So let's be explicit:

`data1` will be a list, where the first element (at index 0) is the name of the data set, in this case, data1. This is stored as a string. The rest of the elements are technically optional and can be an integer or a float. So data1 might look like this:

`['data1', 4,17,25,47]`

The same pattern will hold for data2 and any other columns you may want to add.

With c3 we can make lots of types of charts very, very easily!

You can find more examples and the complete server for splines, bar charts, and pie charts in the github **here**.

Free GIS Systems, in Python

Now that we've learned how to use flask and do some web development, we'll turn out attention to something somewhat different: GeoDjango. GeoDjango is the Django MVC framework, with added support for Geographic Information Systems. A GIS system is a visual way of representing data on a physical map. The most well known example of this is google maps. With google maps you can embed nearly limitless amounts of data and make that data intuitive, easy to understand and interpret, and also make the data useful. That's really the goal of all of data science, and certainly the goal of this book. A good GIS system aims to make data obvious and clear so human connections can be made easily and without any technical ability or understanding.

So enough chatting about how awesome this stuff is, let's do some of it! Fortunately Django is one of the most mature and well documented web frameworks around. So I'll just reference **that documentation** so you can get up to speed with Django before moving onto GeoDjango. Hopefull you went through the example application at this point and are now ready for the magic that is GeoDjango.

Installation - GeoDjango

First we'll install Geodjango, which luckily comes with Django1.8. So I'll you'll need to do really is install django, instructions on how to do that can be found **here**, assuming you want the source or anything else.

But really all you need to do is: `sudo pip install django`

Before we can get started with building we'll need to install PostGIS, steps on how to do that can be found **here**

Next we'll need to install the postgres python module - pyscopg2: `sudo pip install pyscopg2`

Next all we need to do is set up GeoDjango to work with PostGIS, and we are off to the races.

Unfortunately the GeoDjango documentation for the first example is confusion and leaves out a few steps, so first we'll have to correct that:

Please note these directions are for Ubuntu 14.04

Installation:

Python: `sudo pip install django #make sure you are installing django 1.8`

`sudo pip install pyscopg2`

PostGres:

`sudo apt-get update`

`sudo apt-get install -y postgresql postgresql-contrib`

Testing postgres install:

`sudo -u postgres createuser -P USER_NAME_HERE`

`sudo -u postgres createdb -O USER_NAME_HERE DATABASE_NAME_HERE`

`psql -h localhost -U USER_NAME_HERE DATABASE_NAME_HERE`

Adding PostGIS support:

`sudo apt-get install -y postgis postgresql-9.3-postgis-2.1`

`sudo -u postgres psql -c "CREATE EXTENSION postgis; CREATE EXTENSION postgis_topology;" DATABASE_NAME_HERE`

changing everything to trusted, rather than requiring authentication - DO THIS FOR LOCAL DEVELOPMENT ONLY!!!

`sudo emacs /etc/postgresql/9.1/main/pg_hba.conf`

Change line:

```
local all postgres peer
```

To

```
local all postgres trust
```

Then restart postgres:

```
sudo service postgresql restart
```

Getting started with geodjango:

Now we are ready to get started:

```
django-admin startproject geodjango
cd geodjango
python manage.py startapp world
```

Now we'll go into the settings.py file:

```
emacs geodjango/settings.py
```

and edit the databases connection to look like this:

```
DATABASES = {
    'default': {
         'ENGINE': 'django.contrib.gis.db.backends.postgis',
         'NAME': 'geodjango',
         'USER': 'geo',
     }
}
```

Notice that we haven't created the 'geodjango'-database so we'll do that now:

```
sudo -u postgres createuser -P geo
sudo -u postgres createdb -O geo geodjango
sudo -u postgres psql -c "CREATE EXTENSION postgis; CREATE EXTENSION postgis_topology;" geodjango
```

we'll also need to edit the installed aps, in the same file:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.gis',
    'world'
)
```

Great, now we can save and close that.

Next we'll need some data to visualize:

```
mkdir world/data
```

`cd world/data`

`wget http://thematicmapping.org/downloads/TM_WORLD_BORDERS-0.3.zip`

`unzip TM_WORLD_BORDERS-0.3.zip`

`cd ../..`

Now let's inspect our data so we now how our model should look - we should try to be consistent with how the data is annotated for portability and extensibility.

For this we'll need gdal - `sudo apt-get install libgdal-dev python-gdal gdal-bin # the python library is unnecessary but nice to have :)`

Now we can inspect the annotation in the shapefile of our geospatial data:

`ogrinfo -so world/data/TM_WORLD_BORDERS-0.3.shp TM_WORLD_BORDERS-0.3`

We'll use this output to map to our models.py file:

`emacs world/models.py` and type:

```
from django.contrib.gis.db import models

class WorldBorder(models.Model):
    # Regular Django fields corresponding to the attributes in the
    # world borders shapefile.
    name = models.CharField(max_length=50)
    area = models.IntegerField()
    pop2005 = models.IntegerField('Population 2005')
    fips = models.CharField('FIPS Code', max_length=2)
    iso2 = models.CharField('2 Digit ISO', max_length=2)
    iso3 = models.CharField('3 Digit ISO', max_length=3)
    un = models.IntegerField('United Nations Code')
    region = models.IntegerField('Region Code')
    subregion = models.IntegerField('Sub-Region Code')
    lon = models.FloatField()
    lat = models.FloatField()

    # GeoDjango-specific: a geometry field (MultiPolygonField), and
    # overriding the default manager with a GeoManager instance.
    mpoly = models.MultiPolygonField()
    objects = models.GeoManager()

    # Returns the string representation of the model.
    def __str__(self):              # __unicode__ on Python 2
        return self.name
```

We are now ready to run our first migration :)

`python manage.py makemigrations`

`python manage.py sqlmigreate world 0001`

`python manage.py migrate`

Making our first map:

Now that we've done all the setup, we can leverage geodjango immmediately to create an interactive map!

We'll need to make a few small edits to some files, but essetentially, we are done:

open the admin.py file and type the following:

emacs world/admin.py:

```
from django.contrib.gis import admin
from models import WorldBorder

admin.site.register(WorldBorder, admin.GeoModelAdmin)
```

Finally we simply need create our admin credentials:

`python manage.py createsuperuser` `python manage.py runserver`

Now head over to **http://127.0.0.1:8000/admin/**

and enter your credentials. Now you have a GIS system that you can play with, load data into, and get analysis out of!

Chapter 4

Searching across massive data sets

- facial recognition, facial comparison, and image search - building a cbir - easy
- searching across text in mass - easy
- multithreading applications for faster processing - easy
- using Hadoop and spark for search - easy

Facial recognition, facial comparison, and image search

An important innovation in the world of computers has been search. We've more or less figured search out, and for the rest of the world, with databases, this is very useful, but for most folks working in government, things are still stored in a file system. If you want to do file search efficiently, just use **Solr**.

However, after text based documents, the second biggest need for search is images. There are some standard tools out there, but it's actually worth it to play around with your own solution. Sometimes you'll need some custom stuff, sometimes you'll need to something special, like my boss asked me to do - image recognition and search off of one picture. I had to get creative since usually results are not that good for a single picture against other pictures, which are not the same shot. To be clear my task was this:

If someone was in a picture, find them in all other pictures they appear in.

No easy feat. But I got pretty close. I ended up settling on two techniques/ tools - OpenCVs face comparison algorithms and a wonderful post by the writer of **pyimagesearch**. If you haven't seen it, I recommend checking out that blog, it's full of great posts about computer vision.

So face comparison searches and compares faces against each other, producing a distance metric, the smaller the distance between two faces, IE the smaller the output, the more likely they are the same person's face. For pyimagesearch, I made use of their **backgroud comparison post**. In this post Adrian lays out how to compare the backgrounds of two images - by transforming the pictures into histograms, using a visual bag of words, and then making use of a chi^2 distance, to build an index. Finally, we search across the index to look for a picture or a similar picture in our set of pictures. From this we can see if the face and background match for a particular picture - if they do, it's probably the same person. But each of these metrics own their own is powerful for investigations. Then we can see who else has been in a the same room, and thus who else knows each other. We can also see this if we can find multiple people in the same picture.

One note - we could have used Caffe for the background image search. The trouble with Caffe is, I find it extremely difficult to install. Also, I'm not terribly good at C++, which is a major drawback. The following code is written entirely in python (or wrappers in python). Caffe does have python wrappers for some of its stuff, and it is a wonderful library if you have the computational power to truly leverage it, but most of the time in government you don't have the ability to install whatever you want and you definitely don't have the computational power to leverage such a tool.

Background compare:

The complete code with data can be found **here** `index_search.py`:

```
import numpy as np
import cv2
import argparse
import csv
from glob import glob


args = argparse.ArgumentParser()
args.add_argument("-d","--dataset",help="Path to the directory that contains the ima
args.add_argument("-i","--index",help="Path to where the computed index will be stor
args.add_argument("-q","--query",help="Path to the query image")
args.add_argument("-r","--result-path",help="Path to the result path")
```

```
args = vars(args.parse_args())

class ColorDescriptor:
    def __init__(self,bins):
        self.bins = bins

    #off_center means that we expect no people in the center of the picture
    #this is the feature construction and processing function
    #cX,cY stand for center X and center Y, aka the middle of the picture
    #off_center creates four quadrants for the picture
    #full_picture checks the full picture and only creates one histogram per picture
    #face_compare creates a segmentation around possible faces, and uses only the face to crea
    def describe(self,image,off_center=False,full_picture=False,face_compare=False):
        image = cv2.cvtColor(image,cv2.COLOR_RGB2HSV)
        features = []
        (h,w) = image.shape[:2]
        (cX,cY) = (int(w*0.5), int(h*0.5))
        segments = [(0,cX,0,cY),(cX,w,0,cY),(cX,w,cY,h),(0,cX,cY,h)]
        if off_center:
            for (startX,endX,startY,endY) in segments:
                cornerMask = np.zeros(image.shape[:2],dtype="uint8")
                cv2.rectangle(cornerMask,(startX,startY),(endX,endY),255,-1)
                hist = self.histogram(image,cornerMask)
                features.extend(hist)
        elif full_picture:
            hist = self.histogram(image,None)
            features.extend(hist)
        elif face_compare:
            cascPath = "haarcascade_frontalface_default.xml"
            faceCascade = cv2.CascadeClassifier(cascPath)
            #gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
            # Detect faces in the image
            faces = faceCascade.detectMultiScale(
                image,
                scaleFactor=1.1,
                minNeighbors=2,
                minSize=(25, 25),
                flags = cv2.cv.CV_HAAR_SCALE_IMAGE
            )

            for (x,y,w,h) in faces:
                cornerMask = np.zeros(image.shape[:2],dtype="uint8")
                cv2.rectangle(cornerMask, (x,y), (x+w, y+h), 255, -1)
                hist = self.histogram(image,cornerMask)
                features.extend(hist)
        else:
            ellipMask = np.zeros(image.shape[:2],dtype="uint8")
            (axesX, axesY) = (int(w * 0.75) / 2, int(h * 0.75) / 2)
            cv2.ellipse(ellipMask, (cX, cY), (axesX,axesY), 0,0,360,255,-1)
            for (startX,endX,startY,endY) in segments:
                cornerMask = np.zeros(image.shape[:2],dtype="uint8")
                cv2.rectangle(cornerMask,(startX,startY),(endX,endY),255,-1)
                cornerMask = cv2.subtract(cornerMask,ellipMask)
                hist = self.histogram(image,cornerMask)
                features.extend(hist)
            return features

    def histogram(self,image,mask):
        hist = cv2.calcHist([image],[0,1,2],mask,self.bins,
        [0, 180, 0, 256, 0, 256])
        hist2 = cv2.normalize(hist,np.zeros(image.shape[:2],dtype="uint8")).flatten
        return hist2

class Searcher:
    def __init__ (self,indexPath):
        self.indexPath = indexPath
    def search(self, queryFeatures, limit=10):
        results = {}
        with open(self.indexPath) as f:
            reader = csv.reader(f)
            try:
                for row in reader:
                    features = [float(x) for x in row[1:] if x!= '']
                    d = self.chi2_distance(features, queryFeatures)
                    results[row[0]] = d
            except IndexError:
                pass
        results = sorted([(v,k) for (k,v) in results.items()])
        return results[:limit]
    def chi2_distance(self,histA,histB, eps = 1e-10):
        d = 0.5 * np.sum([((a-b)**2) / (a+b+eps)
                         for (a,b) in zip(histA,histB)])
        return d

if __name__ == "__main__":
    #i have no idea where these numbers came from.. check - http://www.pyimagesearch
    cd = ColorDescriptor((8,12,3))

    if args["index"] and args["dataset"]:
        with open(args["index"],"w") as output:
            for img_type in [".png",".jpg",".PNG",".JPG",".img",".jpeg",".jif",".jf
                for imagePath in glob(args["dataset"] + "/*" + img_type):
                    imageID = imagePath[imagePath.rfind("/") + 1:]
                    image = cv2.imread(imagePath)
                    features = cd.describe(image)
                    features = [str(f) for f in features]
                    output.write("%s,%s\n" % (imageID, ",".join(features)))
    if args["index"] and args["query"]:
        query = cv2.imread(args["query"])
        features = cd.describe(query)
        searcher = Searcher(args["index"])
        results = searcher.search(features,limit=4)

        cv2.imshow("query",query)
        for (score,resultID) in results:
            result = cv2.imread(args["result_path"] + "/" + resultID)
            cv2.imshow("Result",result)
            cv2.waitKey(0)
```

Rather than explaining all this code, because Adrian explains all of it and extremely well, I'll simply explain how to use it:

`python index_search.py -d [directory of pictures] -i [name of file] #create an index`

`python index_search.py -i [name of index file] -q [path to query picture] -r [path to picture directory] #search for a picture`

A specific example:

`python index_search.py -d pic_db/ -i index.csv`

`python index_search.py -i index.csv -q person.jpg -r pic_db/`

So here's how the code works:

First let's look at the main method:

```
cd = ColorDescriptor((8,12,3))

if args["index"] and args["dataset"]:
    with open(args["index"],"w") as output:
        for img_type in [".png",".jpg",".PNG",".JPG",".img",".jpeg",".jif",".jfif",".jp2",".t
            for imagePath in glob(args["dataset"] + "/*" + img_type):
                imageID = imagePath[imagePath.rfind("/") + 1:]
                image = cv2.imread(imagePath)
                features = cd.describe(image)
                features = [str(f) for f in features]
                output.write("%s,%s\n" % (imageID, ",".join(features)))
if args["index"] and args["query"]:
    query = cv2.imread(args["query"])
    features = cd.describe(query)
    searcher = Searcher(args["index"])
    results = searcher.search(features,limit=4)

    cv2.imshow("query",query)
    for (score,resultID) in results:
        result = cv2.imread(args["result_path"] + "/" + resultID)
        cv2.imshow("Result",result)
        cv2.waitKey(0)
```

As you can see there isn't a ton going on here, we loop through all the images in the pictures folder, doing the feature transformations on each picture and then generating an index with all the feature-histogram-vectors. Now let's dig a little deeper into the describe method. The describe method acts on the matrix representation of an image. The image is a matrix at this point, because it was read in with open cv's imread method. So really the describe method is a matrix transformation, which produces features according to some rules about the image. The way in which you decide to describe images, turns out to be very very important. Essentially the describe is doing a segmentation and then a transformation.

One fun thing I did was, I allowed you to segment pictures by face, into four corners, or the way Adrian does it; with four corners and an elliptical in the center. My ways don't always yield good results, and in fact more than half the time Adrians method works the best, but sometimes, my methods have been effective so I will mention them here.

To understand that we'll look at the segmentation methods one at a time:

```
for (startX,endX,startY,endY) in segments:
    cornerMask = np.zeros(image.shape[:2],dtype="uint8")
    cv2.rectangle(cornerMask,(startX,startY),(endX,endY),255,-1)
    hist = self.histogram(image,cornerMask)
    features.extend(hist)
```

Here we setup a cornerMask which is a matrix of zeroes. Notice that we apply the rectangle method, which takes in starting and finishing coordinates and then 'draws' a rectangle on the cornerMask. Then we create a histogram with the segmentation of the image defined by the rectangle we 'drew' which acts as sort of a boundary. Depending on how you draw your segmentation, will effect your histogram and thus your features.

The full picture is uninteresting, because it is just the histogram transformation with no segmentation.

The next interesting segmentation is one done with semantic meaning built in - finding the startX,startY and endX,endY from the faces in the picture:

```
cascPath = "haarcascade_frontalface_default.xml"
faceCascade = cv2.CascadeClassifier(cascPath)
#gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Detect faces in the image
faces = faceCascade.detectMultiScale(
    image,
    scaleFactor=1.1,
    minNeighbors=2,
    minSize=(25, 25),
    flags = cv2.cv.CV_HAAR_SCALE_IMAGE
)
```

```
for (x,y,w,h) in faces:
    cornerMask = np.zeros(image.shape[:2],dtype="uint8")
    cv2.rectangle(cornerMask, (x,y), (x+w, y+h), 255, -1)
    hist = self.histogram(image,cornerMask)
    features.extend(hist)
```

Here we use opencv's face detection classifier to find all the 'boxes' around the potential faces in the image. This is then used to create our rectangles, simiarly to the previous example. However, now rather than segmenting on corners, we segment on faces. We then use this segmentation to create histograms and finally feature vectors of just the faces. Unfortunately, in practice this technique tends to perform the worst, for now. I need to figure out how to better tune my parameters so that this is reliable, or I may abandon it as an idea. But still, it was fun to try!

The final method is what Adrian did:

```
ellipMask = np.zeros(image.shape[:2],dtype="uint8")
(axesX, axesY) = (int(w * 0.75) / 2, int(h * 0.75) / 2)
cv2.ellipse(ellipMask, (cX, cY), (axesX,axesY), 0,0,360,255,-1)
for (startX,endX,startY,endY) in segments:
    cornerMask = np.zeros(image.shape[:2],dtype="uint8")
    cv2.rectangle(cornerMask,(startX,startY),(endX,endY),255,-1)
    cornerMask = cv2.subtract(cornerMask,ellipMask)
    hist = self.histogram(image,cornerMask)
    features.extend(hist)
```

Creating an elliptical mask for the center of the picture and generating segments for the remaining four corners and then subtracting the center from the four corners. Nothing else is new here and thus it can largely be ignored.

The next important piece of the main function is performing a query which is captured here:

```
query = cv2.imread(args["query"])
features = cd.describe(query)
searcher = Searcher(args["index"])
results = searcher.search(features,limit=4)
```

Notice that we describe the query image the same way we described the index. This is imperative - if we describe our feature vectors differently in indexing and querying, our queries won't work.

The search function is fairly simple:

```
results = {}
with open(self.indexPath) as f:
    reader = csv.reader(f)
```

```
        try:
            for row in reader:
                features = [float(x) for x in row[1:] if x!= '']
                d = self.chi2_distance(features, queryFeatures)
                results[row[0]] = d
        except IndexError:
            pass
    results = sorted([(v,k) for (k,v) in results.items()])
    return results[:limit]
```

Essentially, it applies the chi^2 distance function to each of the features in the index against the query image's features. The results are returned in sorted order, lowest to highest. This is because if the distance is smallest, they are the images can be expected to be the least different.

Understanding Face Comparison, With OpenCV

The `face_compare2.py` contains how I do face comparison with open CV. It works quiet well, but I do confess it is not my most well organized piece of code. I have not yet learned how to be elegant with image manipulation, alas, the struggles of perfection in a sea of desires and things one aspires to be perfect at. But such is the calamity of curious minds, and it is something I am proud of, to be pulled in so many possible directions.

Note: For this code to work from this repo, you'll need to install OpenCV from the follow githubs: **opencv**, **opencv-contrib**, and it's not a bad idea to install **opencv-extra**, but only if you have space for the extra data, it's not necessary.

And the code that I'll reference throughout this section can be found **here**

In any event, there is a lot to unpack in that file and so I'm simply going to explain some high level steps:

First we instantiate our recognizers:

```
recog = {}
recog["eigen"] = cv2.face.createEigenFaceRecognizer()
recog["fisher"] = cv2.face.createFisherFaceRecognizer()
recog["lbph"] = cv2.face.createLBPHFaceRecognizer()
```

Notice that recently OpenCV has changed and now the face recognizers live in a seperate supplemental library. The core of OpenCV can be found here: **https://github.com/Itseez/opencv** and the face recognizers can be found here: **https://github.com/Itseez/opencv_contrib**. I'm not sure why they decided to move such a handy set of tools out into some contrib repo, but that is how it is.

The next thing to do is normalize the pictures:

normalize_cv(filename,compare,directory_of_pictures)

In this case, this means making them all the same height and width, because the face comparison algorithms require this. I'm not sure why this is the case.

Next we read in a picture and hand label it 1:

```
face = cv2.imread(new_filename,0)
face,label = face[:, :], 1
```

And we read in a different picture with a different face and hand label it 2:

```
compare_face = cv2.imread(new_compare, 0)
compare_face, compare_label = compare_face[:,:], 2
```

This establishes the base comparison for what will be considered a face we are searching for and a face that is different from the one we want. Perhaps using 2 isn't the best, because the distance isn't great enough, but this is intended to be a toy example, you should feel free to tune these "magic" numbers to improve the precision of your own code.

Next we train the each recognizer on the training data:

```
for recognizer in recog.keys():
    recog[recognizer].train(image_array,label_array)
```

And then we compare all the trained data against the directory of pictures:

```
test_images = [(np.asarray(cv2.imread(img,0)[:,:]),img) for img in test_images]
possible_matches = []
for t_face,name in test_images:
    t_labels = []
    for recognizer in recog.keys():
        try:
            [label, confidence] = recog[recognizer].predict(t_face)
            possible_matches.append({"name":name,"confidence":confidence,"recognizer":recogni
```

Notice that we simply need to call the predict function on each image in our directory.

Finally, we simply can print out all the possible matches:

```
for i in possible_matches:
    print i
```

Running this code is fairly simple and can be accomplished with the following:

`python face_compare2.py [first picture] [baseline comparison picture] [directory of pictures to search against]`

Searching in Mass with Solr

Installation

I'll borrow heavily from these installation instructions found **here** They detail how to install solr for Java8 with python support on ubuntu 14.04. After that we'll look at django integration and flask integration.

Basic django integration - **https://www.goldmund-wyldebeast-wunderliebe.nl/tech-blog/blog-posts/introduction-to-solr** Basic flask integration - **https://github.com/adsabs/flask-solrquery**, **https://github.com/willowtreeapps/flask-solr**

Multithreading applications

While Python has a Global Interpretter Lock (GIL), meaning it cannot be truly multithreaded, there are a lot of things you can still do that look and feel like multithreading. And you can absolutely parallelize your code. For more on the GIL check out these sources:

- **PythonWiki**
- **These awesome slides from David Beazley**
- **This sweet general purpose wikipedia article on GILs**

If you are new to doing multiple things in one program, I'll just give you the two second crash course. Feel free to skip ahead if this is review.

What's all this then?

When you multithread an application the general idea is that you'll have multiple threads executing different parts of your code independent of one another, but that all of these independent threads will at some point depend on each other again. This is known as blocking code, because threads cannot continue until other threads complete. So let's look at a very dumb and simple example.

Note: Two of the following examples come from **this great stackoverflow thread** - summing.py and scraping.py

summing.py:

```
import threading

class SummingThread(threading.Thread):
     def __init__(self,low,high):
         super(SummingThread, self).__init__()
```

```
        self.low=low
        self.high=high
        self.total=0

    def run(self):
        for i in range(self.low,self.high):
            self.total+=i


thread1 = SummingThread(0,500000)
thread2 = SummingThread(500000,1000000)
thread1.start() # This actually causes the thread to run
thread2.start()
thread1.join()  # This waits until the thread has completed
thread2.join()
# At this point, both threads have completed
result = thread1.total + thread2.total
print result
```

In this example we create multiple threads to sum different pieces of the numbers 0 to 1000000. Notice that in this example the code cannot move onto the result until both threads 1 and 2 complete their sums over the total range. This example shows us one way of creating threads and making use of them to get stuff differently. Now let's compare this with the sequential running example.

summing_time_test.py

```
import threading
from time import time

class SummingThread(threading.Thread):
    def __init__(self,low,high):
        super(SummingThread, self).__init__()
        self.low=low
        self.high=high
        self.total=0

    def run(self):
        for i in range(self.low,self.high):
            self.total+=i

def summing():
    thread1 = SummingThread(0,500000)
    thread2 = SummingThread(500000,1000000)
    thread1.start() # This actually causes the thread to run
    thread2.start()
    thread1.join()  # This waits until the thread has completed
    thread2.join()
    # At this point, both threads have completed
    return thread1.total + thread2.total

def regular_sum():
    summa = 0
    for i in xrange(1000000):
        summa += i
    return summa

summing_start = time()
summing()
print "Summing Total Time:", time()-summing_start

regular_sum_start = time()
regular_sum()
print "Regular Total Time:", time() - regular_sum_start
```

You can find this code **here** and experiment yourself :)

As you'll see, if you try this code, the sequential solution actually performs an order of magnititude faster, every time. So why would you ever do this? Well, we looked at sort of a toy example to understand how blocking could happen. It's important to note that in this example every single input ran in the same amount of time, a bad candidate for mutlithreading. If there are some inputs where computation takes significantly longer, than there will be a vast speed up from doing things out of order. Specifically, processing the quickly evalutated inputs at the same time as the ones that will take longer. Of course, tuning this can be tricky because you may not always know what will be a problem until your code is already running. Of course, we can always create a situation where we know certain inputs will be computationally more intensive then others. The classical choice for this is computing the fibonacci numbers recursively.

```
import threading
from time import time

def fib(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

class SummingThread(threading.Thread):
     def __init__(self,low,high):
         super(SummingThread, self).__init__()
         self.low=low
         self.high=high
        self.total=0

    def run(self):
        for i in range(self.low,self.high):
            self.total+=fib(i)

def summing():
    thread1 = SummingThread(0,15)
    thread2 = SummingThread(15,30)
    thread1.start() # This actually causes the thread to run
    thread2.start()
    thread1.join()  # This waits until the thread has completed
    thread2.join()
    # At this point, both threads have completed
    return thread1.total + thread2.total

def regular_sum():
    summa = 0
    for i in range(30):
        summa += fib(i)
    return summa

summing_start = time()
summing()
print "Summing Total Time:", time()-summing_start

regular_sum_start = time()
regular_sum()
print "Regular Total Time:", time() - regular_sum_start
```

Result when taking the sum for the first 30 fibonacci numbers:

Summing Total Time: 1.10466504097 Regular Total Time: 1.83267211914

As you can see there is a slight improvement when using threading over iteration here. As you may have guessed this improvement increases with two threads as n get's bigger. Feel free to try some experiments on your own to verify this. It's important to note that the above result will not be exactly the same, even when you try running the above code. The only guarantee is that threading will necessarily run faster than just straight iteration.

Now let's look at a slightly more practical example:

scraping.py:

```
import Queue
import threading
import urllib2

# called by each thread
def get_url(q, url):
    q.put(urllib2.urlopen(url).read())
```

```
theurls = ["http://google.com", "http://yahoo.com"]

q = Queue.Queue()

for u in theurls:
    t = threading.Thread(target=get_url, args = (q,u))
    t.daemon = True
    t.start()

s = q.get()
print s
```

This is an example of some code that is not blocking. This means threads can execute simultaneously and don't have to wait for each other to move onto other tasks. Notice the use of a queue here, which allows us to store our results in one structure, without knowing (or caring) what order they entered the queue. The reason this is powerful, is let's say we need to scrape a website many times or we need to scrape a bunch of websites at the same time and join some data from all of them together (a task one needs to do often in the anti-human trafficking world), this will be much, much faster with threading, because you can process requests independently of each other. If you had to wait for each website to scraped before moving onto the next, it would invariably take much, much longer.

Now let's look at yet another way mutlithreading can be used:

installer.py:

```
from subprocess import call
from multiprocessing import Process
import argparse

def install(requirement,how_to_install,args):
    if args["left_split"]:
        requirement = requirement.split(args["left_split"])[0]
    elif args["right_split"]:
        requirement = requirement.split(args["right_split"])[1]
    if 'sudo' in how_to_install:
        how_to_install = how_to_install.replace("[PACKAGE]",requirement).split(" ")
    else:
        how_to_install = ["sudo"] + how_to_install.replace("[PACKAGE]",requirement)
    call(how_to_install)

args = argparse.ArgumentParser()
args.add_argument("-ls","--left-split",help="the package name is to the left of the
args.add_argument("-rs","--right-split",help="the package name is to the right of th

args = vars(args.parse_args())
```

```
with open("requirements.txt","r") as f:
    dependencies = f.read().split("\n")
    dependencies = [dep for dep in dependencies if dep != '']
with open("how_to_install.txt","r") as f:
    how_to_install = f.read().strip()

for dep in dependencies:
    p = Process(target=install,args=(dep,how_to_install,args,))
    p.run()
```

The above code can be found **here**.

This application is slightly more involved than the previous ones but the idea should be clear enough. We typically install things sequentially, this application allows us to install requirements in parallel. Notice here we make use of the `Process` object instead of the threading interface. The `Process` object is merely a notational convienence and works the same way as threading.Thread but is much easier to write down and doesn't require any business with classes, which is kind of wonderful :). You can pass in a queue with a `Process` object very easily as the following simple example shows:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()    # prints "[42, None, 'hello']"
    p.join()
```

This example was lifted directly from the **mutliprocessing docs**.

As you can see, we simply pass in a queue to the target function and pass in the results, rather than doing a return statement.

One quick final note about the little installer I wrote. This installer runs, MUCH faster than installing sequentially for most long install jobs. I found on average I speed up of 10 minutes for most small jobs and for very long ones a savings of up to an hour.

This installer illustrates the point about multithreading better than I could make it otherwise. Of course, this assumes the package manager in question does not have the ability to natively parallelize or multithread the installation process.

Chapter 5 - Bringing Together Disperate Data Sources 6

In the previous chapters I have tried to prepare you for building tools by showing you specific techniques and ideas. The intention of these building block chapters was to prepare you to build your own systems that help address societal issues. The issue I care most deeply about is ending slavery, so in this chapter I will present the design process and implementation of three systems I have built for that purpose. Each of these systems is still in development, because good software is never finished. And as the bad guys evolve, so must these systems. However much the details may change, the design and main idea of each system will likely be invariant. And thus we can think of each system as the realization of a discrete set of ideas, where the details shift.

Investa_gator

This system canonicalizes and automates the investigation process for finding a sex slave on the internet. The system has a few components, a web scraper, a natural language processing engine, and a database. There are a few other components still under very active development, so they cannot be called a part of the system yet. These components are a image search and facial recognition tool, a GIS system, and a set of realtime visualizations.

As you may have guessed one of the goals of this book was to give insight into how these newer components might look, but sadly they are still ideas that lack a concrete place within the system. So while describing the system, I will mention these newer features in passing and describe how they might be integrated into the platform, but give no rigorous explanation or provide any implementation details.

The full tool in its current implementation can be found **here**.

The Web Scraper

The webscraper I've written for this tool pulls content from backpage.com, a website that makes up a large part of the US and international market for commerical sex, a proportion of which is sex slavery. The design of the web scraper is a direct result of the user interaction with it. A user interacts with the web scraping piece of the tool in a few ways. Assuming that a user has a few ads that they know are connected to a human trafficking case, they can add those ads to investa_gators list of ads that are known instances of human trafficking. Therefore the scraper needs to be able to scrape individual advertisements.

Once all the ads we know are related to a human trafficking case have been uploaded, investa_gator goes to work scraping other ads on the site. It compares the ads we know are trafficking to all the ads it scrapes and when it finds a match, the match is appended to a database. Eventually an alert system will be added, emailing investigators when new ads are found. However this functionality doesn't exist presently.

Below is the essential components of the web scraper. The full scraper can be found **here**.

crawler.py/scrape method:

```
def scrape(self,links=[],ads=True,translator=False):
    responses = []
    values = {}
    data = []

    if ads:
        for link in links:
            r = requests.get(link)
            responses.append(r)
    else:
        for link in links:
            r = requests.get(link)
            text = unidecode(r.text)
            html = lxml.html.fromstring(text)

            links = html.xpath("//div[@class='cat']/a/@href")
            for link in links:
                if len(self.base_urls) > 1 or len(self.base_urls[0]) > 3:
                    time.sleep(random.randint(5,27))
                try:
                    responses.append(requests.get(link))
                    print link
                except requests.exceptions.ConnectionError:
                    print "hitting connection error"
                    continue
    for r in responses:
        text = r.text
        html = lxml.html.fromstring(text)
        values["title"] = html.xpath("//div[@id='postingTitle']/a/h1")[0].text_content()
        values["link"] = unidecode(r.url)
        values["new_keywords"] = []
        try:
            values["images"] = html.xpath("//img/@src")
        except IndexError:
            values["images"] = "weird index error"
        pre_decode_text = html.xpath("//div[@class='postingBody']")[0].text_content().replace
        values["text_body"] = pre_decode_text
        try:
            values["posted_at"] = html.xpath("//div[class='adInfo']")[0].text_content().repla
        except IndexError:
            values["posted_at"] = "not given"
        values["scraped_at"] = str(datetime.datetime.now())
        body_blob = TextBlob(values["text_body"])
        title_blob = TextBlob(values["title"])
        values["language"] = body_blob.detect_language() #requires the internet - makes use o
        values["polarity"] = body_blob.polarity
        values["subjectivity"] = body_blob.sentiment[1]
        if values["language"] != "en" and not translator:
            values["translated_body"] = body_blob.translate(from_lang="es")
            values["translated_title"] = title_blob.translate(from_lang="es")
        else:
            values["translated_body"] = "none"
            values["translated_title"] = "none"
        text_body = values["text_body"]
        title = values["title"]
        values["phone_numbers"] = self.phone_number_parse(values)
        data.append(values)

    return data
```

Notice there are two major steps in this method - the web scraping piece, requesting the content; and the html parsing piece, processing the content that was brought into memory.

Understanding the web scraping piece:

```
        if ads:
            for link in links:
                r = requests.get(link)
                responses.append(r)
        else:
            for link in links:
                r = requests.get(link)
                text = unidecode(r.text)
                html = lxml.html.fromstring(text)
                links = html.xpath("//div[@class='cat']/a/@href")
                for link in links:
                    if len(self.base_urls) > 1 or len(self.base_urls[0]) > 3:
                        time.sleep(random.randint(5,27))
                    try:
                        responses.append(requests.get(link))
                        print link
                    except requests.exceptions.ConnectionError:
                        print "hitting connection error"
                        continue
```

In order to understand this code, we'll need to understand the structure of backpage's escort ad section. The way this works is similar to craigslist: ads appear on a page with one line hyperlinked descriptions of the content of the ad. Thus there is a central list of all the ads of a specific content type. What we are doing above is allowing individual ads to be scraped and processed, assuming you are scraping an individual ad - like we do when an investigator uploads an individual ad. Otherwise, we are scraping all of backpage, which means we must parse the html page in question and download all the ads on a given page.

To find all the ads on a given page we simply look for a div tag, with class cat and then the a tag within the div. That is accomplished with this line:

`links = html.xpath("//div[@class='cat']/a/@href")`

Notice that we make use of `time.sleep`, this is because we don't want to make too many requests to backpage at once. If we do that we'll get blocked, so we sleep for a random amount of time. The reason we sleep for a random amount of time is because if you sleep the same amount of time everytime, backpage might figure this out and block any ip address that makes requests every X seconds. Most investigative units have access to rotating IP addresses, so backpage is unable to block by ip address, however they can still figure out the signature via timing.

The next piece of interest is the try,except block:

```
try:
    responses.append(requests.get(link))
    print link
except requests.exceptions.ConnectionError:
    print "hitting connection error"
    continue
```

We shouldn't expect the content backpage posts to still be there when we try to scrape it. This is because backpage often will take down ads that either violate it's internal rules about posting or for some other reason. So we can't assume any advertisement will actually exist when we try to scrape it, just because it existed before. It's important that this be handled in a try, except because we want the server to be able to continue to scrape ads, even if a given ad failed, because this does happen somewhat rarely. Also, it's often the case that ads taken down will be reposted and so its likely if we continue to scrape backpage, we'll eventually get a complete set of all ads we are interested in.

Now that we understand the scraping step, let's look at the processing step:

```
for r in responses:
    text = r.text
    html = lxml.html.fromstring(text)
    values["title"] = html.xpath("//div[@id='postingTitle']/a/h1")[0].text_content()
    values["link"] = unidecode(r.url)
    values["new_keywords"] = []
    try:
        values["images"] = html.xpath("//img/@src")
    except IndexError:
        values["images"] = "weird index error"
    pre_decode_text = html.xpath("//div[@class='postingBody']")[0].text_content().replace("\n
    values["text_body"] = pre_decode_text
    try:
        values["posted_at"] = html.xpath("//div[class='adInfo']")[0].text_content().replace("
    except IndexError:
        values["posted_at"] = "not given"
    values["scraped_at"] = str(datetime.datetime.now())
    body_blob = TextBlob(values["text_body"])
    title_blob = TextBlob(values["title"])
    values["language"] = body_blob.detect_language() #requires the internet - makes use of goo
    values["polarity"] = body_blob.polarity
    values["subjectivity"] = body_blob.sentiment[1]
    if values["language"] != "en" and not translator:
        values["translated_body"] = body_blob.translate(from_lang="es")
        values["translated_title"] = title_blob.translate(from_lang="es")
    else:
        values["translated_body"] = "none"
        values["translated_title"] = "none"
    text_body = values["text_body"]
    title = values["title"]
    values["phone_numbers"] = self.phone_number_parse(values)
    data.append(values)

return data
```

This piece is fairly straight forward - for each ad in our list of responses we turn the html into something we can run xpath queries on via: html = `lxml.html.fromstring(text)`. From there we simply include the information we care about:

- title - `values["title"] = html.xpath("//div[@id='postingTitle']/a/h1")[0].text_content()`
- link to original posting - `values["link"] = unidecode(r.url)`
- links to any images - `values["images"] = html.xpath("//img/@src")`
- the text content of the advertisement -

  ~~~
  ```
  pre_decode_text = html.xpath("//div[@class='postingBody']")[0].text_content(
  values["text_body"] = pre_decode_text
  ```
  ~~~

- the time the ad was scraped - `values["scraped_at"] = str(datetime.datetime.now())`
- the language of the ad - `values["language"] = body_blob.detect_language()`
- the polarity of the ad - `values["polarity"] = body_blob.polarity`
- the subjectivity score of the ad - `values["subjectivity"] = body_blob.sentiment[1]`
- any phone numbers in the ad - `values["phone_numbers"] = self.phone_number_parse(values)`

Most of this should be pretty straight forward, but a few of the nlp tasks probably deserve some further explanation.

The polarity and subjectivity scores serve as a signature for a given ad, so if the phone numbers or other attributes (which aren't currently being captured but will be in the future), aren't present it's still possible to say whether or not two ads might be related. However such a comparison is more tenious. Yet, it implies the possibility of two sets of posters at least being aware of each other. Of course, it's not enough to build a case, but is certainly useful for investigators to be aware of and gives you something to query against in the database in a broad sort of way.

It's worth noting that this was only a first pass at a document comparison metric. I have since developed a library for doing document similarity that will at some point be integrated into this tool. It can be found **here**.

Getting the Phone number

In the last section I mentioned the method `phone_number_parse`. While this method is innocent looking enough, it is actually the heart and soul of the tool and my biggest accomplishment.

There has been an increasingly escalting obfuscation war between traffickers and law enforcement to hide phone numbers in plain sight.

I516 ha7ve se7en o7bfu4sca2tio1n l0ike this.

And ob5fu1sixca7tion li7ke th7i7s ThErE as well, fiVe 1 2 4 my measurements are 32 34 32.

And even more weird cases.

So what the hell do you do, to get the phone number - the most important piece of the puzzle:

I've written three functions:

1) turn all the words into numbers.

```
def letter_to_number(self,text):
        text= text.upper()
        text = text.replace("ONE","1")
        text = text.replace("TWO","2")
        text = text.replace("THREE","3")
        text = text.replace("FOUR","4")
        text = text.replace("FIVE","5")
        text = text.replace("SIX","6")
        text = text.replace("SEVEN","7")
        text = text.replace("EIGHT","8")
        text = text.replace("NINE","9")
        text = text.replace("ZERO","0")
        return text
```

Since we should assume numbers can be written as words, and these numbers will be written in weird mixes of upper and lower case characters we simply get ride of this problem by captializing all the written out numbers in the text and then transform them to their numeric form.

2) parse the numbers from the text

```
def phone_number_parse(self,values):
    phone_numbers = []
    text = self.letter_to_number(values["text_body"])
    phone = []
    counter = 0
    found = False
    possible_numbers = []
    for ind,letter in enumerate(text):
        if letter.isdigit():
            phone.append(letter)
            found = True
        else:
            if found:
                counter += 1
            if counter > 15 and found:
                phone = []
                counter = 0
                found = False
    #country codes can be two,three digits
        if len(phone) == 10 and phone[0] != '1':
            possible_numbers.append(''.join(phone))
            phone = phone[1:]
        if len(phone) == 11 and phone[0] == '1':
            possible_numbers.append(''.join(phone))
            phone = phone[1:]
    for number in possible_numbers:
        if self.verify_phone_number(number):
            phone_numbers.append(number)
    return phone_numbers
```

Here we grab all the possible phone numbers. The idea is simple enough: we iterate through the text of the ad, appending all digits to a list. If the list reaches 10 numbers or 11 numbers we simply cut off the first number from the list, creating a sliding window. This gives us a set of possible phone numbers.

3) most importantly - verify your phone number is correct - thank you twilio! And thank you Rob Spectre, you wonderful human you.

```
def verify_phone_number(self,number):
    data = pickle.load(open("twilio.creds","r"))
    r = requests.get("http://lookups.twilio.com/v1/PhoneNumbers/"+number,auth=da
    if "status_code" in json.loads(r.content).keys():
        return False
    else:
        return True
```

The fine folks at twilio happened to write an api that will test to make sure you phone numbers are real. With the combination of this and the rest of the tool, I was able to make this tool truly powerful. In fact, this piece is so important, it doesn't even exist in the Memex project - the most comprehensive anti-human trafficking tool in the space.

Running an Investigation

Once we are reasonably convinced we know everything a manual search will yield we can run an investigation. This allows the investigators time to be spent on other tasks, while new advertisements will be logged - allowing us to completely understand the network of the traffickers.

```
from textblob.classifiers import NaiveBayesClassifier as NBC
from textblob.classifiers import DecisionTreeClassifier as DTC
from sklearn.metrics.pairwise import linear_kernel
from sklearn.feature_extraction.text import TfidfVectorizer

#..snip..

    def doc_comparison(new_document,doc_list):
        total = 0.0
        for doc in doc_list:
            total += consine_similarity(new_document,doc)[1]
        if total/len(doc_list) > 0.5: #play with this
            return "trafficking"
        else:
            return "not trafficking"

    def cosine_similarity(documentA,documentB):
        docs = [documentA,documentB]
        tfidf = TfidfVectorizer().fit_transform(docs)
        cosine_similarities = linear_kernel(tfidf[0:1], tfidf).flatten()
        return cosine_similarities


    def investigate(self):

        data = self.scrape(self.base_urls)
        train_crud = CRUD("sqlite:///database.db",Ads,"ads")
        #getting dummy data from http://www.dummytextgenerator.com/#jump
        dummy_crud = CRUD("sqlite:///database.db",TrainData,"training_data")
        train = train_crud.get_all()
        dummy = dummy_crud.get_all()
        t_docs = [elem.text for elem in train_crud.get_all()] #all documents with trafficking
        train = [(elem.text,"trafficking") for elem in train] + [(elem.text,"not trafficking")
        cls = []
        #make use of tdf-idf here
        cls.append(NBC(train))
        cls.append(DTC(train))
        for datum in data:
            for cl in cls:
                if cl.classify(datum["text_body"]) == "trafficking":
                    self.save_ads([datum])
            #so I don't have to eye ball things
            if doc_comparison(datum["text_body"],t_docs) == "trafficking":
                self.save_ads([datum])
        time.sleep(700) # wait ~ 12 minutes
        self.investigate() #this is an infinite loop, which I am okay with.
```

As you can see, the investigate function will take the ads already classified as trafficking and look for new ads that are similar enough to be considering trafficking as well. I'm making use of two classification schemes here - Naive Bayesian Classification and Tree Classification.

Understanding Naive Bayesian Classification

To understand a naive bayesian classifier it's best to understand the steps involved:

1) hand label a set of texts as certain mappings:

```
[ ("Hello there, I'm Eric","greeting"),
  ("Hi there, I'm Jane","greeting"),
  ("Hi, how are you?","greeting"),
  ("Hello","greeting"),
  ("I'm leaving now, Jane","parting"),
  ("Goodbye","parting"),
  ("parting is such sweet sore, but I'm sleepy, so get out.","parting")
]
```

2) Transform each word in the training set so that it can be described as a set of independent proabilities, that map to a given label:

In this case we split each piece of text by mapping the words to numbers:

```
"Hello there, I'm Eric" -> Freq(Hello) = 1,  Freq(there,) = 1,  Freq(I'm) = 1, Freq
prob(Hello) = 1/4, prob(there,) = 1/4, prob(I'm) = 1/4, prob(Eric) = 1/4
```

We then apply one of a set of functions (typically a **maximium likelihood estimator**) to these mathematical quantities and say this maps to the label. In this case "greeting"

So we can sort of think of this as:

```
transform = f("Hello there, I'm Eric")
g(transform) = "greeting"
```

3) Verification of our model to our liking

The next step is to provide a verification set that is hand labeled as well. The model we've constructed from our training set is applied to this verification set and then checked against the expected label. If the label and the prediction from the model match, we claim the model is correct for this result.

Once we have enough favorable results we are ready to make use of our text classification system for real world problems

Making use of Cosine Similarity (to be lazy)

As you can see, we make use of Tf-Idf to and document similarity to assess whether two pieces of text are similar, initially (and throughout). This avoids the problem of having to hand label results - but is somewhat of a hack for now. The main reason for doing this is Naive Bayesian Classification isn't useful without enough labeled data, but typically when a trafficking case comes in, we only get 4 or maybe 5 ads. Of course, the analysts can go back in later when more ads come in, but it's not always easy to find these new ads. This is because some ads get taken down, either by the owner or by backpage over time. Therefore, an analyst may miss a new ad pertaining to a specific case, and we cannot afford to let luck dictate evidence collection.

The larger point here is there needs to be some automatic way of adding in documents that are similar enough.

Caveat Worth Mentioning

This tool is in alpha and I'm adding new features regularly. The next step here is to make use of hardcore algorithms. My guess is I'll settle on SVMs but I haven't been playing around with this long enough to test which algorithm will be most performant. If you want to understand text classification with serious algorithms I'd highly recommend checking out:

- **This scikit learn example**
- **This stack overflow answer**

Analyzing Leads and Collecting Information for Prosecution

Most traffickers also have their own personal websites. Once we have enough personal information from scraping backpage, we can typically start to find these domains.

Scraping other websites - A generalized web scraper and Alert System.

Ad sites like backpage all function more or less the same way - they post a ton of ads per day and there are a lot of pages, so you need specialized scrapers. Fortunately the ads are all very regular, so you only need to write those scrapers once.

However, there are lots of little website that will sometimes be crucial to an investigation. So I wrote a little general purpose tool that forensicly maps and stores websites. It will also rescrape them regularly, as content is updated as often as once per day.

Alert System

Grabbing all the things

At the heart of this tool is a single function - mapper

```
    def mapper(self,url,depth,link_list):
        """Grabs all the links on a given set of pages, does this recursively."""
        if depth <= 0:
            return link_list
        links_on_page = self.link_grab(url)
        tmp = []
        for link in links_on_page:
            if not link in link_list:
                link_list.append(link)
                tmp = self.mapper(link,depth-1,link_list)
                for elem in tmp:
                    if not elem in link_list:
                        link_list.append(elem)
        return link_list
```

The most important piece of this tool is making recursive calls, at each depth call to the mapper.

`link_grab` grabs all the links from a html page and then heads to the next depth of the page. Notice that depth is decremented each time the mapper is called. Typically websites are 3 or 4 deep, meaning we can grab a lot of links, very quickly making use of this method.

Storing all the things

Once we have everything grabbed I store it by downloading the html and taking the SHA256 hash to ensure that the website is as we downloaded it, for legal proceedings.

Sending Alerts

Rather than paying for scriptable email support, a friend of mine found a way to script sending and receiving emails via gmail with python. Thanks **Bryan Britten**

It is HIGHLY recommended that you create a new gmail address if you want to be able to script it, because you need to do the following:

https://www.google.com/settings/security/lesssecureapps

This essentially makes your gmail address acessible by third party applications, which introduces a ton of security issues.

Now that we can use our gmail to send email, let's look at our Emailer class - this will be used to alert analysts when the website was updated in some critical way.

```
import smtplib
import time
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import json

class Emailer:

    def __init__(self,addr='',pw="",website=None): #addr is your email address, pw is passwor
        self.addr = addr
        self.pw = pw
        self.msg = MIMEMultipart('alternative')
        self.receiver = [''] #email to send to
        if website:
            self.msg['Subject'] = "Update to website"+website
        else:
            self.msg['Subject'] = "Update to website"
        self.msg["From"]=self.addr
        self.msg["To"] = ','.join(self.receiver)

    def add_message(self,text):
        if type(text) == type(' '):
            tmp = MIMEText(text,'plain')
            self.msg.attach(tmp)
    def send(self):
        s = smtplib.SMTP('smtp.gmail.com', 587) #used because I don't know how to use Outlook
        s.starttls()
        s.login(self.addr, self.pw)
        s.sendmail(self.addr, self.receiver, self.msg.as_string())
        s.close()
    def add_website(self,site):
        self.website = website
        self.msg['Subject'] = "Update to website"+website
```

The important things here are:
Login:

```
s = smtplib.SMTP('smtp.gmail.com', 587) #used because I don't know how to use Outlook SMTP
s.starttls()
s.login(self.addr, self.pw)
s.sendmail(self.addr, self.receiver, self.msg.as_string())
```

And
Sending:

```
self.msg.attach(tmp)
```

Making use of Our Library

Now that we can send email programatically, let’s make use of it to diff our files:

```
#any variable with _t is today
#any variable with _y is yesterday
for hash_set_t in today_hashes:
    for ind_t,hashing_t in enumerate(hash_set_t):
        name_t = hashing_t.split("000")[1].split(":")[0]
        hash_val_t = hashing_t.split(":")[1]
        for hash_set_y in yesterday_hashes:
            for ind_y,hashing_y in enumerate(hash_set_y):
                hash_val_y = hashing_y.split(":")[1]
                if name_t in hashing_y:
                    if hash_val_t != hash_val_y:
                        self.emailer.add_website(site)
                        self.emailer.add_message("the website was updated")
                        self.emailer.send()
```

If the files are different we alert the analyst something has changed, which may be of interest. Due to the nature of the content and the fact that it’s stored on google’s servers we include as little information as possible in the alert.

Appendix Title A

This Is an A-Head

An appendix is generally used for extra material that supplements your main book content.

Index