

ULTIMATE STEP BY STEP GUIDE TO  
**DATA SCIENCE USING**

# Python

REAL WORLD USE CASES WITH  
DOWNLOADABLE CODE EXAMPLES!



**D A N E Y A L A N I S**

# **ULTIMATE STEP BY STEP GUIDE TO DATA SCIENCE USING PYTHON**

---

REAL WORLD USE CASES WITH DOWNLOADABLE  
CODE EXAMPLES

DANEYAL ANIS

# CONTENTS

## Introduction

1. Getting Started
2. Use Case 1 – Web Scrapping
3. Use Case 2 – Image Processing
4. Use Case 3 – Different File Type Processing
5. Use Case 4 – Sending and Receiving Emails
6. Use Case 5 – Dynamic Time Warping for Speech Analytics
7. Use Case 6 – Time Series Analysis and Forecasting
8. Use Case 7 – Fraud Analysis
9. Use Case 8 – Processing Geospatial Data
10. Use Case 9 – Creating Recommender Systems

## Afterword

11. Post your Review
12. Website and Free Gift (Code to Download)!
13. References

Text and code copyright 2021 Daneyal Anis

All rights reserved.

No part of this book may be reproduced, or transmitted in any form or by any means, electronic, mechanical, magnetic, or photographic, including photocopying, recording, or by any information storage or retrieval system or otherwise, without express written permission of the publisher. No warranty liability whatsoever is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions.

Published by: Daneyal Anis

Date of Publication: November 2021

Language: English

*This book is dedicated to everyone who persevered through these  
tough times and kept an eye on their dreams*



## INTRODUCTION

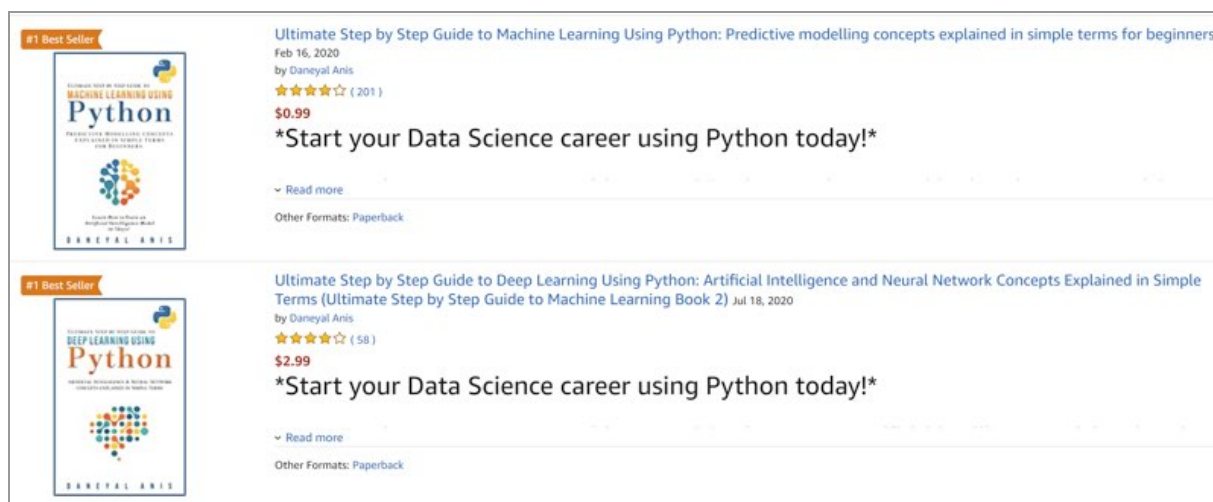
Welcome and congratulations on continuing your machine learning and data science journey! World of machine learning is constantly evolving but beauty of it all is that there are some very common ways in which it is being applied in everyday life. We see these use cases and patterns emerge over and over – from intelligent web scrapping and automation to advanced time series analysis and fraud detection.

The [first best selling book in this series, Ultimate Step by Step Guide to Machine Learning using Python](#), gets you started on your journey by including step by step instructions to set up Python, introduces you to basic syntax, data structures and data cleaning. It then takes you through a real-life use case where you create a machine learning predictive model from scratch! As an added benefit, it comes with code samples that you can run and experiment with on your own! If you are not familiar with basic concepts of Python, how to get it set up and would like to be introduced to foundational Python libraries like Numpy, Seaborn and Scikit-learn, this is the book for you! To purchase this book, follow this link to get redirected to your local Amazon site: <http://mybook.to/MachineLearningPython>.

The [second best selling book in this series, Ultimate Step by Step Guide to Deep Learning using Python](#) gets into deep learning and neural networks concepts. It further differentiates machine learning models from deep learning models and as a bonus, shows you how you can deploy and optimize your models at scale in the cloud. It delves deep



into concepts such as linear and logistical regression analysis and visually shows how decision tree, support vector machine (SVM) and other model optimization algorithms work. It demystifies neural network concepts including Convolutional and Recurrent Neural Networks (CNN and RNN) plus popular advanced Python libraries like TensorFlow, Keras, PyTorch and much more! To purchase this book, follow this link to get redirected to your local Amazon site: <http://mybook.to/DeepLearningPython>.



If you haven't already, I highly encourage you to pick up both previous books in this series, as they take you from basic to advanced concepts with hands on exercises. Since the introductory set up and foundational concepts have already been covered in the previous books in this series, we will not spend time on them in this book.

The goal of this book is to cover the most common machine learning and data science use cases with code examples you can execute on your own. After completing this book and the exercises included within, you will be ready to

solve real world problems with the power of artificial intelligence.

Ready to get started? Let's jump right in!



## GETTING STARTED

There are detailed instructions available via the Python website on how to install Python on your machine – whether it is Windows, Mac OS, or Linux under [Python website in the Getting Started section](#).

I recommend you read through it as there is a lot of good information and helpful links for beginners. However, installing Python by itself is not very helpful or user friendly. You are better off installing Python alongside an IDE (Integrated Development Environment) – as that comes with tools and development environment to execute and debug your code.

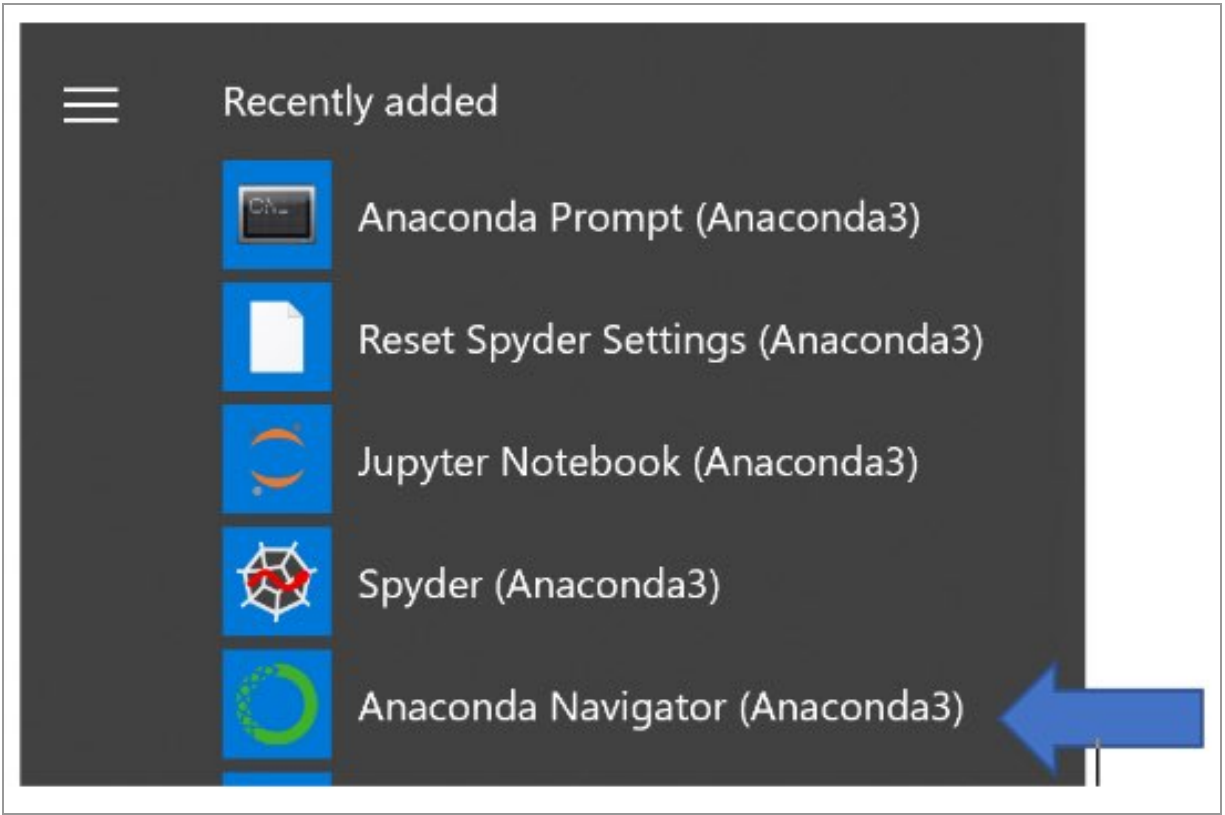
For that I recommend starting with [Anaconda Distribution](#). It is an open-source tool that installs industry standard IDEs and foundational Python libraries that we will be describing in this book in more detail. When you click on the above link, it will take you to a page to download and install package for your operating system e.g., Windows, macOS or Linux.

We will be using [Jupyter Notebook](#) as our development environment for this book – Jupyter is part of the Anaconda distribution package and will be installed on your machine along with Python.

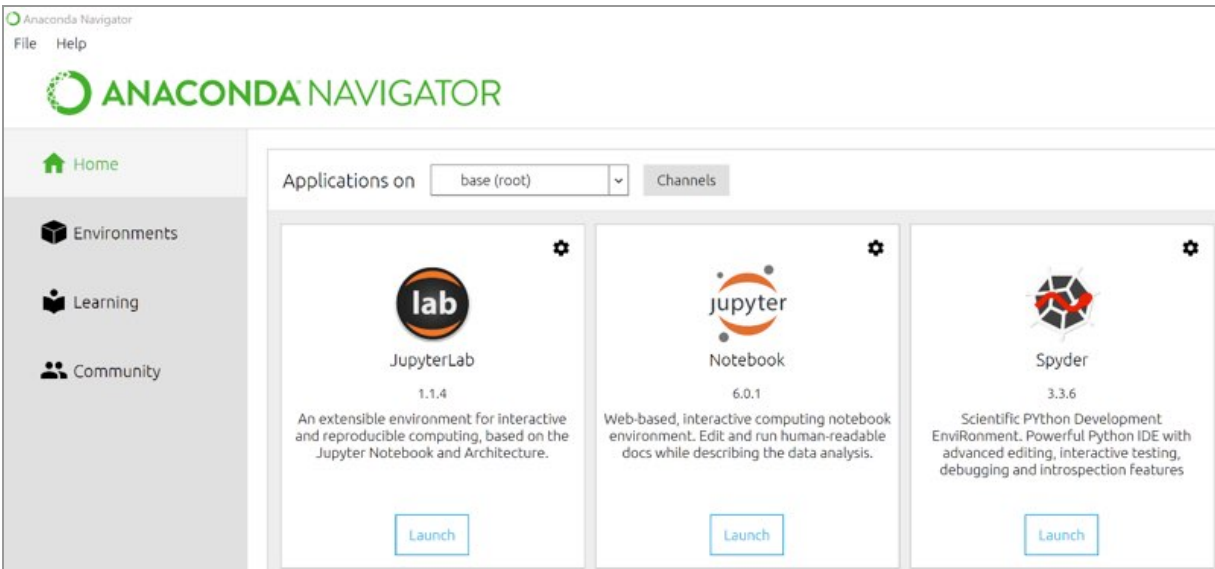
Jupyter is a powerful web-based development environment that we will be using in this book to execute our code and I have made all the source code used in this book available on my website as a ‘.ipnyb’ Jupyter file.

Once Anaconda is installed on your machine, launch Anaconda Navigator from your menu. Screenshot below for

reference:



ONCE OPEN, you should see the following Anaconda Navigator dashboard with all the tools available to you:



Launch Jupyter Notebook by clicking on the ‘Launch’ button under that application icon. Once you do that, the following web browser window will open:



Click on ‘New’ and the ‘Python 3’ in the top right. Once you do that, a new browser window will open with your new project and Python environment ready to execute. You can rename your project as ‘My First Python Project’ by clicking on ‘Untitled’ at the top of the screen (highlighted below for reference).

Done? Alright, you are ready to go!

jupyter

Untitled

Last Checkpoint: 2 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

File Edit View Insert Cell Kernel Widgets Help

Save New Refresh Copy Paste Up Down Run Stop Restart Next Code

In [ ]: ▶





## USE CASE 1 – WEB SCRAPING

**I**nternet is a massive source of data but not all websites make this data easily accessible via API calls or downloadable as .csv files. To be able to access and process this data, you need a way to be able to ‘scrape’ the data off the website. That’s where the power of Python comes in.

It comes with a powerful library called BeautifulSoup that allows you to parse and extract data from any web page for your use. Let’s cover an example end to end.

The main steps involved in extracting data from a web page end to end include:

1. Make a http or https call to the website by using the url, like you would in a browser
2. The website returns the page as html
3. Since html code is typically nested in starting and ending tags: <html> </html>, you use a parser like the BeautifulSoup Python library that lets you organize this information into a tree, that you can easily traverse and access the pertinent details you are interested in. You can use this approach to parse xml data as well

First step is to install the Python libraries we will need. Fire up your Anaconda prompt and run the following commands to install the libraries you will need:

```
pip install requests  
pip install html5lib
```

## **pip install bs4**

I will explain each of the above libraries as we use them in the Python code below (NOTE: All the Python code and data used in this, and subsequent chapters is available for download via my website – instructions are at the end of the book).

In the meanwhile, once you are done installing these libraries, you should get a confirmation like this below in your Anaconda prompt:

### **Successfully installed bs-4**

Now we are ready to execute the Python code, let the fun begin!

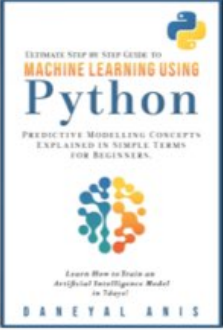
Python library 'requests' downloads the web page as HTML, that we want to parse by making a GET request.

### **import requests**

For this example, we will use the <https://daneyalauthor.com/> web page

```
page = requests.get("https://daneyalauthor.com")
```

For reference, the page looks like this:



**Ultimate Step by Step Guide To  
Python Programming**

Predictive modeling concepts explained in simple terms for beginners

[Join & Get the Bonus Code Samples](#)

Copyright 2020 Daneyal Anis - Disclaimer

We first check whether the data was successfully received. A status\_code of 200 means the get call was successful.

### page.status\_code

We get output as '200' when we run the above line of code, so it was successful as shown in the snapshot below:

```
In [1]: #Python Library 'requests' downloads the web page as HTML, that we want to parse by making a GET request
import requests

In [2]: #For the purpose of this example, we will use the https://daneyalauthor.com/ web page
page = requests.get("https://daneyalauthor.com")

In [3]: #We first check whether the data was successfully received. A status_code of 200 means the get call was successful
page.status_code

Out[3]: 200
```

Now let's see what HTML code content we got from the get call code we made.

### page.content

Below is the output we get:

```
In [4]: #Now Let's see what HTML code content we got from the get call code we made
page.content

<html>
  <head>
    <title>Daneyal Author Home Page &raquo; Feed</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="https://daneyalauthor.com/wp-content/themes/gstater/css/style.css">
    <link rel="alternate" type="application/rss+xml" title="Daneyal Author Home Page &raquo; Feed" href="https://daneyalauthor.com/feed/" />
    <link rel="alternate" type="application/rss+xml" title="Daneyal Author Home Page &raquo; Comments Feed" href="https://daneyalauthor.com/comments/feed/" />
    <link rel="alternate" type="application/rss+xml" title="Daneyal Author Home Page &raquo; Landing Page Comments Feed" href="https://daneyalauthor.com/sample-page/feed/" />
    <script>
      window.wpemojiSettings = {
        "baseUrl": "https://s.w.org/images/core/emoji/12.0.0-1/72x72/",
        "ext": ".png",
        "svgUrl": "https://s.w.org/images/core/emoji/12.0.0-1/svg/",
        "svgExt": ".svg",
        "source": {
          "concatemoji": "https://daneyalauthor.com/wp-includes/js/wp-emoji-release.min.js?ver=5.3.8"
        }
      };
      !function(e,a,t){var n,r,o,i=a.createElement("canvas"),p=i.getContext&&i.getContext("2d");function s(e,t){var a=String.fromCharCode;return p.clearRect(0,0,i.width,i.height),p.fillText(a.apply(this,e),0,0);e=i.toDataURL();return p.clearRect(0,0,i.width,i.height),p.fillText(a.apply(this,t),0,0);e=i.toDataURL();function c(e){var t=a.createElement("script");t.src=e,t.defer=t.type="text/javascript",a.getElementsByTagName("head")[0].appendChild(t)}for(o=Array("flag","emoji"),t.supports={everything:!0,everythingExceptFlag:!0},r=0;r<o.length;r++)t.supports[o[r]]=function(e){if(!p||!p.fillText)return!1;switch(p.textBaseline="top",p.font="600 32px Arial",e){case"flag":return s([127987,65039,8205,9895,65039],[127987,65039,8203,9895,65039])?!1:s([55356,56826,55356,56819],[55356,56826,8203,55356,56819])&&s([55356,57332,56128,56423,56128,56418,56128,56421,56128,56430,56128,56423,56128,56447],[55356,57332,8203,56128,56423,8203,56128,56418,8203,56128,56421,8203,56128,56430,8203,56128,56423,8203,56128,56424,55357,56424,55356,57340],[55357,56424,55356,57340],[55357,56424,55356,57342,8205,55358,56605,8205,55357,56424,55356,57340],[55357,56424,55356,57342,8203,55358,56605,8203,55357,56424,55356,57340])}return!1}[o[r]],t.supports.everything=t.supports.everything&&t.supports[o[r]];t.supports.everythingExceptFlag=t.supports.everythingExceptFlag&&t.supports[o[r]];t.supports.everythingExceptFlag=t.supports.everythingExceptFlag&&t.supports.flag,t.DOMReady=!1,t.readyCallback=function(){t.DOMReady=!0}}
```

That's a lot of content! We will need to parse it in a structured fashion. That's where BeautifulSoup library comes in that has a built-in HTML parser! Isn't Python awesome?

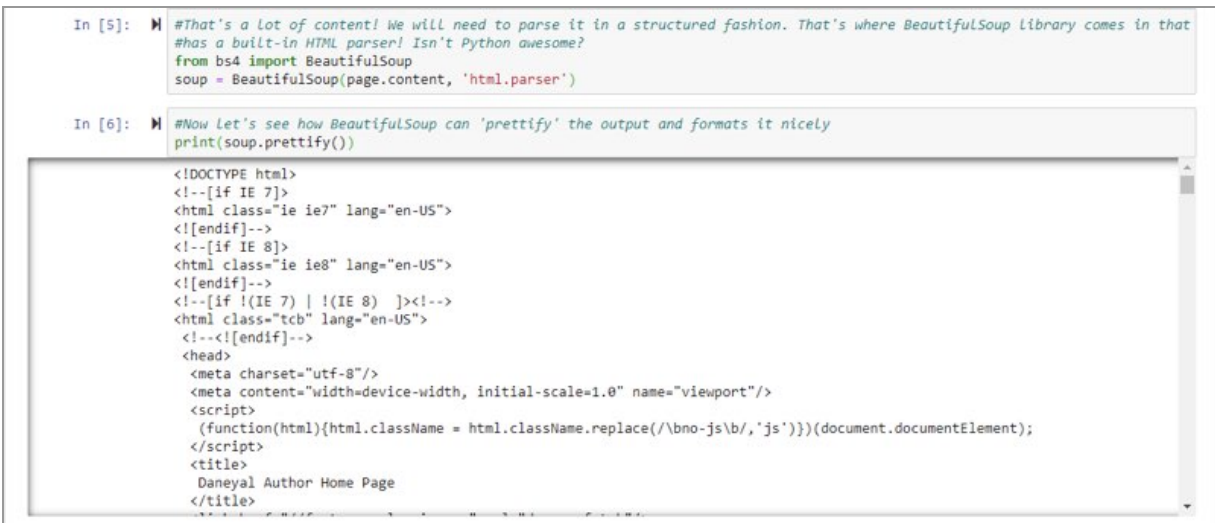
### from bs4 import BeautifulSoup

```
soup = BeautifulSoup(page.content, 'html.parser')
```

Now let's see how BeautifulSoup can 'prettify' the output and formats it nicely.

```
print(soup.prettify())
```

We get the following output when we run the above line of code:



```
In [5]: #That's a lot of content! We will need to parse it in a structured fashion. That's where BeautifulSoup library comes in that
#has a built-in HTML parser! Isn't Python awesome?
from bs4 import BeautifulSoup
soup = BeautifulSoup(page.content, 'html.parser')

In [6]: #Now Let's see how BeautifulSoup can 'prettify' the output and formats it nicely
print(soup.prettify())

<!DOCTYPE html>
<!--[if IE 7]>
<html class="ie ie7" lang="en-US">
<![endif]-->
<!--[if IE 8]>
<html class="ie ie8" lang="en-US">
<![endif]-->
<!--[if !(IE 7) | !(IE 8) ]><!-->
<html class="tcb" lang="en-US">
<!--<![endif]-->
<head>
<meta charset="utf-8"/>
<meta content="width=device-width, initial-scale=1.0" name="viewport"/>
<script>
(function(html){html.className = html.className.replace(/\bno-js\b/, 'js')}})(document.documentElement);
</script>
<title>
Daneyal Author Home Page
</title>
```

Much better! Now on to the parsing. We can now traverse the HTML code by going through the nest tags in the HTML code.

That's accomplished by using the 'children' feature as all nested HTML tags are children of the parent tag. We can organize the data as a Python list data structure, this is explained in [first best-selling book in this series, Ultimate Step by Step Guide to Machine Learning using Python.](#)

```
list(soup.children)
```

We get the following output when we run the above line of code:

```
In [7]: #Much better! Now on to the parsing. We can now traverse the HTML code by going through the nest tags in the HTML code.
#That's accomplished by using the 'children' feature as all nest HTML tags are children of the parent tag
#We can organize the data as a Python List data structure
#(explained in the first book in the series called: Ultimate Step by Step Guide to Machine Learning Using Python)

In [8]: list(soup.children)

Out[8]: ['html',
'\n',
'[if IE 7]>\n<html class="ie ie7" lang="en-US">\n<![endif]',
'\n',
'[if IE 8]>\n<html class="ie ie8" lang="en-US">\n<![endif]',
'\n',
'[if !(IE 7) | !(IE 8) ]><!',
'\n',
<html class="tcb" lang="en-US">
<!--<![endif]<!-->
<head>
<meta charset="utf-8"/>
<meta content="width=device-width, initial-scale=1.0" name="viewport"/>
<script>(function(html){html.className = html.className.replace(/\bno-js\b/, 'js')})(document.documentElement);</script>
<title>Daneyal Author Home Page</title>
<link href="//fonts.googleapis.com" rel="dns-prefetch"/>
<link href="//s.w.org" rel="dns-prefetch"/>
<link crossorigin="" href="https://fonts.gstatic.com" rel="preconnect"/>
<link href="https://daneyalauthor.com/feed/" rel="alternate" title="Daneyal Author Home Page » Feed" type="application/r
```

Now let's find out what different type of HTML elements we have in our list by running the following line of code:  
**[type(item) for item in list(soup.children)]**

```
In [9]: #Now Let's find out what different type of HTML elements we have in our List
[type(item) for item in list(soup.children)]

Out[9]: [bs4.element.Doctype,
bs4.element.NavigableString,
bs4.element.Comment,
bs4.element.NavigableString,
bs4.element.Comment,
bs4.element.NavigableString,
bs4.element.Comment,
bs4.element.NavigableString,
bs4.element.Tag,
bs4.element.NavigableString]
```

Nice! We have the following HTML object types:

1. Doctype object, which as the name implies, contains information about the type of HTML document this is
2. NavigableString, which is the text in the HTML document
3. Tag, which represents the nested tags in the document

#### 4. Comment, which represents the comments on the web page

For this example, we will focus on the Tag object – that is item 8 in the above list, if we count from 0 for the first item.

**html = list(soup.children)[8]**

Let's see what nested tags are included in the Tag object by using the 'children' feature in BeautifulSoup library.

**list(html.children)**

```
In [10]: #Nice! We have the following HTML object types:
#1) Doctype object, which as the name implies, contains information about the type of HTML document this is
#2) NavigableString, which is the text in the HTML document
#3) Tag, which represents the nested tags in the document
#4) Comment, which represents the comments on the web page
#For this example, we will focus on the Tag object - that is item 8 in the above list, if we count from 0 for the first
#item

html = list(soup.children)[8]

In [11]: #Let's see what nested tags are included in the Tag object by using the 'children' feature in BeautifulSoup library
list(html.children)

Out[11]: ['\n', '<![endif]>', '\n', <head>
<meta charset="utf-8"/>
<meta content="width=device-width, initial-scale=1.0" name="viewport"/>
<script>(function(html){html.className = html.className.replace(/\bno-js\b/, 'js'))(document.documentElement);</script>
<title>Daneyal Author Home Page</title>
<link href="//fonts.googleapis.com" rel="dns-prefetch"/>
<link href="//s.w.org" rel="dns-prefetch"/>
<link crossorigin="" href="https://fonts.gstatic.com" rel="preconnect"/>
<link href="https://daneyalauthor.com/feed/" rel="alternate" title="Daneyal Author Home Page » Feed" type="application/r
ss+xml"/>
<link href="https://daneyalauthor.com/comments/feed/" rel="alternate" title="Daneyal Author Home Page » Comments Feed" t
ype="application/rss+xml"/>
<link href="https://daneyalauthor.com/sample-page/feed/" rel="alternate" title="Daneyal Author Home Page » Landing Page
Comments Feed" type="application/rss+xml"/>
<script>
window._wpemojiSettings = {"baseUrl":"https://s.w.org/images/core/emoji/12.0.0-1/72x72
V/", "ext":".png", "svgUrl":"https://s.w.org/images/core/emoji/12.0.0-1/svgV/", "svgExt":".svg", "source":{"concatemoj
i":"https://daneyalauthor.com/wp-includes/js/wp-emoji-release.min.js?ver=5.3.8"}};
!function(e,a,t){var n,r,o,i=a.createElement("canvas"),p=i.getContext&&i.getContext("2d");functio
```

Now suppose we are interested in finding what is in the <title> tag of the HTML code. For that we use the find\_all function which will find all instances of a specific HTML tag.

**soup.find\_all('title')**

And it looks like there is only one instance of the 'title' tag in the HTML, when we run this line of code:

```
In [12]: M #Now suppose we are interested in finding what is in the <title> tag of the HTML code. For that we use the find_all function
#which will find all instances of a specific HTML tag
soup.find_all('title')

Out[12]: [Daneyal Author Home Page</title>]</pre></div><div data-bbox="120 178 880 305" data-label="Text"><p>Let's also look for Cascading Style Sheets (CSS) use in this web page as that helps with the page layout, colors, and fonts. Typically, we will see CSS embedded in the p tags in HTML. So, let's use the select method that returns a list data structure</p></div><div data-bbox="120 362 371 386" data-label="Text"><p><b>SOUP.SELECT("DIV P")</b></p></div><div data-bbox="147 422 869 502" data-label="Text"><pre>In [13]: <b>M</b> #Let's also look for Cascading Style Sheets (CSS) use in this web page as that helps with the page layout, colors and fonts
#Typically we will see CSS embedded in the p tags in HTML. So let's use the select method that returns a list data structure
soup.select("div p")

Out[13]: [<p data-css="tve-u-1640e1d43db"><strong>Ultimate Step by Step Guide To</strong></p>,
<p>Predictive modeling concepts explained in simple terms for beginners</p>,
<p>Copyright 2020 Daneyal Anis - <a class="tve-froala fr-basic" data-css="tve-u-1705659c993" href="https://www.daneyalauthor.com/copyright" style="outline: none;">Disclaimer</a></p>]</pre></div><div data-bbox="120 538 880 615" data-label="Text"><p>The above code returned all instances where CSS style sheets were used, embedded in the p tags – as shown in the 'data-css' part of the HTML code snippet above.</p></div><div data-bbox="120 617 880 748" data-label="Text"><p>If you recall from the <a href="https://daneyalauthor.com">https://daneyalauthor.com</a> web page image earlier in the chapter, there is also a button on the page with a label “Join & Get the Bonus Code Samples”. If you were curious about what action that button triggers, you can run the following line of code:</p></div><div data-bbox="157 750 456 774" data-label="Text"><p><b>soup.find_all('button')</b></p></div><div data-bbox="147 810 553 846" data-label="Text"><pre>In [32]: <b>M</b> #Let's find where does the button on the page takes us
soup.find_all('button')

Out[32]: [<button class="" style="" type="submit">Subscribe Now</button>]</pre></div>
```

As you can see from the snapshot above, the button triggers a subscription action, allowing users to sign up for my mailing list while getting the bonus code samples that are shared in the book! Isn't that fantastic?

Now let's use a different example where we extract a table from HTML and organize as a Panda data frame in Python. These data structures are explained in detail in the [first best-selling book in this series, Ultimate Step by Step Guide to Machine Learning using Python](#), so I would highly recommend reviewing that if you haven't already.

For that, let's use list of academy award winning movies from wiki.

```
import pandas as pd
url = 'https://en.wikipedia.org/wiki/List_of_Academy_Award-winning_films'
dfs = pd.read_html(url)
print(len(dfs))
```

In the above code snippet, we imported the 'pandas' library that allows us to create the Panda data frame. We then read the url of the wiki page by using the 'read\_html' function and then print out the results. For reference, if you go to the above referenced wiki page url, you will see an HTML table of academy award winning movies as follows:



Film	Year	Awards	Nominations
<i>Nomadland</i>	2020/21	3	6
<i>The Father</i>	2020/21	2	6
<i>Judas and the Black Messiah</i>	2020/21	2	6
<i>Minari</i>	2020/21	1	6
<i>Mank</i>	2020/21	2	10
<i>Sound of Metal</i>	2020/21	2	6
<i>Ma Rainey's Black Bottom</i>	2020/21	2	5
<i>Promising Young Woman</i>	2020/21	1	5
<i>Tenet</i>	2020/21	1	2
<i>Soul</i>	2020/21	2	3
<i>Another Round</i>	2020/21	1	2
<i>My Octopus Teacher</i>	2020/21	1	1
<i>Colette</i>	2020/21	1	1
<i>If Anything Happens I Love You</i>	2020/21	1	1
<i>Two Distant Strangers</i>	2020/21	1	1

Here are the results we get when we execute the above lines of code:

```
In [20]: #Now Let's use a different example where we extract a table from HTML and organize as a Panda dataframe in Python
#For that, let's use list of academy award winning movies from wiki
import pandas as pd
|
url = 'https://en.wikipedia.org/wiki/List_of_Academy_Award-winning_films'
dfs = pd.read_html(url)
print(len(dfs))
2
```

This means there are two tables on the wiki page. Let's look at the table output for first table

```
df = dfs[0]
print(df)
```

Note that we used [0] in the data frame array to grab the first table. Here are the results we get when we execute the above line of code:

```
In [28]: # This means there are two tables on the wiki page. Let's take a look at the table output for first table
df = dfs[0]
print(df)
```

	Film	Year	Awards	\
0	Nomadland	2020/21	3	
1	The Father	2020/21	2	
2	Judas and the Black Messiah	2020/21	2	
3	Minari	2020/21	1	
4	Mank	2020/21	2	
...	...	...	...	...
1327	The Yankee Doodle Mouse	1943	1	
1328	The Yearling	1946	2	
1329	Yesterday, Today and Tomorrow (Ieri, oggi, dom...	1964	1	
1330	You Can't Take It with You	1938	2	
1331	Zorba the Greek (Alexis Zorbas)	1964	3	

	Nominations
0	6
1	6
2	6
3	6
4	10
...	...
1327	1
1328	7
1329	1
1330	7
1331	7

[1332 rows x 4 columns]

We can also access individual columns by using column names in the table:

```
print(df['Film'])  
print(df['Year'])  
print(df['Awards'])
```

Here are the results we get when we execute the above lines of code:

```

In [22]: #We can also access individual columns this way:
print(df['Film'])
print(df['Year'])
print(df['Awards'])

0          Nomadland
1          The Father
2    Judas and the Black Messiah
3          Minari
4          Mank
...
1327    The Yankee Doodle Mouse
1328    The Yearling
1329    Yesterday, Today and Tomorrow (Ieri, oggi, dom...
1330    You Can't Take It with You
1331    Zorba the Greek (Alexis Zorbas)
Name: Film, Length: 1332, dtype: object
0    2020/21
1    2020/21
2    2020/21
3    2020/21
4    2020/21
...
1327    1943
1328    1946
1329    1964
1330    1938
1331    1964
Name: Year, Length: 1332, dtype: object
0    3
1    2
2    2
3    1
4    2
..
1327    1
1328    2
1329    1
1330    2
1331    3
Name: Awards, Length: 1332, dtype: object

```

Now, assume we are only interested in movie name and year and don't want the rest of the table. We can create a subset of the data frame, only containing these two columns:

```

df2 = df[['Film','Year']]
print(df2)

```

```
In [29]: #Assume we are only interested in movie name and year  
df2 = df[['Film','Year']]  
print(df2)
```

	Film	Year
0	Nomadland	2020/21
1	The Father	2020/21
2	Judas and the Black Messiah	2020/21
3	Minari	2020/21
4	Mank	2020/21
...	...	...
1327	The Yankee Doodle Mouse	1943
1328	The Yearling	1946
1329	Yesterday, Today and Tomorrow (Ieri, oggi, dom...	1964
1330	You Can't Take It with You	1938
1331	Zorba the Greek (Alexis Zorbas)	1964

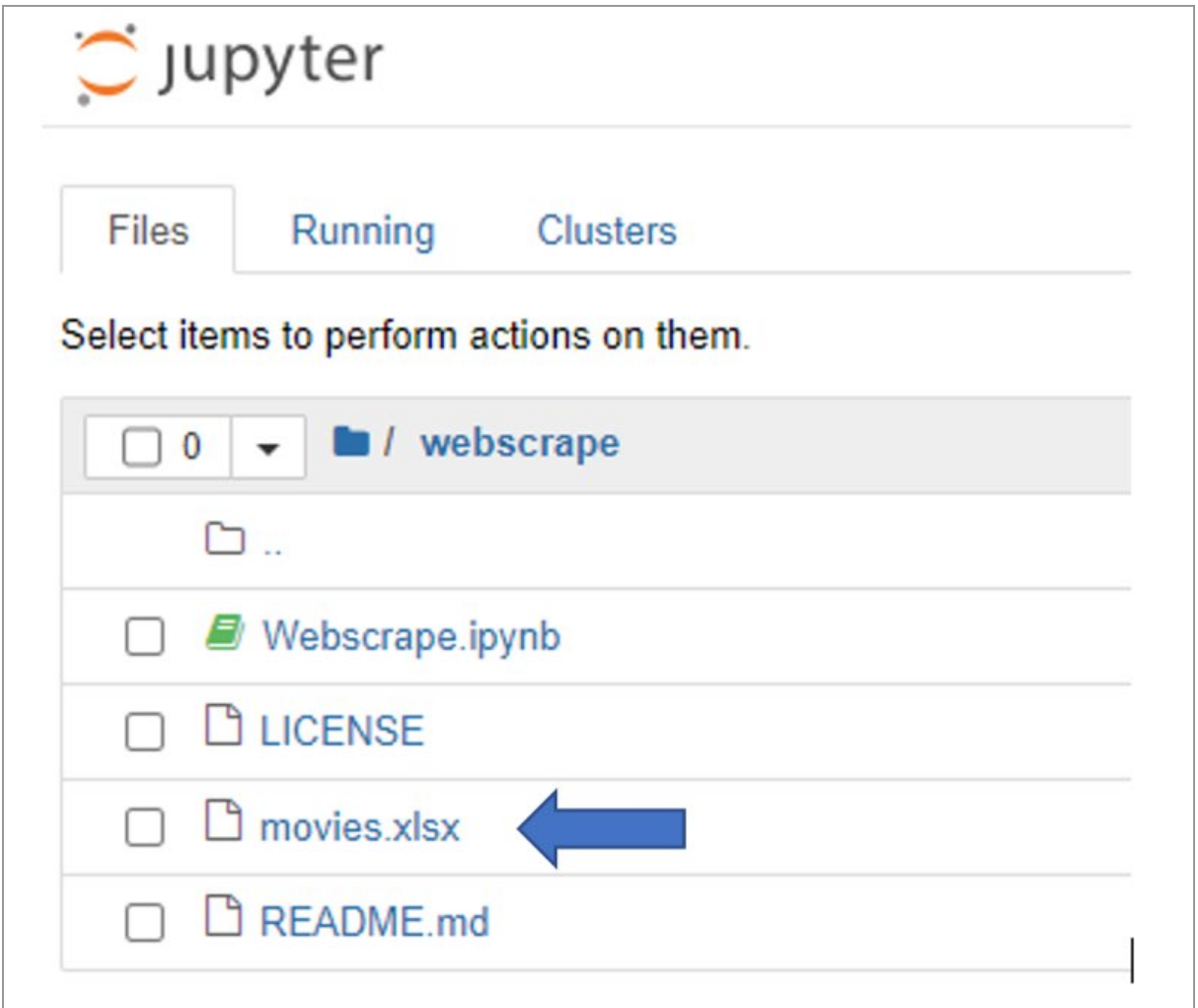
[1332 rows x 2 columns]

Now that the data is nicely organized in a Panda data frame, let's export to excel for additional analysis by using 'to\_excel' function.

```
df2.to_excel('movies.xlsx')
```

```
In [30]: #Now that the data is nicely organized in a Panda data frame, Let's export to excel for additional analysis  
#by using 'to_excel' function  
df2.to_excel('movies.xlsx')
```

You should now see the excel file called 'movies.xlsx' created in the folder where your Python code resides.



There you have it! That's how you scrape and access data from an HTML web page into Python.



## USE CASE 2 – IMAGE PROCESSING

**I**mage processing is a very popular use case for data science – from simple usage like applying image filters to your Instagram photos to more complex use cases like cancer cell analysis by applying pattern recognition.

As described in detail in the [second best-selling book in this series, Ultimate Step by Step Guide to Deep Learning using Python](#), Convolutional Neural Networks (CNN) are widely used with image data. The name derives from the Convolutions that are realized between matrices in each layer. It essentially is a mathematical operation that translates each image into a matrix, so it can be processed and manipulated as required for the purpose at hand.

Luckily, Python comes with several built-in powerful libraries for image processing that we will use to illustrate this concept in more detail below.

We will start with pyplot and scikit-image libraries which is an open-source Python library and works well with Numpy arrays.

```
import matplotlib.pyplot as plt  
%matplotlib inline
```

We import sample images and filters from scikit-image library. Full list of test images available can be seen at this URL:

```
https://scikit-image.org/docs/dev/api/skimage.data.html  
from skimage import data, filters
```

We use the checkerboard test image from the library and print it in grayscale

```
image = data.checkerboard()
```

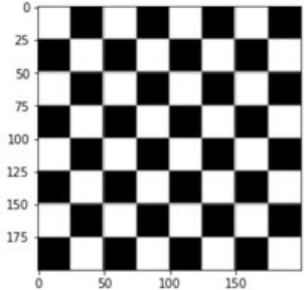
**plt.imshow(image, cmap='gray')**

```
In [11]: import matplotlib.pyplot as plt
%matplotlib inline

#We import sample images and filters from scikit-image Library
#Full list of test images available at this URL
#https://scikit-image.org/docs/dev/api/skimage.data.html
from skimage import data, filters

#We use the checkerboard test image from the Library and print it in grayscale
image = data.checkerboard()
plt.imshow(image, cmap='gray')
```

Out[11]: <matplotlib.image.AxesImage at 0x288a88ba688>



Now let's apply some filters on this image. Full list of filters is available at this URL:

<https://scikit-image.org/docs/stable/api/skimage.filters.html>

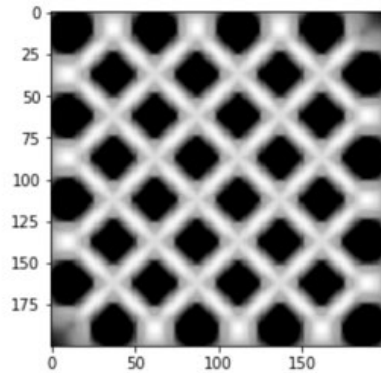
We will first use the 'Sato' image filter which applies a blurry look to the image.

```
edges = filters.sato(image)
plt.imshow(edges, cmap='gray')
```



```
In [15]: ▶ #Now lets apply some filters on this image
#Full list of filters available at this URL
#https://scikit-image.org/docs/stable/api/skimage.filters.html
#We will first use the 'Sato' image filter which applies a tubeness Look to the image
edges = filters.sato(image)
plt.imshow(edges, cmap='gray')
```

Out[15]: <matplotlib.image.AxesImage at 0x288a89e4b88>

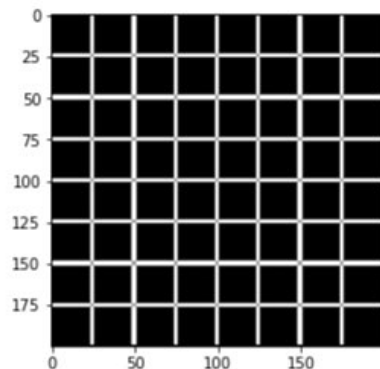


We will then use the 'Scharr' image filter which finds the edge magnitude of the image.

```
edges = filters.scharr(image)  
plt.imshow(edges, cmap='gray')
```

```
In [16]: ▶ #We will then use the 'Scharr' image filter which finds the edge magnitude of the image
edges = filters.scharr(image)
plt.imshow(edges, cmap='gray')
```

Out[16]: <matplotlib.image.AxesImage at 0x288a8a496c8>



Let's now use Numpy, which as introduced in the first book in this series, is one of the foundational Python

libraries. Numpy converts the image into an array data structure for each manipulation. We will still use scikit-image library for loading and displaying the image.

```
import numpy as np  
image = data.coffee()  
plt.imshow(image, cmap='gray')
```

```
In [22]: import numpy as np  
#Let's now use Numpy, which as introduced in the first book in this series, is one of the foundational Python Libraries  
#Numpy converts the image into an array data structure for each manipulation  
#We will still use scikit-image Library for loading and displaying the image  
  
image = data.coffee()  
plt.imshow(image, cmap='gray')
```

Out[22]: <matplotlib.image.AxesImage at 0x288a8df5908>



**type(image)**

Let's convert this image into a Numpy array:

**np.ndarray**

Now let's mask this image:

**mask = image < 85**

**image[mask]=256**

**plt.imshow(image, cmap='gray')**

```
In [24]: type(image)
#Lets convert this image into a Numpy array
np.ndarray

#Now Lets mask this image
mask = image < 85
image[mask]=256
plt.imshow(image, cmap='gray')

Out[24]: <matplotlib.image.AxesImage at 0x288a9e9a308>
```



As introduced in the first book in the series, Scipy is a foundational Python library used for mathematical and scientific calculations. It works well with Numpy data structures and comes with built-in libraries for image processing, as part of scipy.ndimage sub-module.

Detailed documentation available via this URL:  
<https://docs.scipy.org/doc/scipy/reference/tutorial/ndimage.html#correlation-and-convolution>

```
Let's see it in action:
from scipy import ndimage
image = data.chelsea()
Original Image:
plt.imshow(image)
```

```
In [31]: #As introduced in the first book in the series, Scipy is a foundational Python library used for mathematical and scientific
#calculations. It works well with Numpy data structures and comes with built-in libraries for image processing, as part of
#scipy.ndimage sub-module.
#Detailed documentation available via this URL:
#https://docs.scipy.org/doc/scipy/reference/tutorial/ndimage.html#correlation-and-convolution

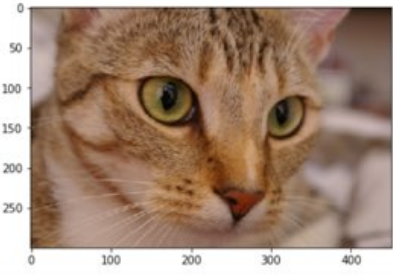
#Lets see it in action

from scipy import ndimage

image = data.chelsea()

#Original Image
plt.imshow(image)

Out[31]: <matplotlib.image.AxesImage at 0x288aa24d7c8>
```



Now let's apply a light Gaussian filter to this cat image to make it blurry. Details of the Gaussian filter and how it works are included in the second book in the series and you can also refer to additional documentation on types of filters available by following this URL:

[https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian\\_filter.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian_filter.html)

```
blurred_image = ndimage.gaussian_filter(image,
sigma=3)
plt.imshow(blurred_image)
```

```
In [34]: #Now Let's apply a light Gaussian filter to this cat image to make it blurry  
#Details of the Gaussian filter and how it works are included in the second book in the series and you can also refer to  
#additional documentation on types of filters available by following this URL:  
#https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian_filter.html  
  
blurred_image = ndimage.gaussian_filter(image, sigma=3)  
plt.imshow(blurred_image)
```

Out[34]: <matplotlib.image.AxesImage at 0x288aa368788>



And now, let's make it blurry by increasing the sigma value for the Gaussian filter:

```
very_blurred_image = ndimage.gaussian_filter(image,  
sigma=5)  
plt.imshow(very_blurred_image)
```

```
In [35]: #And now, Let's make it really blurry by increasing the sigma value for the Gaussian filter  
very_blurred_image = ndimage.gaussian_filter(image, sigma=5)  
plt.imshow(very_blurred_image)
```

Out[35]: <matplotlib.image.AxesImage at 0x288aa8b6688>



There are more advanced libraries for image processing in Python which require additional set up and install steps. Discussion of these libraries is outside the scope of this book, but you are welcome to reference the links below for more details and if you would like to try them out:

1) PIL (Python Imaging Library) - this library is good for additional convolution functions, color formatting and filters:

<https://pillow.readthedocs.io/en/stable/>

2) OpenCV (Open-Source Computer Vision Library) - this library is used for computer vision applications that require a lot of computation firepower. This library is use for faster processing because it is built using C/C++ with a Python wrapper:

<https://github.com/abidrahmank/OpenCV2-Python-Tutorials>

There you go! Now you know enough to process images and apply filters in Python!



## USE CASE 3 - DIFFERENT FILE TYPE PROCESSING

**D**ata scientists often work with different file types to extract and process data before it is ready for analysis. Knowing how to work with different file types is an important skill to have in your toolkit. In this chapter, we will go over Python processing modes for different file types and how to organize this data into built-in Python data structure. Let's get started!

Python can work with the following file types:

- Comma-separated values (CSV)
- XLSX (Excel)
- Plain Text (txt)
- JSON
- XML
- HTML
- PDF (Adobe)
- DOCX (Microsoft Word)
- Images
- MP3
- MP4
- ZIP

Python also has different file processing modes:

Mode

Description

'r'

This is the default read mode.

'w'



This is the writing mode. If a file already exists, it will write to it and if not, it will create a new one with the name provided.

'x'

This is to create a new file and if the file already exists, it will give a failure message.

'a'

This is the mode to add / append to an existing file. If the file does not exist, it will create a new one.

't'

By default, Python assumes it is a text file.

'b'

Switch to this mode for processing the file in binary mode.

'+'

This is the update mode – so the file is opened in reading and writing mode.

LET'S first start with reading from CSV file and storing its information in a Panda data frame. For this example, we will create a .csv file in the same folder where the Python code resides for easy processing.

Let's first read a CSV file containing comma separated values for Oscar winning movies. The file looks like this:

```
csvtest.csv - Notepad
File Edit Format View Help
Film,Year,Awards,Nominations
7 Faces of Dr. Lao,1964,0 (1),1
7th Heaven,1927/28,3,5
8 Mile,2002,1,1
12 Years a Slave,2013,3,9
20 Feet from Stardom,2013,1,1
"20,000 Leagues Under the Sea",1954,2,3
1917,2019,3,10
2001: A Space Odyssey,1968,1,4
A Fantastic Woman,2017,1,1
A Force in Readiness,1961,0 (1),0
A Girl in the River: The Price of Forgiveness,2015,1,1
A Herb Alpert and the Tijuana Brass Double Feature,1966,1,1
```

We will import contents of the csv file into a Pandas data frame:

**import pandas as pd**

**df\_csv = pd.read\_csv('csvtest.csv')**

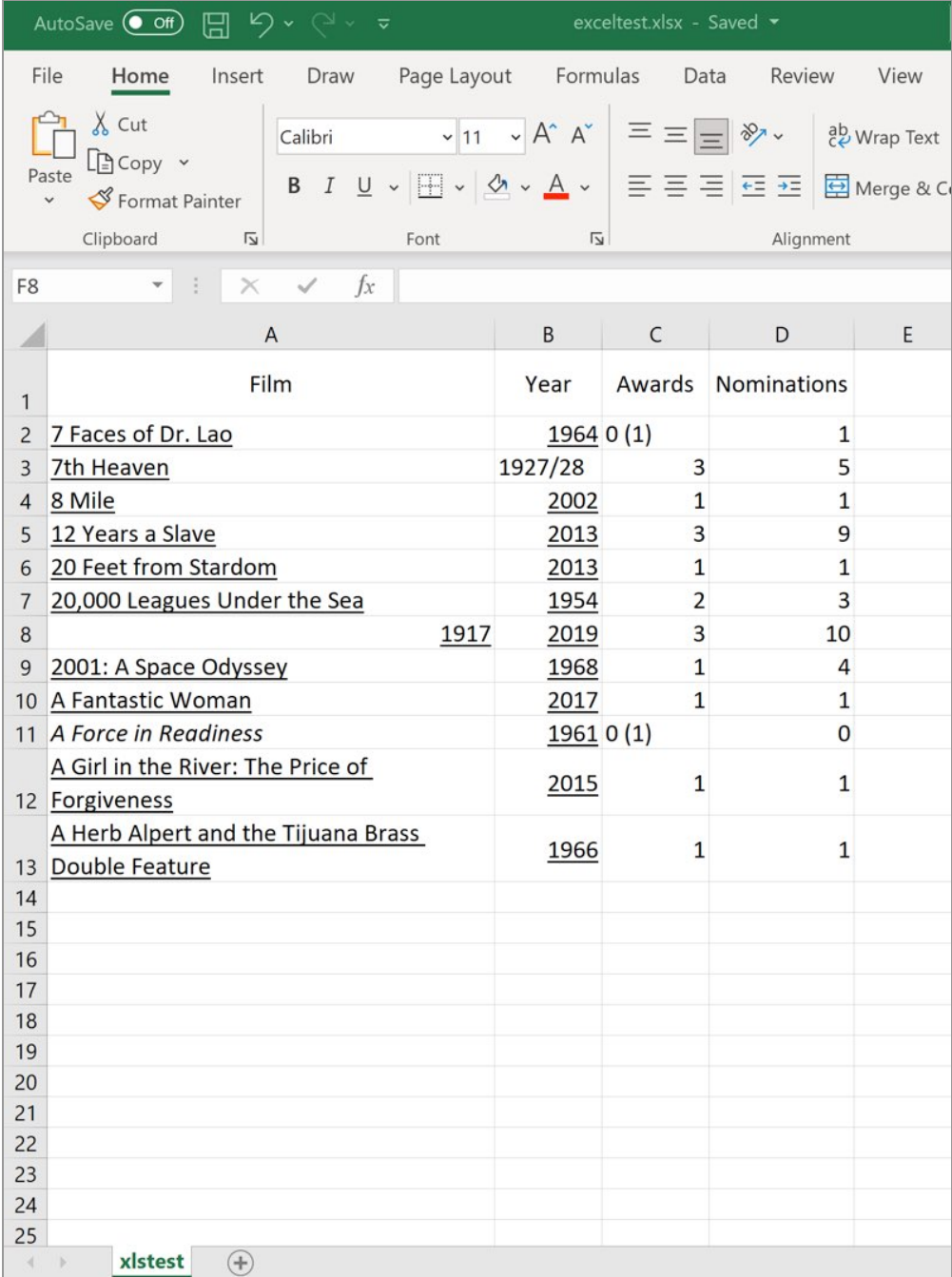
Now let's display the contents of this file in a data frame:

**display(df\_csv)**

```
In [8]: #We will import contents of the csv file into a Pandas data frame
import pandas as pd
df_csv = pd.read_csv('csvtest.csv')
#Now lets display the contents of this file in a dataframe
display(df_csv)
```

	Film	Year	Awards	Nominations
0	7 Faces of Dr. Lao	1964	0 (1)	1
1	7th Heaven	1927/28	3	5
2	8 Mile	2002	1	1
3	12 Years a Slave	2013	3	9
4	20 Feet from Stardom	2013	1	1
5	20,000 Leagues Under the Sea	1954	2	3
6	1917	2019	3	10
7	2001: A Space Odyssey	1968	1	4
8	A Fantastic Woman	2017	1	1
9	A Force in Readiness	1961	0 (1)	0
10	A Girl in the River: The Price of Forgiveness	2015	1	1
11	A Herb Alpert and the Tijuana Brass Double Fea...	1966	1	1

We can do the same with an excel file. Our sample excel file looks like this:



The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E
	Film	Year	Awards	Nominations	
1					
2	<u>7 Faces of Dr. Lao</u>	<u>1964</u>	0 (1)		1
3	<u>7th Heaven</u>	<u>1927/28</u>	3		5
4	<u>8 Mile</u>	<u>2002</u>	1		1
5	<u>12 Years a Slave</u>	<u>2013</u>	3		9
6	<u>20 Feet from Stardom</u>	<u>2013</u>	1		1
7	<u>20,000 Leagues Under the Sea</u>	<u>1954</u>	2		3
8		<u>1917</u>	<u>2019</u>	3	10
9	<u>2001: A Space Odyssey</u>	<u>1968</u>	1		4
10	<u>A Fantastic Woman</u>	<u>2017</u>	1		1
11	<u>A Force in Readiness</u>	<u>1961</u>	0 (1)		0
12	<u>A Girl in the River: The Price of Forgiveness</u>	<u>2015</u>	1		1
13	<u>A Herb Alpert and the Tijuana Brass Double Feature</u>	<u>1966</u>	1		1
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					
25					

However, since excel can have multiple tabs, we must specify the sheet name:

```
df_xls = pd.read_excel('exceltest.xlsx',  
sheet_name='xlstest')
```

Display contents of excel file:

```
display(df_xls)
```

```
In [7]: # We can do the same with an excel file. However, since excel can have multiple tabs, we have to specify sheet name  
df_xls = pd.read_excel('exceltest.xlsx', sheet_name='xlstest')  
  
# Display contents of excel file  
display(df_xls)
```

	Film	Year	Awards	Nominations
0	7 Faces of Dr. Lao	1964	0 (1)	1
1	7th Heaven	1927/28	3	5
2	8 Mile	2002	1	1
3	12 Years a Slave	2013	3	9
4	20 Feet from Stardom	2013	1	1
5	20,000 Leagues Under the Sea	1954	2	3
6	1917	2019	3	10
7	2001: A Space Odyssey	1968	1	4
8	A Fantastic Woman	2017	1	1
9	A Force in Readiness	1961	0 (1)	0
10	A Girl in the River: The Price of Forgiveness	2015	1	1
11	A Herb Alpert and the Tijuana Brass Double Fea...	1966	1	1

Now that we have the data nicely organized in a Pandas dataframe, we can process it like a data structure. We can access the columns:

```
print(df_xls.columns.ravel())
```

```
In [9]: # Now that we have the data nicely organized in a Pandas dataframe, we can process it like a data structure  
# We can access the columns  
print(df_xls.columns.ravel())  
  
['Film' 'Year' 'Awards' 'Nominations']
```

And we can access contents of a specific column, by using the .tolist() function. In the below example, we list all movie names in our excel file:

```
print(df_xls['Film'].tolist())
```

```
In [10]: #And we can also access contents of a specific column, by using the .tolist() function. In the below example, we list
#all movie names in our excel file
print(df_xls['Film'].tolist())

['7 Faces of Dr. Lao', '7th Heaven', '8 Mile', '12 Years a Slave', '20 Feet from Stardom', '20,000 Leagues Under the Sea', 1
917, '2001: A Space Odyssey', 'A Fantastic Woman', 'A Force in Readiness', 'A Girl in the River: The Price of Forgiveness',
'A Herb Alpert and the Tijuana Brass Double Feature']
```

You can even convert the excel file content into other formats like csv and json. First, we show results as JSON:

**print('Print Excel as JSON:', df\_xls.to\_json(orient='records'))**

```
In [13]: #You can even convert the excel file content into other formats like csv and json. First we show results as JSON
print('Print Excel as JSON:', df_xls.to_json(orient='records'))

Print Excel as JSON: [{"Film": "7 Faces of Dr. Lao", "Year": 1964, "Awards": "0 (1)", "Nominations": 1}, {"Film": "7th Heaven", "Year": "1927/28", "Awards": 3, "Nominations": 5}, {"Film": "8 Mile", "Year": 2002, "Awards": 1, "Nominations": 1}, {"Film": "12 Years a Slave", "Year": 2013, "Awards": 3, "Nominations": 9}, {"Film": "20 Feet from Stardom", "Year": 2013, "Awards": 1, "Nominations": 1}, {"Film": "20,000 Leagues Under the Sea", "Year": 1954, "Awards": 2, "Nominations": 3}, {"Film": "1917", "Year": 2019, "Awards": 3, "Nominations": 10}, {"Film": "2001: A Space Odyssey", "Year": 1968, "Awards": 1, "Nominations": 4}, {"Film": "A Fantastic Woman", "Year": 2017, "Awards": 1, "Nominations": 1}, {"Film": "A Force in Readiness", "Year": 1961, "Awards": "0 (1)", "Nominations": 0}, {"Film": "A Girl in the River: The Price of Forgiveness", "Year": 2015, "Awards": 1, "Nominations": 1}, {"Film": "A Herb Alpert and the Tijuana Brass Double Feature", "Year": 1966, "Awards": 1, "Nominations": 1}]
```

And then we show the results as a CSV:

**print('Print Excel as CSV', df\_xls.to\_csv(index=False))**

```
In [14]: #And then we show the results as a CSV
print('Print Excel as CSV', df_xls.to_csv(index=False))

Print Excel as CSV Film,Year,Awards,Nominations
7 Faces of Dr. Lao,1964,0 (1),1
7th Heaven,1927/28,3,5
8 Mile,2002,1,1
12 Years a Slave,2013,3,9
20 Feet from Stardom,2013,1,1
"20,000 Leagues Under the Sea",1954,2,3
1917,2019,3,10
2001: A Space Odyssey,1968,1,4
A Fantastic Woman,2017,1,1
A Force in Readiness,1961,0 (1),0
A Girl in the River: The Price of Forgiveness,2015,1,1
A Herb Alpert and the Tijuana Brass Double Feature,1966,1,1
```

To read a json file, like excel and csv files, you can use the Pandas read\_json function. For this purpose, we use JSON file sample from:

<https://json.org/example.html>

Our JSON file sample looks like this:

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup languages such as DocBook.",
            "GlossSeeAlso": ["GML", "XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

```
df_json = pd.read_json('jsontest.json')
```

DISPLAY CONTENTS OF JSON FILE:

```
display(df_json)
```

```
In [15]: #To read a json file, similar to excel and csv files, you can use the Pandas read_json function. For this purpose,
#we use JSON file sample from https://json.org/example.html
df_json = pd.read_json('jsontest.json')

#Display contents of json file
display(df_json)
```

glossary	
GlossDiv	{'title': 'S', 'GlossList': {'GlossEntry': {1...
title	example glossary

To process an XML file, we use ElementTree library. We will use the parse function to process the xmltest sample that we got from:

<https://docs.python.org/3/library/xml.etree.elementtree.html>

It contains sample data for countries and their neighbors:

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
  <country name="Panama">
    <rank>68</rank>
    <year>2011</year>
    <gdppc>13600</gdppc>
    <neighbor name="Costa Rica" direction="W"/>
    <neighbor name="Colombia" direction="E"/>
  </country>
</data>
```

```
import xml.etree.ElementTree as ET
tree = ET.parse('countries.xml')
```

Now that we have parsed the country data as a tree structure, let's get the root of the tree:

```
root = tree.getroot()
```

```
In [24]: M #To process an XML file, we use ElementTree library  
#We will use the parse function to process the xmltest sample that we got from  
#https://docs.python.org/3/library/xml.etree.elementtree.html  
#It contains sample data for countries and their neighbours  
  
import xml.etree.ElementTree as ET  
tree = ET.parse('countries.xml')  
#Now that we have parsed the country data as a tree structure, Lets get the root of the tree  
root = tree.getroot()
```

Let's see what the root of the tree contains. Each element of the tree, including the root, has tags that describe it:

**root.tag**

**root.attrib**

```
In [28]: M #Lets see what the root of the tree contains. Each elements of the tree, including the root has tags that describe it  
root.tag  
  
Out[28]: 'data'  
  
In [26]: M root.attrib  
  
Out[26]: {}
```

Now print all children of the root of the tree, and their corresponding tags and attributes

**for child in root:**

**... print(child.tag, child.attrib)**

```
In [27]: M #Now print all children of the root of the tree, and their corresponding tags and attributes  
for child in root:  
...     print(child.tag, child.attrib)  
  
country {'name': 'Liechtenstein'}  
country {'name': 'Singapore'}  
country {'name': 'Panama'}
```

NOTE: We will not cover parsing HTML files in this chapter as that was covered in detail in previous web scraping chapter and we used BeautifulSoup library for that. Similarly for processing image files please check out the chapter on image processing in this book.

For processing PDF files, you will have to first install PDF mining library from this link:

<https://euske.github.io/pdfminer/>



You can then run the following line of code to extract the PDF file as txt

```
pdf2txt.py pdftest.pdf
```

For processing Word (.docx) files, you will have to first install doc2txt library by using this command:

```
pip install docx2txt
```

Once installed, you can run the following code to extract the DOCX file as txt:

```
import docx2txt  
text = docx2txt.process("doctest.docx")
```

Print the contents of the word file:

```
print (text)
```



```
In [31]: #For processing Word (.docx) files, you will have to first install doc2txt library by using this command  
#pip install docx2txt  
#Once installed, you can run the following code to extract the DOCX file as txt  
import docx2txt  
text = docx2txt.process("doctest.docx")
```

```
In [32]: #Print the contents of the word file  
print (text)
```

Film  
Year  
Awards  
Nominations  
7 Faces of Dr. Lao  
1964  
0 (1)  
1  
7th Heaven  
1927/28

Sometimes data scientists can also receive an archived ZIP file to process in Python. You must first import zipfile library and then read the results into a data frame.

```
import zipfile
```

```
archive = zipfile.ZipFile('data.zip', 'r')
```

Let's look at the content of this zip file:

data.zip		
<input type="checkbox"/>	Name	Type
	countries.xml	XML Document
	csvtest.csv	Microsoft Excel Comma S...
	doctest.docx	Microsoft Word Document
	exceltest.xlsx	Microsoft Excel Worksheet
	jsontest.json	JSON File
	pdftest.pdf	Adobe Acrobat Document

Once we have the entire archive read in, we can read one of the files in the archive into a data frame:

```
df_archive = archive.read('csvtest.csv')
display(df_archive)
```

```
In [34]: #Sometimes data scientists can also receive an archive ZIP file to process in Python
#You have to first import zipfile library and then read the results into a dataframe
import zipfile
archive = zipfile.ZipFile('data.zip', 'r')
#Once we have the entire archive read in, we can read one of the files in the archive into a dataframe
df_archive = archive.read('csvtest.csv')

In [35]: display(df_archive)

b'\xef\xbb\xbfFilm,Year,Awards,Nominations\r\n7 Faces of Dr. Lao,1964,0 (1),1\r\n7th Heaven,1927/28,3,5\r\n8 Mile,2002,1,1\r\n12 Years a Slave,2013,3,9\r\n20 Feet from Stardom,2013,1,1\r\n"20,000 Leagues Under the Sea",1954,2,3\r\n1917,2019,3,10\r\n2001: A Space Odyssey,1968,1,4\r\nA Fantastic Woman,2017,1,1\r\nA Force in Readiness,1961,0 (1),0\r\nA Girl in the River: The Price of Forgiveness,2015,1,1\r\nA Herb Alpert and the Tijuana Brass Double Feature,1966,1,1\r\n'
```

Finally, if you ever want to process multimedia files in Python, you can do that by installing the PyMedia library by following this link: <http://pymedia.org/tut/index.html>

Well, there you go. Now you can process different types of files in Python and prepare the data for additional analysis.



## USE CASE 4 – SENDING AND RECEIVING EMAILS

**P**ython makes it easy to send and receive emails. For that all you need is powerful libraries called `smtplib` and `imap`. Now why would you want to send and receive emails using a programming language you ask? It is to automate the mundane activity of sending mass emails to an email list while using a custom template. It is also to allow you to parse through your different inboxes and look for key information. Python makes it all possible!

Before we get into Python code, let's first explain what SMTP is. It stands for Simple Mail Transfer Protocol. It was created in 1982 and is still in use by big email providers of the world like Gmail, Yahoo Mail, and others. In simplest terms, it is the language used by mail servers to communicate with each other to send and receive emails. Got it? Let's move on.

Python library `smtplib` uses the SMTP protocol and has built-in functions to send emails.

Let's first import the libraries we are going to use. Notice that we imported BeautifulSoup library too as we will be using it later to parse HTML content for our emails. You will also notice that we imported the MIME standards - that's what will allow us to email messages in different formats like binary, ASCII, HTML, and others.

```
import smtplib  
from email import encoders  
from email.mime.text import MIMEText  
from email.mime.multipart import MIMEMultipart  
from email.mime.base import MIMEBase
```

## from bs4 import BeautifulSoup as bs

```
In [15]: ¶ #Let's first import the libraries we are going to use. Notice that we imported BeautifulSoup library too as we will be using  
#it later to parse HTML content for our emails. You will also notice that we imported the MIME standards - that's what  
#will allow us to email messages in different formats like binary, ASCII, HTML and others.  
  
import smtplib  
from email import encoders  
from email.mime.text import MIMEText  
from email.mime.multipart import MIMEMultipart  
from email.mime.base import MIMEBase  
from bs4 import BeautifulSoup as bs
```

Now let's define our parameters like email address, password, email subject, sending and receiving email addresses - so we can input the required values as needed. Remember, these values are placeholders only and you can plug in your own credentials for sending your emails. Note that though, I would not recommend hard coding your credentials because if you share your code, your account can be compromised. Also, when providing receiving email, instead of putting in one email, you can put in multiple email addresses or even an email list that can be read in from a file (refer to the earlier chapter on how to read in excel or csv files as a Python data structure).

Let's first set the credentials - as in our email and password:

```
EMAIL = "myemail@test.com"
```

```
PASSWORD = "mypassword"
```

We then also specify the sender and receiver email addresses:

```
FROM = "sender@test.com"
```

```
TO = "receiver@test.com"
```

Lastly, we set the subject of the email:

```
SUBJECT = "Subject of my email"
```

```
In [16]: M #Now let's define our parameters like email address, password, email subject, sending and receiving email addresses -  
#so we can input the required values as needed. Remember, these values are placeholders only and you can plug in your  
#own credentials for sending your emails. Note that though, I would not recommend hard coding your credentials because  
#if you share your code, your account can be compromised. Also, when providing receivers email, instead of putting in  
#one email, you can put in multiple email addresses or even an email list that can be read in from a file  
  
#Credentials  
EMAIL = "myemail@test.com"  
PASSWORD = "mypassword"  
#Sender's email  
FROM = "sender@test.com"  
#Receiver's email  
TO = "receiver@test.com"  
#Subject of the email  
SUBJECT = "Subject of my email"
```

Now we are going to initialize our message object, so it is ready to send:

```
msg = MIMEMultipart("alternative")
```

Setting the 'from' property in the message to the FROM email address value we provided earlier:

```
msg["From"] = FROM
```

Setting the 'to' property in the message to the TO email address value we provided earlier:

```
msg["To"] = TO
```

Setting the 'subject' of the message to the SUBJECT value we provided earlier:

```
msg["Subject"] = SUBJECT
```

```
In [17]: M #Now we are going to initialize our message object so it is ready to send.  
msg = MIMEMultipart("alternative")  
#Setting the from property in the message to the FROM email address value we provided earlier  
msg["From"] = FROM  
#Setting the to property in the message to the TO email address value we provided earlier  
msg["To"] = TO  
#Setting the subject of the message to the SUBJECT value we provided earlier |  
msg["Subject"] = SUBJECT
```

Now let's create HTML and text versions of the email message. This is the HTML version of the message:

```
html = """  
I love <b>Python </b>!  
"""
```

Now let's convert this HTML version to text using HTML parser object:

```
text = bs(html, "html.parser").text
```

```
In [18]: #Now Let's create HTML and text versions of the email message  
#This is the HTML version of the message  
html = """  
I love <b>Python </b>!  
"""  
  
#Now let's convert this HTML version to text using HTML parser object  
text = bs(html, "html.parser").text
```

Alternatively, you can also read the HTML and text versions of your message from the corresponding files in your folder. Setting the body of the email as HTML by reading from an HTML file. In this case the file is in the same folder as the Python code. If it is in a different folder, you must provide a fully qualified path:

```
html = open("message.html").read()
```

And then convert to text the same way as before using html parser:

```
text = bs(html, "html.parser").text
```

```
In [19]: #Alternatively, you can also read the HTML and text versions of your message from the corresponding files in your folder  
#Setting the body of the email as HTML by reading from an HTML file. In this case the file is in the same folder as the  
#Python code. If it is in a different folder, you have to provide a fully qualified path  
html = open("message.html").read()  
#And then convert to text the same as before using html parser  
text = bs(html, "html.parser").text
```

Now let's finish building the message, by setting the text and HTML values:

```
text_part = MIMEText(text, "plain")
```

```
html_part = MIMEText(html, "html")
```

Finally, we attach the email body to the mail message - both text and HTML versions:

```
msg.attach(text_part)
```

```
msg.attach(html_part)
```

```
In [20]: M #Now let's finish building the message, by setting the text and HTML values
text_part = MIMEText(text, "plain")
html_part = MIMEText(html, "html")
#Finally, we attach the email body to the mail message - both text and HTML versions
msg.attach(text_part)
msg.attach(html_part)
```

After constructing the HTML and text version of the message and attaching it to the mail, let's see what it looks like:

```
print(msg.as_string())
```

You will notice that the output is broken out in sections, separated by characters.

- 1) First part is message header including sender, receiver, and email subject
- 2) Second part is the message in text format
- 3) Third part is the message in HTML format

```
In [12]: M #After constructing the HTML and text version of the message and attaching it to the mail, let's see what it looks like
print(msg.as_string())
#You will notice that the output is broken out in sections, separated by characters.
#1) First part is message header including sender, receiver and email subject
#2) Second part is the message in text format
#3) Third part is the message in HTML format |

Content-Type: multipart/alternative; boundary="=====0507507794483387575=="
MIME-Version: 1.0
From: sender@test.com
To: receiver@test.com
Subject: Subject of my email

-----0507507794483387575==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

I love Python
-----0507507794483387575==
Content-Type: text/html; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

<HTML><B>I love Python</B></HTML>
-----0507507794483387575=---
```

To make this process repeatable, let's create a function that takes From, To, Subject and Message as arguments, so it can send the message for us, and we can call this function whenever we need to send a message again with a different set of email recipients and message content:



**def send\_message (EMAIL, PASSWORD, FROM, TO, msg):**

Here we are using the gmail SMTP, but you can use any others of your choosing like Yahoo, Outlook etc. Full list of SMTP protocols and corresponding ports to use can be found here:

<https://www.arclab.com/en/kb/email/list-of-smtp-and-pop3-servers-mailserver-list.html>

NOTE: If you are using gmail email for this purpose, you will need to set the "Allow less secure apps" option in your gmail settings. This will indeed make your gmail account less secure, so we recommend using a separate account for this purpose for sending and receiving mass emails as opposed to your personal account.

Initializing the SMTP server connection:

**server = smtplib.SMTP("smtp.gmail.com", 587)**

Connecting to the SMTP server as secure TLS (Transport Layer Security) mode:

**server.starttls()**

Logging in with your credentials that are passed to the function:

**server.login(email, password)**

Sending the email:

**server.sendmail(FROM, TO, msg.as\_string())**

Terminating SMTP session:

**server.quit()**

```
In [21]: M #To make this process repeatable, Let's create a function that takes From, To, Subject and Message as arguments, so
#it can send the message for us and we can call this function whenever we need to send a message again with a different
#set of email recipients and message content
def send_message (EMAIL, PASSWORD, FROM, TO, msg):
    #Here we are using the gmail SMTP but you can use any others of your choosing like Yahoo
    #Outlook etc. Full List of SMTP protocols and corresponding ports to use can be found here:
    #https://www.arclab.com/en/kb/email/List-of-smtp-and-pop3-servers-mailserver-List.html
    #NOTE: If you are using gmail email for this purpose, you will need to set the "Allow Less secure apps" option in your
    #gmail settings. This will indeed make your gmail account less secure, so we recommend using a separate account for
    #this purpose for sending and receiving mass emails as opposed to your personal account

    #Initializing the SMTP server connection
    server = smtplib.SMTP("smtp.gmail.com", 587)
    #Connecting to the SMTP server as secure TLS (Transport Layer Security) mode
    server.starttls()
    #Logging in with your credentials that are passed to the function
    server.login(email, password)
    #Sending the email
    server.sendmail(FROM, TO, msg.as_string())
    #Terminating SMTP session
    server.quit()
```

Finally, we send the message by calling the function we defined above. Now in this case we will get an SMTP authentication error as per below because we are using placeholder credentials but if you use your real credentials, you should receive an email in the corresponding email inbox application:

**send\_message(EMAIL, PASSWORD, FROM, TO, msg)**

```
In [22]: M #Finally we send the message by calling the function we defined above. Now in this case we will get an SMTP authentication
#error as per below because we are using fake credentials but if you use your real credentials, you should receive
#an email in the corresponding email inbox application

send_message(EMAIL, PASSWORD, FROM, TO, msg)
```

You can also add attachments to your message by adding list of files to be included in the email. Below is a sample list of files as an example:

```
files_to_send = [
    "test.txt",
    "test.pdf",
]
```

Initializing the message as before:

```
msg = MIMEMultipart("alternative")
```

Setting the sender's email:

```
msg["From"] = FROM
```

Setting the recipient's email:

```
msg["To"] = TO
```

Setting the subject of the email:

```
msg["Subject"] = SUBJECT
```

Setting the body of the message:

```
html = open("message.html").read()
```

Converting the message from HTML to text as before:

```
text = bs(html, "html.parser").text
```

```
text_part = MIMEText(text, "plain")
```

```
html_part = MIMEText(html, "html")
```

Attaching the email body to the mail message. First adding text and then HTML to the body:

```
msg.attach(text_part)
```

```
msg.attach(html_part)
```

```
for file in files_to_send:
```

Creating this loop to read the file list created earlier and going through them one at a time:

```
with open(file, "rb") as f:
```

Read each file in the list:

```
data = f.read()
```

Add as attachment to the file:

```
attach_part = MIMEBase("application", "octet-stream")
```

```
attach_part.set_payload(data)
```

Add 64 bit encoding

```
encoders.encode_base64(attach_part)
```

Including message header

```
attach_part.add_header("Content-Disposition",  
f"attachment; filename= {file}")
```

```
msg.attach(attach_part)
```

## Finally sending the message

### `send_mail(email, password, FROM, TO, msg)`

```
In [23]: M #You can also add attachments to your message by adding list of files to be included in the email
#Below is a sample list of files as an example
files_to_send = [
    "test.txt",
    "test.pdf",
]
#Initializing the message as before
msg = MIME multipart("alternative")
#Sender's email
msg["From"] = FROM
#Recipient's email
msg["To"] = TO
#Subject of the email
msg["Subject"] = SUBJECT
#Setting the body of the message
html = open("message.html").read()
#Converting the message from HTML to text
text = bs(html, "html.parser").text
text_part = MIMEText(text, "plain")
html_part = MIMEText(html, "html")

#Attaching the email body to the mail message
#First adding text and then HTML to the body
msg.attach(text_part)
msg.attach(html_part)
```

```
for file in files_to_send:
    #Creating this loop to read the file list created earlier and looping
    with open(file, "rb") as f:
        #Read each file in the list
        data = f.read()
        #Add as attachment to the file
        attach_part = MIMEBase("application", "octet-stream")
        attach_part.set_payload(data)
        #Add 64 bit encoding
        encoders.encode_base64(attach_part)
        #Including message header
        attach_part.add_header("Content-Disposition", f"attachment; filename= {file}")
        msg.attach(attach_part)
# Finally send the message
send_mail(email, password, FROM, TO, msg)
```

There you have it! A handy way to send multiple emails with different attachments and email templates - a very good use case for automation via Python. But why stop there? Now to take it one step further, let's look at how we can automate reading emails via Python as well. We will also look at how to automate downloading of email attachments. For that we will use the IMAP protocol that comes built-in as a handy library in Python called `imaplib`. IMAP is different from POP3 protocol we used earlier in this chapter because POP3 protocol reads and downloads the email from the server to read it offline, while IMAP protocol leaves the email

message on the server while reading it. Let's import the necessary libraries we need to read emails:

```
import imaplib  
import email  
from email.header import decode_header  
import webbrowser  
import os
```

```
In [25]: ▶ #But why stop there. Now to take it one step further, Let's take a look at how we can automate reading emails via Python  
#as well. We will also take a look at how to automate downloading of email attachments. For that we will use the IMAP  
#protocol that comes built-in as a handy library in Python called imaplib. IMAP is different from POP3 protocol we used  
#earlier in this chapter because POP3 protocol reads and downloads the email from the server to read it offline, while  
#IMAP protocol leaves the email message on the server while reading it  
  
#Let's import the necessary libraries we need to read emails  
  
import imaplib  
import email  
from email.header import decode_header  
import webbrowser  
import os
```

Let's specify our placeholder account credentials (NOTE: You will need to use your own real credentials for the code to work) and a function to create folders in your email program of choice, without special characters.

First, we set the credentials, just like we did earlier when we were sending the email:

```
In [43]: ▶ #Let's specify our placeholder account credentials (NOTE: You will need to use your own real credentials for the code to  
#work) and also create a function to create folders in your email program of choice, without special characters  
  
#Credentials  
EMAIL = "myemail@test.com"  
PASSWORD = "mypassword"  
  
def clean(text):  
    #Create clean folder names by removing special characters  
    return "".join(c if c.isalnum() else "_" for c in text)
```

Next, we will connect to IMAP, in this case assuming a Gmail account:

```
imap = imaplib.IMAP4_SSL("imap.gmail.com")
```

Authentication step will fail in this case because we are using placeholder credentials, but should work if you use your real credentials. See instructions earlier in the chapter on how to change Gmail settings for access from non-secure apps. Even if you use the correct credentials, Gmail and other apps may still block access if they don't recognize access by an external application like Python - so you will need to enable access by third party apps if you want to use this feature:

**imap.login(EMAIL, PASSWORD)**

```
In [44]: | #Next we will connect to IMAP, in this case assuming a gmail account
imap = imaplib.IMAP4_SSL("imap.gmail.com")
#Authentication step will fail in this case because we are using placeholder credentials, but should work if you use your
#real credentials. See instructions earlier in the chapter on how to change gmail settings for access from non-secure apps
#Even if you use the correct credentials, Gmail and other apps may still block access if they don't recognize access by
#an external application like Python - so you will want to enable access by third party apps if you want to use this feature
imap.login(EMAIL, PASSWORD)|
```

Once you successfully login, you can access emails by specifying the folder you want to retrieve the messages from and specifying how many messages you want to retrieve. In the example below, we use the select method to retrieve 5 messages from our INBOX folder:

**status, messages = imap.select("INBOX")**

Indicating number of top emails to fetch:

**N = 5**

Converting total number of emails in the inbox to integer value:

**messages = int(messages[0])**

```
In [36]: | #Once you successfully login, you can retrieve emails by specifying the folder you want to retrieve the messages from
#and also specifying how many messages you want to retrieve. In the example below, we use the select method to retrieve
#5 messages from the our INBOX folder
status, messages = imap.select("INBOX")
# number of top emails to fetch
N = 5
# total number of emails
messages = int(messages[0])|
```

In this case messages variable now contains total number of messages in our inbox as an integer type so we can create a loop and status variable contains whether the message was retrieved successfully - we are looking for the status 'OK'.

Now let's create a loop and use the IMAP fetch function to retrieve the body of the message for the first 5 messages we want to retrieve. We will use the 'RFC822' standard email format to retrieve each message:

```
for i in range(messages, messages-N, -1):
```

First, we use the fetch method and RFC822 format as mentioned earlier:

```
res, msg = imap.fetch(str(i), "(RFC822)")
```

```
for response in msg:
```

```
if isinstance(response, tuple):
```

We then parse email in bytes format into a message object in Python:

```
msg = email.message_from_bytes(response[1])
```

We then decode the email subject:

```
subject, encoding = decode_header(msg["Subject"])[0]
```

```
if isinstance(subject, bytes):
```

If the subject is in bytes format, we convert to string:

```
subject = subject.decode(encoding)
```

After that, we decode email sender:

```
From, encoding = decode_header(msg.get("From"))[0]
```

```
if isinstance(From, bytes):
```

```
From = From.decode(encoding)
```

```
print("Subject:", subject)
```

```
print("From:", From)
```

If the email message is multipart, we convert each of the parts and process separately:

```
if msg.is_multipart():
Iterating over multi-part message:
for part in msg.walk():
Extract the content type of the email part:
content_type = part.get_content_type()
content_disposition = str(part.get("Content-
Disposition"))
try:
We then get the email body:
body = part.get_payload(decode=True).decode()
except:
pass
if content_type == "text/plain" and "attachment" not in
content_disposition:
We print text part of the message and see the content:
print(body)
elif "attachment" in content_disposition:
We then download the attachment:
filename = part.get_filename()
if filename:
folder_name = clean(subject)
if not os.path.isdir(folder_name):
We create a folder for this downloaded email (using the
subject as the label):
os.mkdir(folder_name)
filepath = os.path.join(folder_name, filename)
and then download attachment and save it:
open(filepath,
"wb").write(part.get_payload(decode=True))
else:
```



We extract content type of email:

```
content_type = msg.get_content_type()
```

and get the email body:

```
body = msg.get_payload(decode=True).decode()
```

```
if content_type == "text/plain":
```

print only text part of the email:

```
print(body)
```

```
if content_type == "text/html":
```

If the content type is HTML, create a new HTML file and open it in browser:

```
folder_name = clean(subject)
```

```
if not os.path.isdir(folder_name):
```

make a folder for this email named after the subject of the email:

```
os.mkdir(folder_name)
```

```
filename = "index.html"
```

```
filepath = os.path.join(folder_name, filename)
```

write the file to the folder:

```
open(filepath, "w").write(body)
```

and then open the HTML file in the default browser:

```
webbrowser.open(filepath)
```

```
print("="*100)
```

Once done, we close the IMAP connection and logout.

```
imap.close()
```

```
imap.logout()
```

```

In [45]: M #In this case messages variable now contains total number of messages in our inbox as an integer type so we can create a
#Loop and status variable contains whether the message was retrieved successfully. We are looking for the status 'OK'
#Now let's create a loop and use the IMAP fetch function to retrieve the body of the message for the first 5 messages
#we want to retrieve. We will use the 'RFC822' standard email format to retrieve each message

for i in range(messages, messages-N, -1):
    # fetch the email message by ID
    res, msg = imap.fetch(str(i), "(RFC822)")
    for response in msg:
        if isinstance(response, tuple):
            # parse a bytes email into a message object
            msg = email.message_from_bytes(response[1])
            # decode the email subject
            subject, encoding = decode_header(msg["Subject"])[0]
            if isinstance(subject, bytes):
                # if it's a bytes, decode to str
                subject = subject.decode(encoding)
            # decode email sender
            From, encoding = decode_header(msg.get("From"))[0]
            if isinstance(From, bytes):
                From = From.decode(encoding)
            print("Subject:", subject)
            print("From:", From)
            # if the email message is multipart
            if msg.is_multipart():
                # iterate over email parts
                for part in msg.walk():
                    # extract content type of email
                    content_type = part.get_content_type()
                    content_disposition = str(part.get("Content-Disposition"))

```

```

try:
    # get the email body
    body = part.get_payload(decode=True).decode()
except:
    pass
if content_type == "text/plain" and "attachment" not in content_disposition:
    # print text/plain emails and skip attachments
    print(body)
elif "attachment" in content_disposition:
    # download attachment
    filename = part.get_filename()
    if filename:
        folder_name = clean(subject)
        if not os.path.isdir(folder_name):
            # make a folder for this email (named after the subject)
            os.mkdir(folder_name)
        filepath = os.path.join(folder_name, filename)
        # download attachment and save it
        open(filepath, "wb").write(part.get_payload(decode=True))
else:
    # extract content type of email
    content_type = msg.get_content_type()
    # get the email body
    body = msg.get_payload(decode=True).decode()
    if content_type == "text/plain":
        # print only text email parts
        print(body)
    if content_type == "text/html":
        # if it's HTML, create a new HTML file and open it in browser
        folder_name = clean(subject)
        if not os.path.isdir(folder_name):
            # make a folder for this email (named after the subject)
            os.mkdir(folder_name)

```

```

filename = "index.html"
filepath = os.path.join(folder_name, filename)
# write the file
open(filepath, "w").write(body)
# open in the default browser
webbrowser.open(filepath)
print("*100")
# close the connection and logout
imap.close()
imap.logout()

```

If the access is all successful, I get confirmation email in my Python output as follows:

**Subject: Congrats Email Test worked!**

**From: myemail@test.com**

**Subject: An email with a text file as an attachment**

**From: Test Account <myemail@test.com>**

**Get the text attachment!**

**Subject: A Test message with attachment**

**From: Test Account <myemail@test.com>**

**It worked!**

Also, I see that the code created folders for downloaded emails in my local directory where the Python code resides. There you have it folks! You can now not only send mass emails using an email list automatically via Python but also download and store emails and attachments locally - what a powerful automation feature via Python!

For additional reading on the Python IMAP and email libraries, please use the links below:

IMAP

library:

<https://docs.python.org/3/library/imaplib.html>

Email

library:

<https://docs.python.org/3/library/email.html>



## USE CASE 5 – DYNAMIC TIME WARPING FOR SPEECH ANALYTICS

**N**ow that we have covered some basic Python real world use cases like web scraping, file, and image processing as well as automating sending and receiving emails, it is time to address more advanced use cases. How about warping time?

Now, now...don't worry...we are not messing with the space time continuum and don't want to start new multiverses like in the comic books. Instead, in this chapter, we are going to discuss the concept of time warping using Python. What is time warping you ask? It is a spin on traditional time series analysis where you want to compare two datasets that occurred over the same period – however, you run into a challenge where the x-axis which represents time is not in the same scale between the two data sets i.e., does not have the same start and end time.

Ever wonder how AI powered home assistants like Amazon Alexa and Google Home recognize your voice and a specific phrase like “Stop” no matter how fast or slow you say it? Or comparing results of financial markets between months but one month had a smaller number of days in the previous year because it was a leap year? That's where time warping comes in – you basically ‘warp’ your time axis to make the two data sets comparable.

In the example in this chapter, we will two matching audio phrases that are said at a different pace and one completely different audio phrase of the same length and then use time warping libraries in Python to compare the

results to find the right match. This type of use case is very common in speech and pattern recognition in the real world.

The phrase we will use is **“One Flew Over the Cuckoo’s Nest”** and it will be stored in two different audio files spoken in different voices and in different inflections. The contrasting phrase we will use is **“I Love Python And I Can’t Stop”**. I will make the audio files along with the Python code available to you, so you can test it for yourself as well.

Let the fun begin!

First you need to install FastDTW library from Python for time warping analysis. Run the following the command in your Python application prompt:

**pip install FastDTW**

Once installed, you are ready to import the FastDTW library and use it in your code. While you are at it, you can also import other libraries you will need for your analysis including scipy wavfile library to process audio files, matplotlib for plotting the results and numpy for advanced calculations – which we will get into below.

```
FROM SCIPY.IO IMPORT wavfile  
from matplotlib import pyplot as plt  
import numpy as np  
from scipy.spatial.distance import euclidean  
from fastdtw import fastdtw
```

Now let’s read the audio files below in for comparison. The first two files have the same phrase, and the third file has a different phrase for comparison purposes.

```
fs1, data1 = wavfile.read("oneflew1.wav")
```

```
fs2, data2 = wavfile.read("oneflew2.wav")
fs3, data3 = wavfile.read("ilovepython.wav")
```

```
In [31]: M #First you need to install FastDW library from Python for time warping analysis
#Run the following the command in your Python application prompt: pip install FastDW

from scipy.io import wavfile
from matplotlib import pyplot as plt
import numpy as np

from scipy.spatial.distance import euclidean
from fastdtw import fastdtw

# Read stored audio files for comparison
fs1, data1 = wavfile.read("oneflew1.wav")
fs2, data2 = wavfile.read("oneflew2.wav")
fs3, data3 = wavfile.read("ilovepython.wav")
```

Let's visualize the audio files we just imported using Python's matplotlib library we imported earlier and setting the plot style to have a white background:

```
from matplotlib.pyplot import figure
plt.style.use("seaborn-whitegrid")
```

The first audio file contains the phrase "One Flew Over the Cuckoo's Nest" in an American male voice. We set the formatting and color parameters for the plot in the code snippet below:

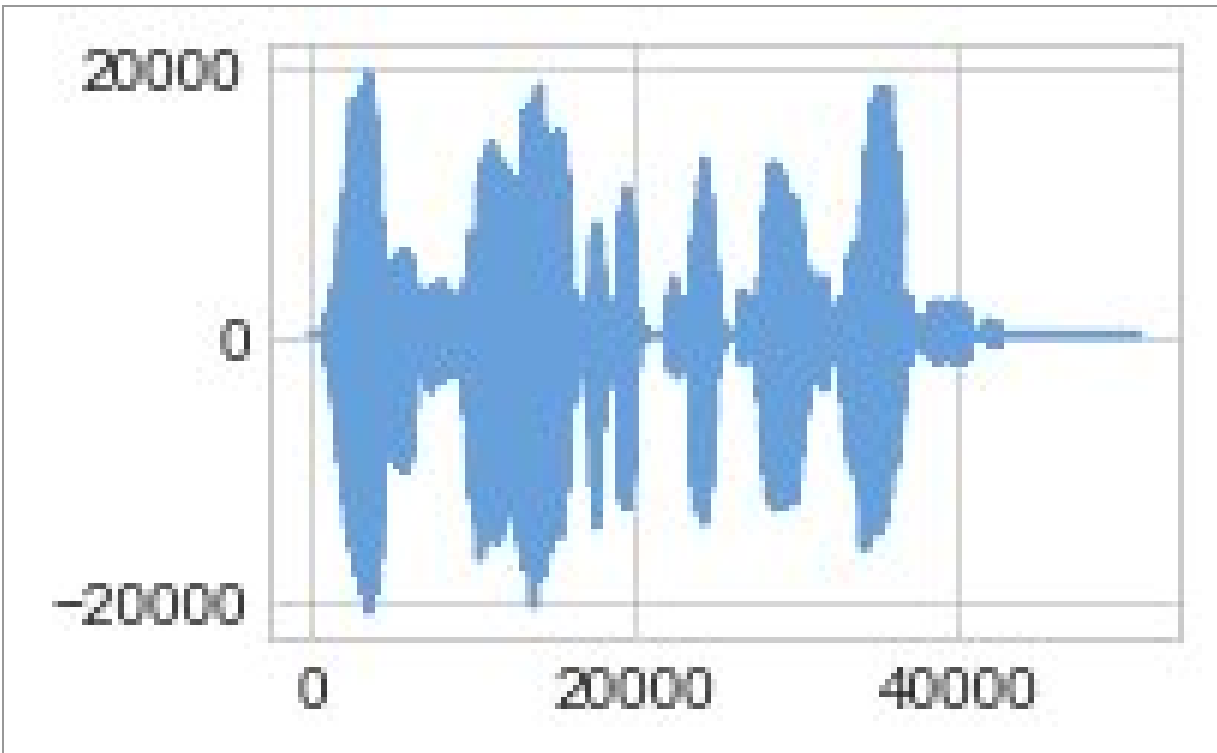
```
ax = plt.subplot(2, 2, 1)
ax.plot(data1, color="#67A0DA")
```

And then we finally visualize the audio file:

```
fig=plt.show()
display(fig)
```

```
In [32]: M #Let's visualize the audio files we just imported using Python's matplotlib library we imported earlier
from matplotlib.pyplot import figure
# Set plot style
plt.style.use("seaborn-whitegrid")

# Create subplots for first audio file for "One Flew Over the Cuckoo's Nest"
ax = plt.subplot(2, 2, 1)
ax.plot(data1, color="#67A0DA")
# Display created figure for first audio file for "One Flew Over the Cuckoo's Nest"
fig=plt.show()
display(fig)
```

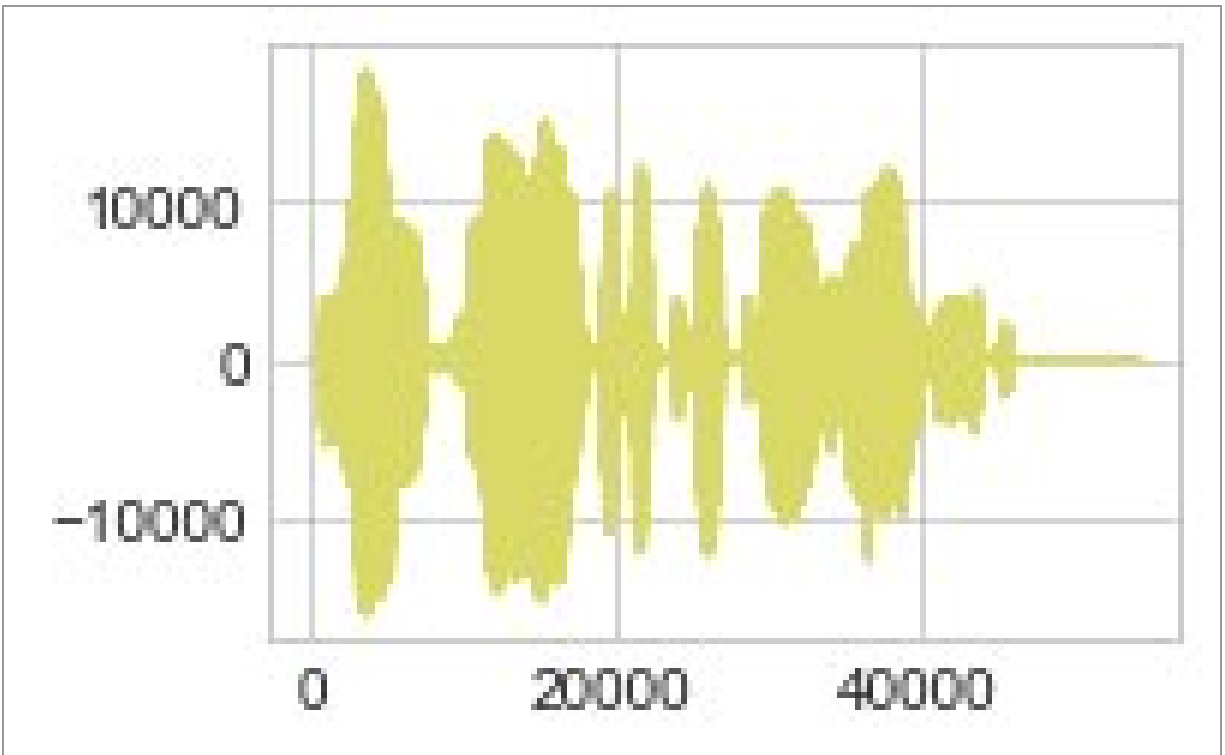


Now let's display the remaining two audio files as well using the same code snippets but different color schemes:

```
In [33]: # Create subplots for second audio file for "One Flew Over the Cuckoo's Nest"
ax = plt.subplot(2, 2, 1)
ax.plot(data2, color="#dad867")

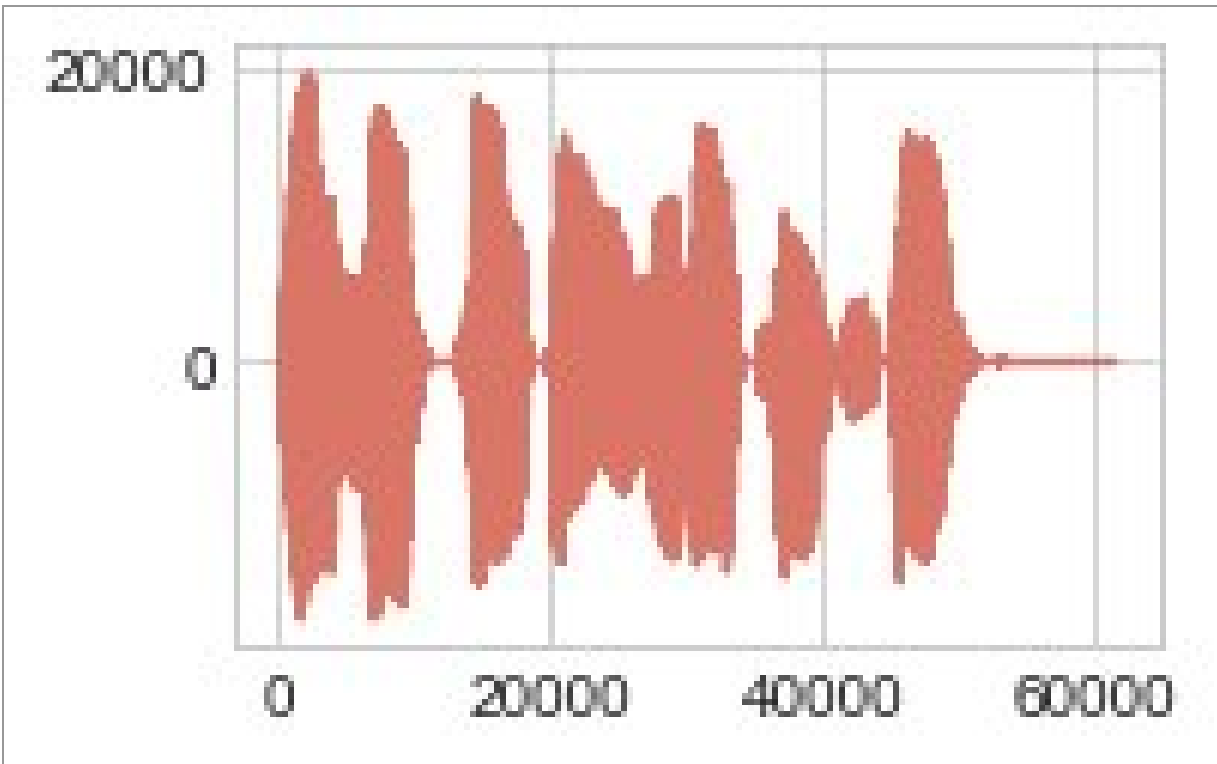
# Display created figure for second audio file for "One Flew Over the Cuckoo's Nest"
fig=plt.show()
display(fig)
```



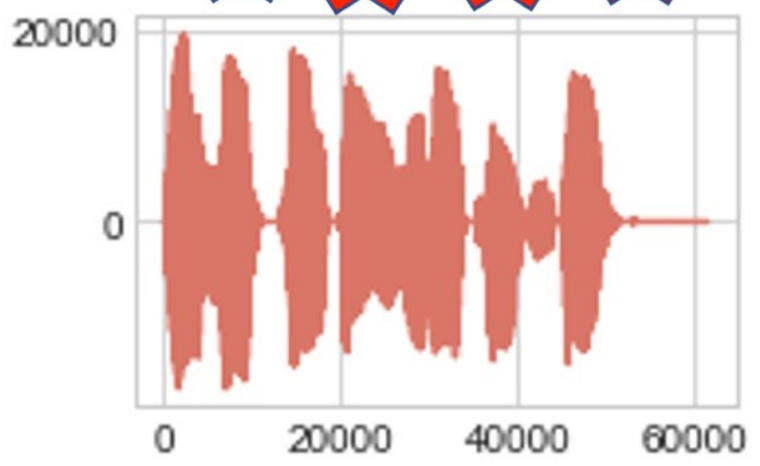
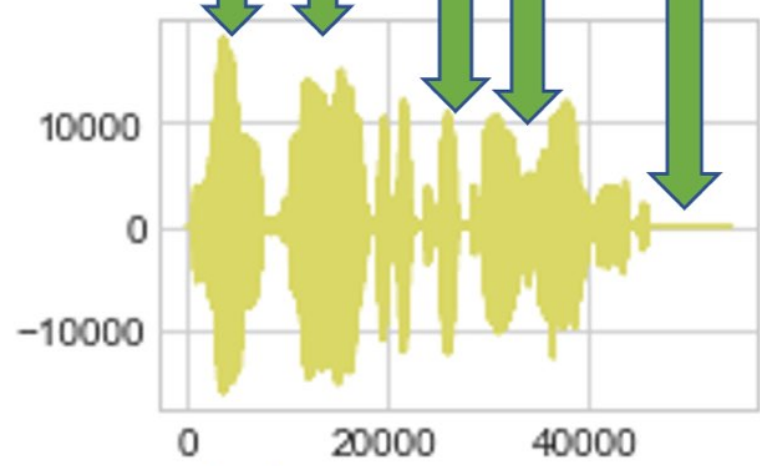
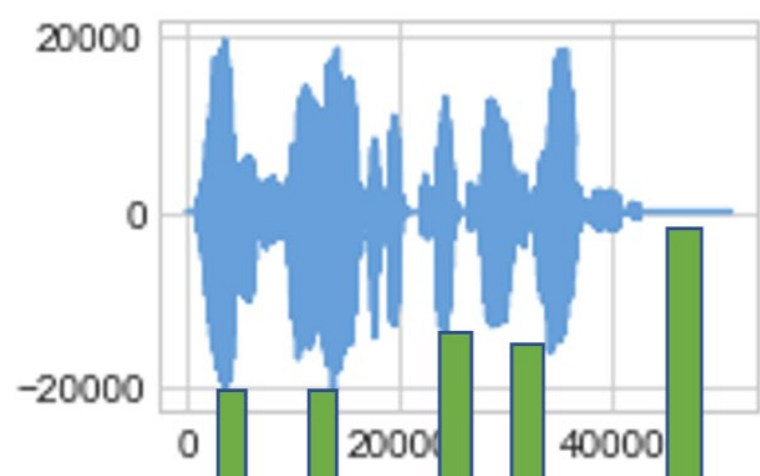


```
In [34]: M # Create subplots for third audio file for "I Love Python and I Can't Stop"
ax = plt.subplot(2, 2, 1)
ax.plot(data3, color="#da7667")

# Display created figure for third audio file for "I Love Python and I Can't Stop"
fig=plt.show()
display(fig)
```



As you can see from the three images above, the first two audio files have the same shape but with different amplitude and pacing because it is the same phrase ("One Flew Over the Cuckoo's Nest"). While the third file has a different shape as it is a different phrase altogether ("I Love Python and I Can't Stop"). This can be seen more easily if we stack the three images above each other and draw lines to the similarities:



First to compare the different audio files above, we will use the traditional Euclidean distance calculation. How does the traditional Euclidean distance calculation work you ask? In the simplest terms, it calculates the distance between any two points as a numerical difference between their coordinates.

So as an example, if a and b are two points on a line, then the distance between them is calculated as:  $(a-b)$  - this is assuming we are in a single dimensional plane. To extend this example to a two-dimensional space, assuming point a has coordinates  $(a_1, a_2)$  and point b has coordinates  $(b_1, b_2)$ , your formula will be  $\sqrt{(a_1-b_1)^2 + (a_2-b_2)^2}$ . In the formula by squaring and then taking a square root, it ensures that you get an absolute value and eliminate any negative values. You can easily extend this formula to n dimensions by having n number of coordinates but still squaring and then taking a square root:  $\sqrt{(a_1-b_1)^2 + (a_2-b_2)^2 + \dots + (a_n-b_n)^2}$ .

Python provides a simple function in numpy library called `linalg` and it calculates the difference between two vectors if they are represented as an array. Obviously, the problem with this approach is that it completely does not consider any time scale or latency differences. So, if we apply this formula to compare the first two wav files that both have the same phrase of "One Flew Over the Cuckoo's Nest".

**`np.linalg.norm(data1[0]-data2[0])`**

results in an output of 1.0 and so does comparing the first and third wav file:

**`np.linalg.norm(data1[0]-data3[0])`**

```
np.linalg.norm(data1[0]-data2[0])
Out[29]: 1.0
In [30]: #We get a result of 1.0 and if we compare one of the first two files with the third file that has the phrase "I Love Python"
np.linalg.norm(data1[0]-data3[0])
Out[30]: 1.0
```

We get the same result as the Euclidean vector difference is not taking the time axis and latency differences into account and therefore is not able to differentiate between the wav files. For time warping comparison, we will use the FastDTW PyPi library we installed and imported earlier. FastDTW compares the distance between the different sound files. Expectation is that the files that are a closer match will have a shorter distance as the FastDTW library will account for the different pacing and amplitude changes when calculating the distance.

Distance between audio file 1 and 2:  
**fastdtw(data1, data2)[0]** is 103331114.0

```
#Distance between audio file 1 and 2
fastdtw(data1, data2)[0]
Out[35]: 103331114.0
```

Distance between audio file 1 and 3:  
**fastdtw(data1, data3)[0]** is 110120061.0

```
In [36]: #Distance between audio file 1 and 3
fastdtw(data1, data3)[0]
Out[36]: 110120061.0
```

As expected, the first two audio files have a shorter distance because they are the same phrase spoken with a different voice, amplitude, and latency while the distance between the first and third file is longer because they are two completely different phrases.

There you go! A simple example to compare audio files and their similarities using Time Warping technique.



## USE CASE 6 – TIME SERIES ANALYSIS AND FORECASTING

**T**ime series forecasting is a very common use case in predictive analytics – specially in an operational and sales space where line managers or salespeople want to know based on historical sales patterns and seasonality how much product and staff, they should have on hand to meet the incoming demand. This is of course based on the assumption that past is a predictor of the future – that is not always the case when you have a crazy year like 2020 where the pandemic completely threw off several predictive models that rely on this assumption. Nevertheless, for the purposes of this chapter and for simplicity, we will stick with the assumption that past predicts the future.

We will also introduce a more sophisticated time series forecasting library called Prophet from Facebook that makes the more advanced calculations and traditional models like ARIMA and Kalman Filter easy to apply at scale without a masters in statistical analysis.

Now be warned that the Prophet library is still relatively new and must be fully tested in the market but does have promise. I found it slow in practice as we used it on a very small dataset of e-commerce orders from July 2018 to Dec 2019 from Kaggle public dataset to predict sales for 2020 (it will be available to you along with the code via a download link later in the book).

Facebook Prophet also is not the most straightforward library to install as it has several dependencies and requires



a C++ compiler before it installs successfully. You can follow the link below to complete the installation steps:

<https://facebook.github.io/prophet/docs/installation.html>

Once you have Facebook Prophet fully installed, you are ready to import the library and predict the future!

```
import pandas as pd  
from fbprophet import Prophet  
import warnings  
warnings.simplefilter(action='ignore',  
category=FutureWarning)
```

```
In [42]: import pandas as pd  
from fbprophet import Prophet  
import warnings  
warnings.simplefilter(action='ignore', category=FutureWarning)  
  
#read csv data into pandas  
data = pd.read_csv("ecommerce_data.csv")  
# Preview the first 5 lines of the Loaded data  
data.head()
```

First, we read csv data into panda data frame:

```
data = pd.read_csv("ecommerce_data.csv")
```

and preview the first 5 lines of the loaded data

```
data.head()
```

```
Out[42]:
```

	ds	product_id	city_id	y
0	7/14/2019	2981	26	3500
1	11/11/2019	2981	26	3000
2	6/13/2019	4254	16	2900
3	6/25/2019	5599	16	2400
4	7/10/2019	3622	16	3300

Let's initialize the Prophet model and set the parameters:

```
model = Prophet  
interval_width=0.95,  
growth="linear",  
daily_seasonality=False,
```

```
weekly_seasonality=False,  
yearly_seasonality=True,  
seasonality_mode="multiplicative"  
)
```

And fit the model to historical data (NOTE: This step takes a while to complete as the Python code executes – even on a relatively small dataset, so be patient).

```
model.fit(data)
```

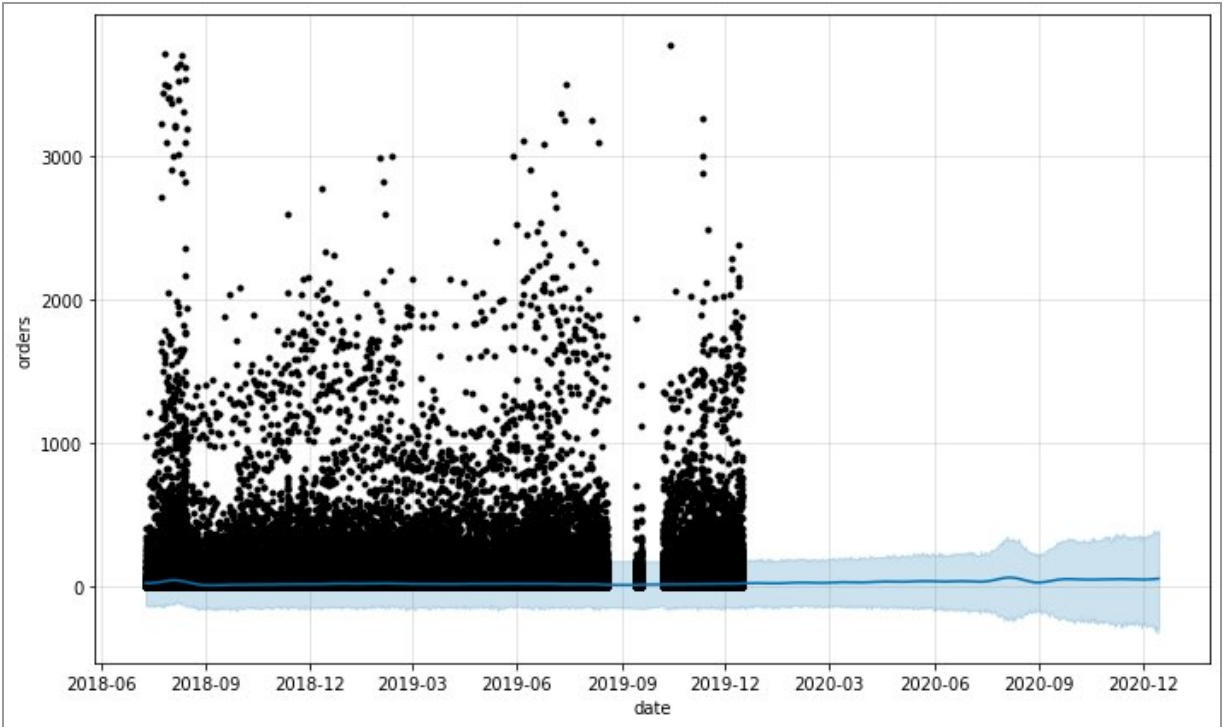
Finally, you will get an output like this to indicate that the model initialization and fitting is complete:

```
<fbprophet.forecaster.Prophet at 0x200373d6bc8>
```

```
In [38]: M #Let's initialize the model and set the parameters  
model = Prophet(  
    interval_width=0.95,  
    growth="linear",  
    daily_seasonality=False,  
    weekly_seasonality=False,  
    yearly_seasonality=True,  
    seasonality_mode="multiplicative"  
)  
# fit the model to historical data  
model.fit(data)  
  
Out[38]: <fbprophet.forecaster.Prophet at 0x200373d6bc8>
```

We are now ready to tell the model to predict 365 days (or 1 year) into the future and then plot the results by putting the date in the x-axis and e-commerce orders on the y-axis:

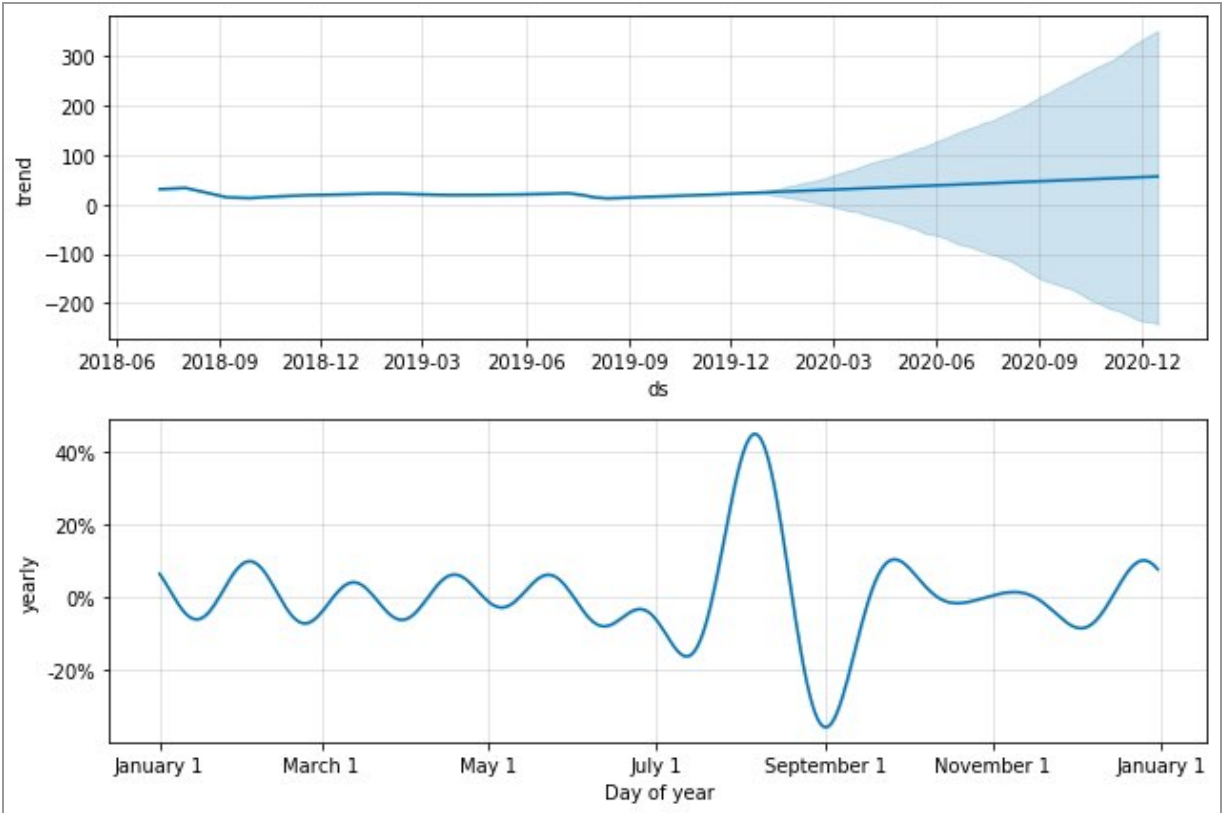
```
In [39]: M future_pd = model.make_future_dataframe(  
    periods=365,  
    freq="d",  
    include_history=True  
)  
# predict over the dataset  
forecast_pd = model.predict(future_pd)  
  
In [43]: M predict_fig = model.plot(forecast_pd, xlabel="date", ylabel="orders")
```



In the above visualization, the dots are actual values of e-commerce order per week and the trend line shows the future. The bands around the trend line show our uncertainty levels (in this case we set it to 95%).

We can also look at the forecast more closely by executing this command:

```
model.plot_components(forecast_pd)
```



The top visual shows our trend – which is increasing slightly as 2020 progresses and the bottom visual shows the seasonality with a huge variation between August to September.

How about that? Predicting the future with just a few lines of code – using the ever-evolving Facebook Prophet library to make the time series forecasting simpler to apply!



## USE CASE 7 – FRAUD ANALYSIS

**F**raud analysis is also a very common machine learning use case specially in the financial services industry. Data Scientists pour over millions of records of historical financial transaction data to determine rulesets that define fraudulent transactions and then build algorithms to detect and stop fraud in its tracks. With the fraudsters constantly changing their patterns the algorithm used by data scientists also must be flexible enough to be able to learn from the history and experience to adjust accordingly.

In the example in this chapter, we will use a public dataset for digital financial transactions through Kaggle. This data will be available to you along with the code in this chapter via a link later in the book.

Most common algorithms used for fraud analysis are typically decision trees or some variations there of including gradient boosting via XGBoost and the like. We will explore 6 million records of this dataset and then recommend an algorithm that yields the highest level of prediction accuracy.

We will start with importing libraries we will need for fraud analysis

```
import pandas as pd  
import numpy as np  
%matplotlib inline  
import matplotlib.pyplot as plt  
import matplotlib.lines as mlines  
import seaborn as sns
```

```

from sklearn.model_selection import train_test_split,
learning_curve
from sklearn.metrics import average_precision_score
from xgboost.sklearn import XGBClassifier
from xgboost import plot_importance, plot_tree
import warnings
warnings.simplefilter(action='ignore',
category=FutureWarning)
warnings.simplefilter(action='ignore',
category=DeprecationWarning)
warnings.simplefilter(action='ignore',
category=UserWarning)

```

```

In [1]: M #We will start with importing libraries we will likely need for fraud analysis
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.metrics import average_precision_score
from xgboost.sklearn import XGBClassifier
from xgboost import plot_importance, plot_tree
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=DeprecationWarning)
warnings.simplefilter(action='ignore', category=UserWarning)

```

We will now import the public fraud dataset I referenced earlier into a panda data-frame and rename mismatched column names in the data for consistency:

```

df = pd.read_csv('fraud_data.csv')
df = df.rename(columns={
'oldbalanceOrig':'oldBalanceOrig',
'newbalanceOrig':'newBalanceOrig',
'oldbalanceDest':'oldBalanceDest',
'newbalanceDest':'newBalanceDest'})
print(df.head())

```

```
In [2]: # Import the fraud dataset into a panda dataframe and rename mismatched column names
df = pd.read_csv('fraud_data.csv')
df = df.rename(columns={'oldBalanceOrig': 'oldBalanceOrig', 'newBalanceOrig': 'newBalanceOrig', \
                       'oldBalanceDest': 'oldBalanceDest', 'newBalanceDest': 'newBalanceDest'})
print(df.head())
```

step	type	amount	nameOrig	oldBalanceOrig	newBalanceOrig	nameDest	oldBalanceDest	newBalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.0	M1979787155	0.0	0.0	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.0	M2044282225	0.0	0.0	0	0
2	1	TRANSFER	181.00	C1305486145	181.0	C553264065	0.0	0.0	1	0
3	1	CASH_OUT	181.00	C840083671	181.0	C38997010	21182.0	0.0	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.0	M1230701703	0.0	0.0	0	0

We now do some data cleaning to check if the data has any null values:

```
df.isnull().values.any()
```

```
In [3]: # Check for any null values in the data
df.isnull().values.any()

Out[3]: False
```

The dataset does have a flag indicating whether a transaction was fraudulent or not and it correlates very well with the number of cash out transactions - basically implying that quick transfer ins and cash outs are a good early indicator of potential fraudulent transactions.

```
print('The types of fraudulent transactions are {}'.format(list(df.loc[df.isFraud == 1].type.drop_duplicates().values)))
```

```
dfFraudTransfer = df.loc[(df.isFraud == 1) & (df.type == 'TRANSFER')]
```

```
dfFraudCashout = df.loc[(df.isFraud == 1) & (df.type == 'CASH_OUT')]
```

```
print ('The number of fraudulent TRANSFERS = {}'.format(len(dfFraudTransfer)))
```



```
print ('The number of fraudulent CASH_OUTs =  
{:}'.format(len(dfFraudCashout)))
```

```
In [4]: #The dataset does have a flag indicating whether a transaction was fraudulent or not and interestingly enough it correlates  
#very well with the number of cash out transactions - basically implying that quick transfer ins and cash outs are a good  
#early indicator of potential fraudulent transactions  
  
print('The types of fraudulent transactions are {}'.format(list(df.loc[df.isFraud == 1].type.drop_duplicates().values)))  
  
dfFraudTransfer = df.loc[(df.isFraud == 1) & (df.type == 'TRANSFER')]  
dfFraudCashout = df.loc[(df.isFraud == 1) & (df.type == 'CASH_OUT')]  
  
print ('The number of fraudulent TRANSFERS = {}'.format(len(dfFraudTransfer)))  
print ('The number of fraudulent CASH_OUTs = {}'.format(len(dfFraudCashout))) |  
  
The types of fraudulent transactions are ['TRANSFER', 'CASH_OUT']  
  
The number of fraudulent TRANSFERS = 4097  
  
The number of fraudulent CASH_OUTs = 4116
```

Now let's clean our data further to narrow it down to the Transfer and Cash out type of transactions and discard meaningless or inconsistently populated columns like nameOrig, nameDest and isFlaggedFraud:

```
X = df.loc[(df.type == 'TRANSFER') | (df.type ==  
'CASH_OUT')]
```

```
randomState = 5
```

```
np.random.seed(randomState)
```

```
Y = X['isFraud']
```

```
del X['isFraud']
```

and eliminate columns shown to be irrelevant for analysis:

```
X = X.drop(['nameOrig', 'nameDest', 'isFlaggedFraud'],  
axis = 1)
```

Since most machine learning algorithms use numeric values for processing, we will binary-encode labelled data in transaction type column:

```
X.loc[X.type == 'TRANSFER', 'type'] = 0
```

```
X.loc[X.type == 'CASH_OUT', 'type'] = 1
```

## X.type = X.type.astype(int)

```
In [5]: # Now let's clean our data to narrow it down to the Transfer and Cash out type of transactions and discard meaningless
# or inconsistently populated columns like nameOrig, nameDest and isFlaggedFraud

X = df.loc[(df.type == 'TRANSFER') | (df.type == 'CASH_OUT')]

randomState = 5
np.random.seed(randomState)

Y = X['isFraud']
del X['isFraud']

# Eliminate columns shown to be irrelevant for analysis
X = X.drop(['nameOrig', 'nameDest', 'isFlaggedFraud'], axis = 1)

# Binary-encoding of Labelled data in 'type'
X.loc[X.type == 'TRANSFER', 'type'] = 0
X.loc[X.type == 'CASH_OUT', 'type'] = 1
X.type = X.type.astype(int)
```

Since the destination account having zero balance after the transfer in and cash out is an indicator of potential fraud, let's mark it more prominently so it is easier for our machine learning model to detect this type of final balance and we will create a specific feature around this point for model training – this will come into play later in this chapter:

```
X.loc[(X.oldBalanceDest == 0) & (X.newBalanceDest == 0) & (X.amount != 0), ['oldBalanceDest', 'newBalanceDest']] = - 1
```

```
X.loc[(X.oldBalanceOrig == 0) & (X.newBalanceOrig == 0) & (X.amount != 0), ['oldBalanceOrig', 'newBalanceOrig']] = np.nan
```

```
X['errorBalanceOrig'] = X.newBalanceOrig + X.amount - X.oldBalanceOrig
```

```
X['errorBalanceDest'] = X.oldBalanceDest + X.amount - X.newBalanceDest
```

```
In [6]: # Since the destination account having zero balance after the transfer in cash out, let's mark it more prominently
# so it is easier for our machine learning model to detect this type of final balance and we will create a specific
# feature around this point for model training - this will come into play later in this chapter

X.loc[(X.oldBalanceDest == 0) & (X.newBalanceDest == 0) & (X.amount != 0), ['oldBalanceDest', 'newBalanceDest']] = - 1
X.loc[(X.oldBalanceOrig == 0) & (X.newBalanceOrig == 0) & (X.amount != 0), ['oldBalanceOrig', 'newBalanceOrig']] = np.nan
X['errorBalanceOrig'] = X.newBalanceOrig + X.amount - X.oldBalanceOrig
X['errorBalanceDest'] = X.oldBalanceDest + X.amount - X.newBalanceDest
```

Now that we have cleaned the data and engineered the features that we think will result in actual fraud output, let's visualize our data to see if we can see fraudulent transactions more clearly:

Let's visualize our data to see if we can see fraudulent transactions more clearly:

```
limit = len(X)  
def plotStrip(x, y, hue, figsize = (15, 10)):
```

```
FIG = PLT.FIGURE(figsize = figsize)  
colours = plt.cm.tab10(np.linspace(1, 2, 8))  
with sns.axes_style('whitegrid'):  
ax = sns.stripplot(x, y, \  
hue = hue, jitter = 0.5, marker = '.', \  
size = 4, palette = 'colorblind')  
ax.set_xlabel('')  
ax.set_xticklabels(['regular tx', 'fraudulent tx'], size =  
18)  
for axis in ['top', 'bottom', 'left', 'right']:  
ax.spines[axis].set_linewidth(2)  
handles, labels = ax.get_legend_handles_labels()  
plt.legend(handles, ['Transfer', 'Cash out'],  
bbox_to_anchor=(1, 1), loc=2, borderaxespad=0, fontsize =  
15);  
return ax  
ax = plotStrip(Y[:limit], X.step[:limit], X.type[:limit])  
ax.set_ylabel('hours', size = 16)  
ax.set_title('Visualizing fraudulent transactions hidden  
in a sea of regular transactions', size = 25);
```

```

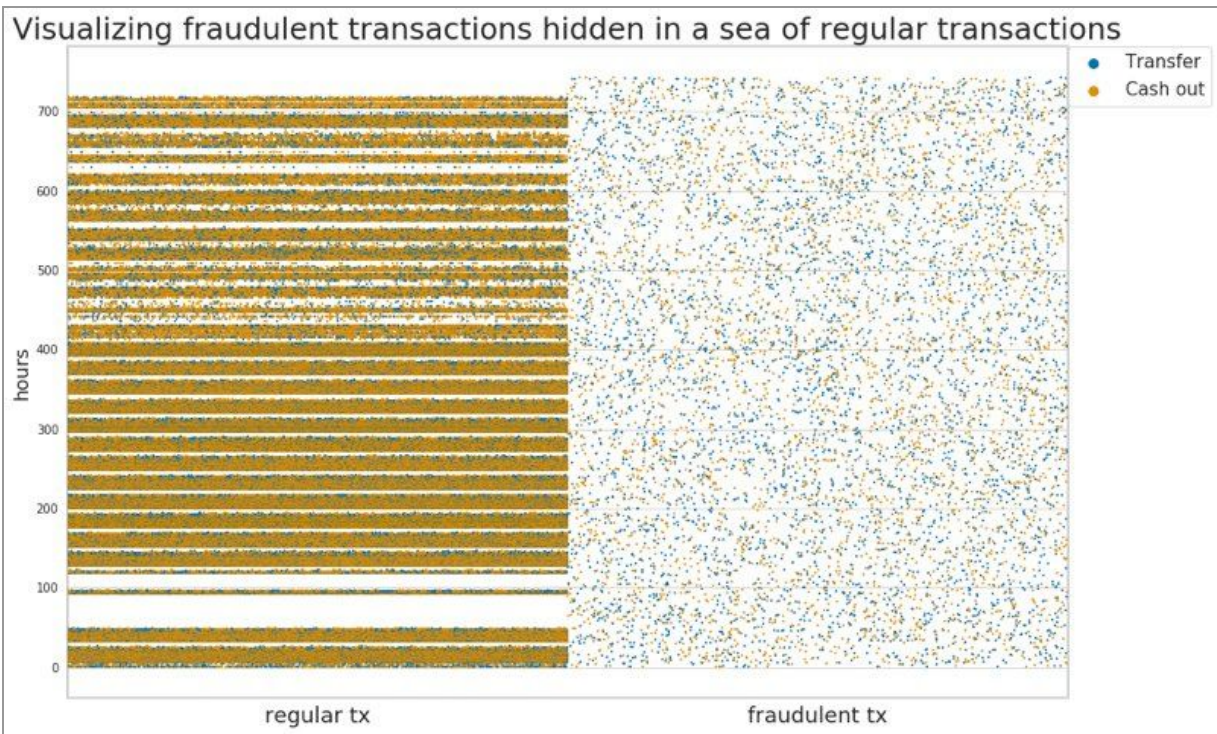
In [7]: #Now that we have cleaned the data and engineered the features that we think will result in actual fraud output,
#Let's visualize our data to see if we can see fraudulent transactions more clearly
limit = len(X)

def plotStrip(x, y, hue, figsize = (15, 10)):
    fig = plt.figure(figsize = figsize)
    colours = plt.cm.tab10(np.linspace(1, 2, 8))
    with sns.axes_style('whitegrid'):
        ax = sns.stripplot(x, y, \
            hue = hue, jitter = 0.5, marker = '.', \
            size = 4, palette = 'colorblind')
        ax.set_xlabel('')
        ax.set_xticklabels(['regular tx', 'fraudulent tx'], size = 18)
        for axis in ['top', 'bottom', 'left', 'right']:
            ax.spines[axis].set_linewidth(2)

        handles, labels = ax.get_legend_handles_labels()
        plt.legend(handles, ['Transfer', 'Cash out'], bbox_to_anchor=(1, 1), loc=2, borderaxespad=0, fontsize = 15);
    return ax

In [8]: ax = plotStrip(Y[:limit], X.step[:limit], X.type[:limit])
ax.set_ylabel('hours', size = 16)
ax.set_title('Visualizing fraudulent transactions hidden in a sea of regular transactions', size = 25);

```

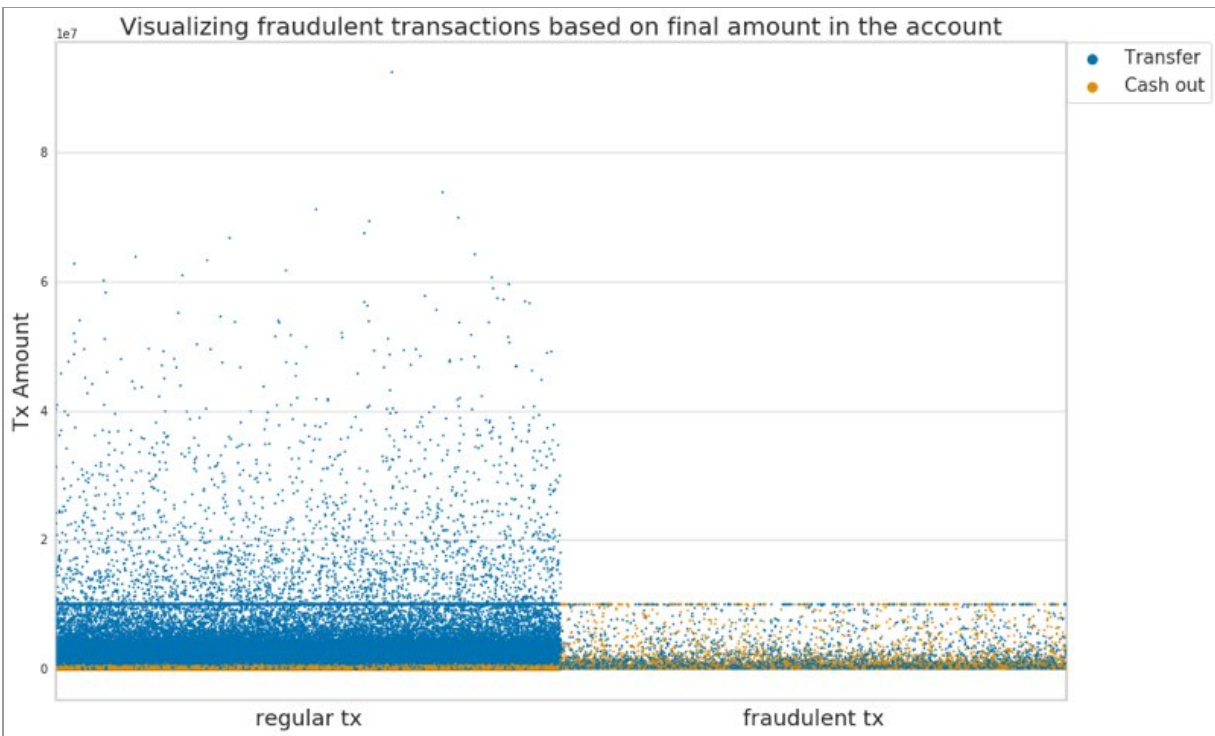


As can be seen from the above visualization, while we can differentiate the fraudulent from regular transactions, it is not as obvious. Let's use the feature we engineered based on final balance of the account being zero to make the fraudulent transactions more obvious:

**limit = len(X)**

```
ax = plotStrip(Y[:limit], X.amount[:limit], X.type[:limit],  
figsize = (15, 10))  
ax.set_ylabel('Tx Amount', size = 18)  
ax.set_title('Visualizing fraudulent transactions based  
on final amount in the account', size = 20);
```

```
In [9]: # As can be seen from the above visualization, while we can differentiate the fraudulent from regular transactions, it is  
# not as obvious. Let's use the feature we engineered based on final balance of the account being zero to make the fraudulent  
# transactions more obvious  
limit = len(X)  
ax = plotStrip(Y[:limit], X.amount[:limit], X.type[:limit], figsize = (15, 10))  
ax.set_ylabel('Tx Amount', size = 18)  
ax.set_title('Visualizing fraudulent transactions based on final amount in the account', size = 20);
```



As mentioned earlier in the chapter, let's use a variation of traditional decision tree algorithm called XGBoost which uses gradient boosting techniques to generally outperform other machine learning algorithms like random forest for fraud analysis by reducing model bias and providing better

accuracy. Let's split our dataset into 20% test and 80% training.

```
trainX, testX, trainY, testY = train_test_split(X, Y,
test_size = 0.2, random_state = randomState)
```

Now we run the XGBoost Classifier algorithm on this dataset and test the accuracy of the model:

```
weights = (Y == 0).sum() / (1.0 * (Y == 1).sum())
xgclf = XGBClassifier(max_depth = 3, scale_pos_weight
= weights, n_jobs = 4)
probabilities = xgclf.fit(trainX,
trainY).predict_proba(testX)
print('Average precision score = {}'.format(average_precision_score(testY, probabilities[:,
1])))
```

```
In [11]: #Now we run the XG Boost Classifier algorithm on this dataset and test the accuracy of the model
weights = (Y == 0).sum() / (1.0 * (Y == 1).sum())
xgclf = XGBClassifier(max_depth = 3, scale_pos_weight = weights, n_jobs = 4)
probabilities = xgclf.fit(trainX, trainY).predict_proba(testX)
print('Average precision score = {}'.format(average_precision_score(testY, probabilities[:, 1])))

[19:29:28] WARNING: ..\src\learner.cc:1061: Starting in XGBoost 1.3.0, the default evaluation metric used with the objective
'binary:logistic' was changed from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore the old behavior.
Average precision score = 0.9983642588456605
```

The average prediction score in the output above was 99.8%, which is very great! Now that we have confirmed that the XGBoost machine learning algorithm produces the most accurate fraud prediction, let's find out which feature in the dataset was the most important for the classification and splitting the tree:

```
fig = plt.figure(figsize = (15, 10))
ax = fig.add_subplot(111)
colours = plt.cm.Set2(np.linspace(1, 2, 8))
ax = plot_importance(xgclf, height = 1, color = colours,
grid = True, \
```

```

show_values = False, importance_type = 'cover', ax =
ax);
for axis in ['top', 'bottom', 'left', 'right']:
ax.spines[axis].set_linewidth(2)
ax.set_xlabel('Importance Value', size = 18);
ax.set_ylabel('Features', size = 18);
ax.set_yticklabels(ax.get_yticklabels(), size = 15);
ax.set_title('Ranking features in order of importance',
size = 18);

```

```

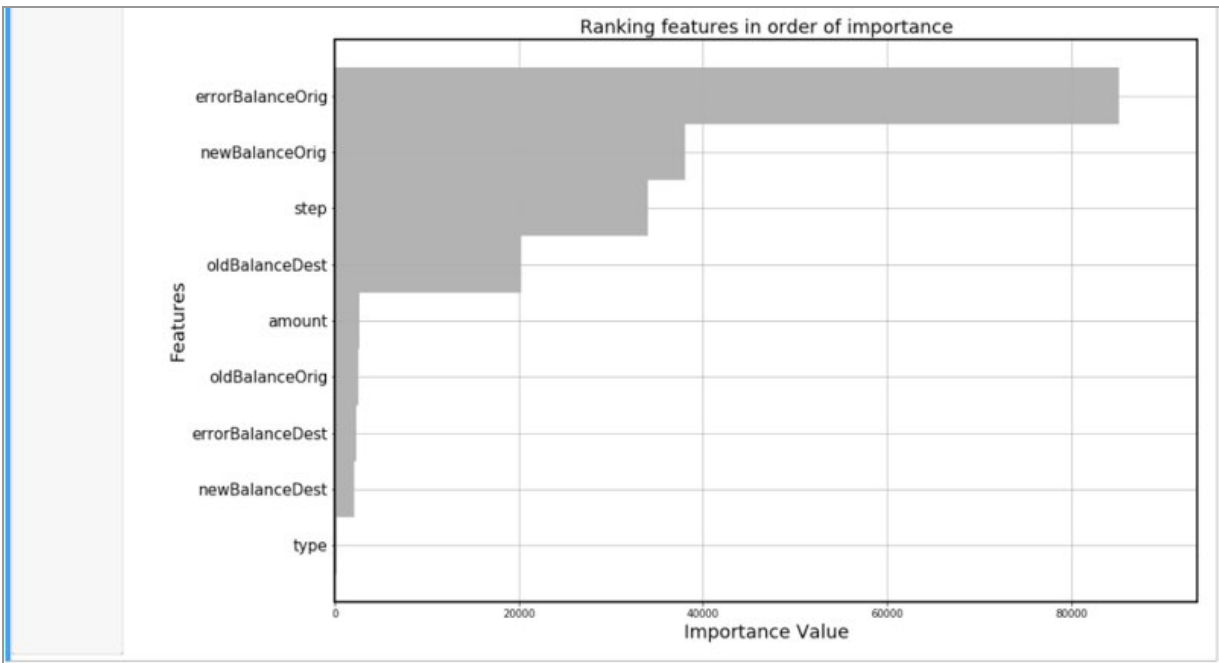
In [12]: #Now that we have confirmed that the XGBoost machine learning algorithm produces the most accurate fraud prediction,
#Let's find out which feature in the dataset was the most important for the classificatin and splitting the tree
fig = plt.figure(figsize = (15, 10))
ax = fig.add_subplot(111)

colours = plt.cm.Set2(np.linspace(1, 2, 8))

ax = plot_importance(xgclf, height = 1, color = colours, grid = True, \
                    show_values = False, importance_type = 'cover', ax = ax);
for axis in ['top', 'bottom', 'left', 'right']: ax.spines[axis].set_linewidth(2)

ax.set_xlabel('Importance Value', size = 18);
ax.set_ylabel('Features', size = 18);
ax.set_yticklabels(ax.get_yticklabels(), size = 15);
ax.set_title('Ranking features in order of importance', size = 18);

```



Not surprisingly our engineered feature based on final zero balance because of fraudulent cash outs is the most important feature for the fraud model. There you have it folks - a high level overview of how to analyze fraud in a dataset. Typically, the datasets don't come with fraud flags, and you have a lot more history to work with to determine rules to detect fraud patterns - but the steps we used in this example to narrow down list of features, visualize their impact and then train the model to generate a precise decision tree-based algorithm are all very much applicable in the real world!





## USE CASE 8 – PROCESSING GEOSPATIAL DATA

Processing geographical and geospatial data for intelligent automation is becoming more and more mainstream with the advent of intelligent drones and autonomous vehicles capable of detecting and moving around obstacles. In addition, geographical and demographic analysis is also becoming quite common when you are looking at opportunity assessment for new businesses.

Rome is the eternal city and my favorite city in Europe. I have a lot of fond memories from it when I went back packing through it with my university friends back in the day. Given that it has been so hard hit in recent times economically, I want to pay tribute to it by using it as part of this book.

Given that Rome is such a popular tourist destination and has so much history, it has expensive real estate as well as population density. With the impacts to its economy, investors can be looking at boroughs of Rome that have a high population and relatively lower real estate prices. In addition, using [FourSquare](#) data, I will also look at type of businesses in each borough to be able to recommend the best Rome neighborhood to start a business in and type of business to start based on the real estate prices and population density.

For this analysis, I used the following data sources:

- I obtained the real estate prices by different neighborhoods of Rome using the [statista.com](#) site that contains data as of December, 2019

[\[https://www.statista.com/statistics/670698/asking-price-for-properties-for-sale-in-rome-by-area-italy/\]](https://www.statista.com/statistics/670698/asking-price-for-properties-for-sale-in-rome-by-area-italy/).

- I obtained the .json file for Rome from carto.com site that will help us create the choropleth map of its neighborhoods  
[\[https://maurizioman.carto.com/tables/rome\\_admin/public/map\]](https://maurizioman.carto.com/tables/rome_admin/public/map).
- I used **Foursquare API** to get the most common venues of given Borough of Rome  
[\[https://foursquare.com/\]](https://foursquare.com/).

As a database, I created the dataset of Rome boroughs by populating the neighborhood names, real estate prices and geographic coordinates. I then saved it in the pandas' dataframe which has the following columns Borough, Average House Price, Latitude and Longitude.

```
import pandas as pd  
import numpy as np  
import requests
```

```
URL = 'ROME_GEO.CSV'  
df = pd.read_csv(url)  
df.head()
```

```
In [107]: import pandas as pd
import numpy as np
import requests

url = 'rome_geo.csv'

df = pd.read_csv(url)
df.head()
```

Out[107]:

	Borough	Avg-HousePrice	Latitude	Longitude
0	CentroStorico	7817	41.8982	12.4773
1	Caracalla	6910	41.8794	12.4931
2	Flaminio	5622	42.1919	12.4725
3	Trastevere	5435	41.8848	12.4704
4	Della Vittoria	5137	41.9182	12.4639

Since our geospatial data is going to be in the form of JSON files, we will import the json library to read the file and matplotlib library to visualize the data:

```
pd.set_option('display.max_columns', None)
```

```
pd.set_option('display.max_rows', None)
```

We first import and transform JSON file into a pandas dataframe:

```
import json
```

```
from pandas.io.json import json_normalize
```

We then import Matplotlib and associated plotting modules:

```
import matplotlib.cm as cm
```

```
import matplotlib.colors as colors
```

We will also import k-means machine learning algorithm library for clustering analysis:

```
from sklearn.cluster import KMeans
```

```
print('Libraries imported.')
```

```
In [2]: M pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)

import json # Library to handle JSON files

from pandas.io.json import json_normalize # transform JSON file into a pandas dataframe

# Matplotlib and associated plotting modules
import matplotlib.cm as cm
import matplotlib.colors as colors

# import k-means from clustering stage
from sklearn.cluster import KMeans

print('Libraries imported.')

Libraries imported.
```

In addition, we will install and import external libraries for mapping and clustering including GeoPy Nominatum, Folium and FourSquare:

```
!conda install -c conda-forge geopy --yes
from geopy.geocoders import Nominatim
!conda install -c conda-forge folium=0.7.0 --yes
import folium
```

```
In [4]: M !conda install -c conda-forge geopy --yes
from geopy.geocoders import Nominatim

!conda install -c conda-forge folium=0.7.0 --yes |
import folium # map rendering library
```

You can read more about GeoPy Nominatum library here:  
<https://geopy.readthedocs.io/en/stable/>

I also encourage you to read more about Folium library here. It is a powerful map rendering library:

<http://python-visualization.github.io/folium/>

We will now use geolocator library to get the latitude and longitude values of Rome:

```
address = 'Rome, IT'
geolocator = Nominatim()
location = geolocator.geocode(address)
latitude = location.latitude
longitude = location.longitude
```

```
print('The geographical coordinate of Rome are {},  
{},'.format(latitude, longitude))
```

```
In [5]: address = 'Rome, IT'  
geolocator = Nominatim()  
location = geolocator.geocode(address)  
latitude = location.latitude  
longitude = location.longitude  
print('The geographical coordinate of Rome are {}, {}'.format(latitude, longitude))
```

```
The geographical coordinate of Rome are 41.8933203, 12.4829321.
```

Now we will create a map of Rome with its boroughs using these latitude and longitude values:

```
map_rome = folium.Map(location=[latitude, longitude],  
zoom_start=9.5)
```

Code below adds markers to the map:

```
for lat, lng, borough in zip(df['Latitude'],  
df['Longitude'], df['Borough']):
```

```
label = '{}'.format(borough)
```

```
label = folium.Popup(label, parse_html=True)
```

```
folium.CircleMarker(  
[lat, lng],
```

```
radius=5,
```

```
popup=label,
```

```
color='blue',
```

```
fill=True,
```

```
fill_color='#3186cc',
```

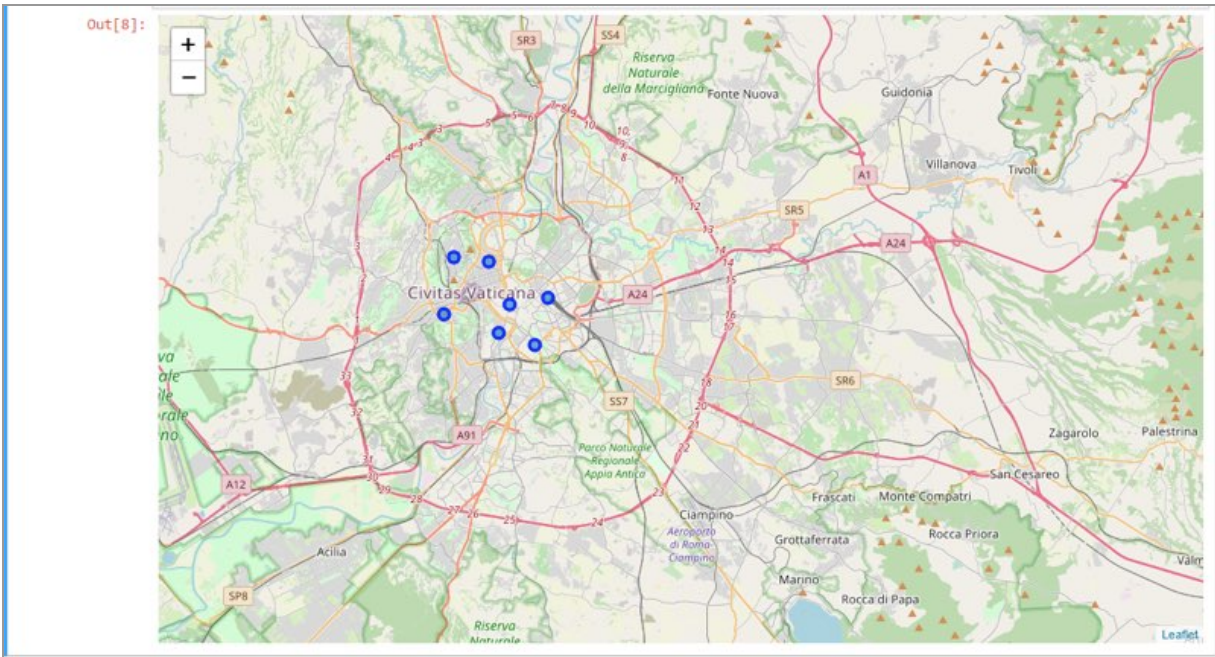
```
fill_opacity=0.7).add_to(map_rome)
```

```
map_rome
```

```
In [8]: map_rome = folium.Map(location=[latitude, longitude], zoom_start=9.5)

# add markers to map
for lat, lng, borough in zip(df['Latitude'], df['Longitude'], df['Borough']):
    label = '{}'.format(borough)
    label = folium.Popup(label, parse_html=True)
    folium.CircleMarker(
        [lat, lng],
        radius=5,
        popup=label,
        color='blue',
        fill=True,
        fill_color='#3186cc',
        fill_opacity=0.7).add_to(map_rome)

map_rome
```



We will now use the FourSquare API to explore the boroughs further. FourSquare collects and presents information about neighborhoods and businesses in the area and conveniently makes it available via an API. For more details, I encourage you to visit:

<https://developer.foursquare.com/>

```
rome_data = df
borough_latitude = rome_data.loc[0, 'Latitude']
borough_longitude = rome_data.loc[0, 'Longitude']
borough_name = rome_data.loc[0, 'Borough']
```

```
print('Latitude and longitude values of {} are {}, {},  
{},'.format(borough_name,  
borough_latitude,  
borough_longitude))
```

```
In [10]: M rome_data = df  
  
borough_latitude = rome_data.loc[0, 'Latitude'] # neighborhood latitude value  
borough_longitude = rome_data.loc[0, 'Longitude'] # neighborhood longitude value  
  
borough_name = rome_data.loc[0, 'Borough'] # neighborhood name  
print('Latitude and longitude values of {} are {}, {}'.format(borough_name,  
borough_latitude,  
borough_longitude))  
  
Latitude and longitude values of CentroStorico are 41.8982, 12.4773.
```

Now let's delve deeper and look at 100 businesses in the 750-meter radius around the boroughs:

```
LIMIT = 100  
radius = 750  
url = 'https://api.foursquare.com/v2/venues/explore?  
&client_id={}&client_secret={}&v={}&ll={},{}&radius=  
{}&limit={}'.format(  
CLIENT_ID,  
CLIENT_SECRET,  
VERSION,  
borough_latitude,  
borough_longitude,  
radius,  
LIMIT)  
url
```

```
In [11]: M LIMIT = 100  
radius = 750  
url = 'https://api.foursquare.com/v2/venues/explore?&client_id={}&client_secret={}&v={}&ll={},{}&radius={}&limit={}'.format(  
CLIENT_ID,  
CLIENT_SECRET,  
VERSION,  
borough_latitude,  
borough_longitude,  
radius,  
LIMIT)  
url]
```



We will also define a function that makes it easy to extract the category details of each venue in the surrounding neighborhoods:

```
def get_category_type(row):  
    try:  
        categories_list = row['categories']  
    except:  
        categories_list = row['venue.categories']
```

```
if len(categories_list) == 0:  
    return None  
else:  
    return categories_list[0]['name']
```

```
In [13]: # function that extracts the category of the venue  
def get_category_type(row):  
    try:  
        categories_list = row['categories']  
    except:  
        categories_list = row['venue.categories']  
  
    if len(categories_list) == 0:  
        return None  
    else:  
        return categories_list[0]['name']
```

Now we are ready to clean the json and structure into a pandas data-frame:

```
venues = results['response']['groups'][0]['items']
```

First, we flatten the json file and normalize its contents:

```
nearby_venues = json_normalize(venues)
```

We then filter the columns in the file:

```
filtered_columns = ['venue.name', 'venue.categories',  
'venue.location.lat', 'venue.location.lng']  
nearby_venues = nearby_venues.loc[:,  
filtered_columns]
```

We then filter by category in each row:

```
nearby_venues['venue.categories'] =  
nearby_venues.apply(get_category_type, axis=1)
```

Finally, we clean the columns to present them on the screen:

```
nearby_venues.columns = [col.split(".")[1] for col in  
nearby_venues.columns]  
nearby_venues.head(20)
```

```
In [111]: M venues = results['response']['groups'][0]['items']  
  
nearby_venues = json_normalize(venues) # flatten JSON  
  
# filter columns  
filtered_columns = ['venue.name', 'venue.categories', 'venue.location.lat', 'venue.location.lng']  
nearby_venues = nearby_venues.loc[:, filtered_columns]  
  
# filter the category for each row  
nearby_venues['venue.categories'] = nearby_venues.apply(get_category_type, axis=1)  
  
# clean columns  
nearby_venues.columns = [col.split(".")[1] for col in nearby_venues.columns]  
|  
nearby_venues.head(20)  
  
Out[111]:
```

	name	categories	lat	lng
0	Pantheon	Monument / Landmark	41.899133	12.476805
1	Pizza e Mozzarella	Pizza Place	41.897598	12.479097
2	Piazza della Rotonda	Plaza	41.899253	12.476779
3	Antica Salumeria	Sandwich Place	41.899209	12.476511
4	Tazza d'Oro	Coffee Shop	41.899435	12.477359
5	Capranica Enoteca & Taverna	Italian Restaurant	41.899989	12.477762
6	Bartolucci	Toy / Game Store	41.899820	12.478812
7	Fiocco Di Neve at the Pantheon	Ice Cream Shop	41.899885	12.476702
8	Basilica di Santa Maria sopra Minerva	Church	41.897892	12.477572
9	Albergo Del Senato	Hotel	41.899186	12.476948
10	Il Panino Ingegnoso	Sandwich Place	41.899982	12.479195
11	La Ciambella	Restaurant	41.896867	12.476912
12	Piazza della Minerva	Plaza	41.897913	12.477421
13	Vini & Cucina	Pizza Place	41.896840	12.476409
14	Armando al Pantheon	Italian Restaurant	41.898995	12.476243

Let's expand our search and create a function to get all the boroughs in Rome:

```
def getNearbyVenues(names, latitudes, longitudes,  
radius=500, LIMIT=100):
```

```
venues_list=[]
```

```
for name, lat, lng in zip(names, latitudes, longitudes):
```

```
print(name)
```

First, we create the API request URL:

```
url = 'https://api.foursquare.com/v2/venues/explore?&client_id={}&client_secret={}&v={}&ll={},{}&radius={}&limit={}'.format(  
    CLIENT_ID,  
    CLIENT_SECRET,  
    VERSION,  
    lat,  
    lng,  
    radius,  
    LIMIT)
```

We then make the GET request:

```
results = requests.get(url).json()["response"]["groups"]  
[0]["items"]
```

Finally, we return only relevant information for each nearby venue:

```
venues_list.append([(  
    name,  
    lat,  
    lng,  
    v['venue']['name'],  
    v['venue']['location']['lat'],  
    v['venue']['location']['lng'],  
    v['venue']['categories'][0]['name']) for v in results])  
nearby_venues = pd.DataFrame([item for venue_list in  
venues_list for item in venue_list])  
nearby_venues.columns = ['Borough',  
    'Borough Latitude',  
    'Borough Longitude',
```

```

'Venue',
'Venue Latitude',
'Venue Longitude',
'Venue Category']
return(nearby_venues)

```

```

In [16]: def getNearbyVenues(names, latitudes, longitudes, radius=500, LIMIT=100):

    venues_list=[]
    for name, lat, lng in zip(names, latitudes, longitudes):
        print(name)

        # create the API request URL
        url = 'https://api.foursquare.com/v2/venues/explore?&client_id={}&client_secret={}&v={}&ll={},{}&radius={}&limit={}'.format(
            CLIENT_ID,
            CLIENT_SECRET,
            VERSION,
            lat,
            lng,
            radius,
            LIMIT)

        # make the GET request
        results = requests.get(url).json()["response"]["groups"][0]["items"]

        # return only relevant information for each nearby venue
        venues_list.append([
            name,
            lat,
            lng,
            v['venue']['name'],
            v['venue']['location']['lat'],
            v['venue']['location']['lng'],
            v['venue']['categories'][0]['name'] for v in results])

    nearby_venues = pd.DataFrame([item for venue_list in venues_list for item in venue_list])
    nearby_venues.columns = ['Borough',
                            'Borough Latitude',
                            'Borough Longitude',
                            'Venue',
                            'Venue Latitude',
                            'Venue Longitude',
                            'Venue Category']

    return(nearby_venues)

```

Running the above function on each borough and creating a new dataframe called rome\_venues:

```

rome_venues =
getNearbyVenues(names=rome_data['Borough'],
                 latitudes=rome_data['Latitude'],
                 longitudes=rome_data['Longitude']
                 )

```

```
In [17]: M rome_venues = getNearbyVenues(names=rome_data['Borough'],
                                         latitudes=rome_data['Latitude'],
                                         longitudes=rome_data['Longitude']
                                         )

CentroStorico
Caracalla
Flaminio
Trastevere
Della Vittoria
Trieste
Bologna
CorsoFrancia
Termini
Camilluccia
Balduina
GregorioVII
```

Checking the size of the resulting data-frame:

**print(rome\_venues.shape)**

**rome\_venues.head()**

```
In [18]: M print(rome_venues.shape)
         rome_venues.head()

(488, 7)

Out[18]:
```

	Borough	Borough Latitude	Borough Longitude	Venue	Venue Latitude	Venue Longitude	Venue Category
0	CentroStorico	41.8982	12.4773	Pantheon	41.899133	12.476805	Monument / Landmark
1	CentroStorico	41.8982	12.4773	Pizza e Mozzarella	41.897598	12.479097	Pizza Place
2	CentroStorico	41.8982	12.4773	Piazza della Rotonda	41.899253	12.476779	Plaza
3	CentroStorico	41.8982	12.4773	Antica Salumeria	41.899209	12.476511	Sandwich Place
4	CentroStorico	41.8982	12.4773	Tazza d'Oro	41.899435	12.477359	Coffee Shop

Checking how many venues were returned for each borough:

```
summary
=
rome_venues.groupby('Borough').count().reset_index()
summary['Count'] = summary['Venue']
summary = summary.drop(['Borough Latitude',
'Borough Longitude', 'Venue', 'Venue Latitude', 'Venue
Longitude', 'Venue Category'], axis=1)
summary
=
summary.sort_values('Count').reset_index(drop=True)
summary.head(12)
```

```
In [109]: summary = rome_venues.groupby('Borough').count().reset_index()
summary['Count'] = summary['Venue']
summary = summary.drop(['Borough Latitude', 'Borough Longitude', 'Venue', 'Venue Latitude', 'Venue Longitude', 'Venue Category'])
summary = summary.sort_values('Count').reset_index(drop=True)
summary.head(12)
```

Out[109]:

	Borough	Count
0	Camilluccia	1
1	Caracalla	11
2	Balduina	22
3	GregorioVII	25
4	Della Vittoria	35
5	Bologna	45
6	CorsoFrancia	45
7	Trastevere	51
8	Termini	56
9	Trieste	97
10	CentroStorico	100

Creating a bar chart and analyzing the results:  
**import matplotlib.pyplot as plt; plt.rcParams()**  
**import numpy as np**  
**import matplotlib.pyplot as plt**

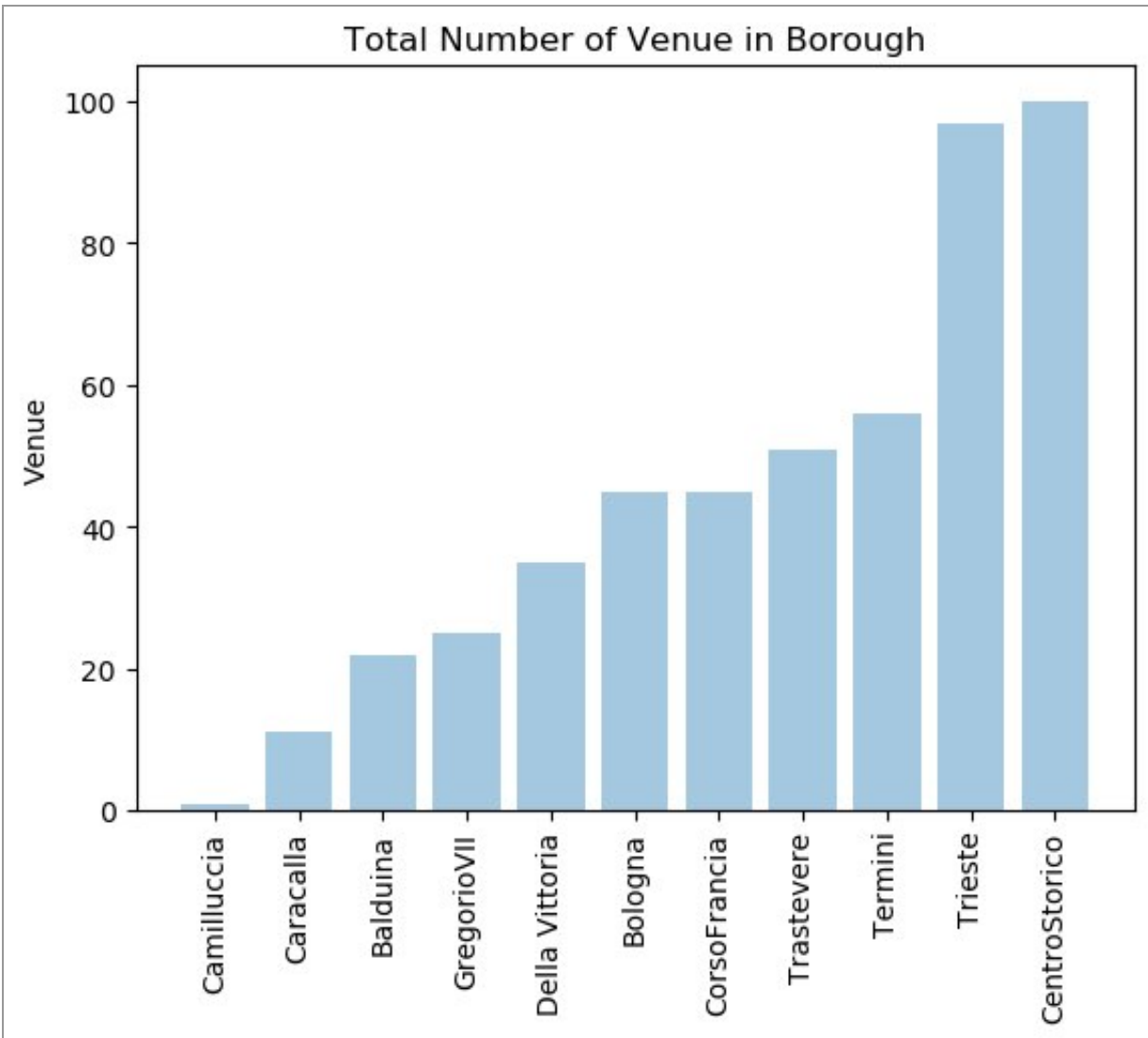
```
OBJECTS = summary.Borough  
y_pos = np.arange(len(objects))  
performance = summary.Count  
plt.bar(y_pos, performance, align='center', alpha=0.4)  
plt.xticks(y_pos, objects)  
plt.ylabel('Venue')  
plt.title('Total Number of Venue in Borough')  
plt.xticks(rotation=90)  
plt.show()
```

```
In [112]: import matplotlib.pyplot as plt; plt.rcParams()
import numpy as np
import matplotlib.pyplot as plt

objects = summary.Borough
y_pos = np.arange(len(objects))
performance = summary.Count

plt.bar(y_pos, performance, align='center', alpha=0.4)
plt.xticks(y_pos, objects)
plt.ylabel('Venue')
plt.title('Total Number of Venue in Borough')
plt.xticks(rotation=90)

plt.show()
```



The above bar chart shows us that Centro Storico and Trieste have close to 100 venues, followed by Tremini, Trastavere, Corso Francia, Della Vittoria and Bologna that

have venues in 40-60 range. Remaining boroughs are less venue rich like Georgio VII, Balduina, Caracalla and Camillucia. Camillucia specially seems low in venues and potentially ripe for further investment.

Let's find out how many unique categories can be curated from all the returned venues:

```
print('There are {} uniques categories.'.format(len(rome_venues['Venue Category'].unique())))
```

```
In [22]: print('There are {} uniques categories.'.format(len(rome_venues['Venue Category'].unique())))\nThere are 101 uniques categories.
```

We will now analyze each borough along with the venues that exist there and do some data cleaning in the process by using one hot encoding to show a 1 where a venue category exists in a neighborhood and 0 where it does not exist:

We first apply one hot encoding for venue categorization:

```
rome_onehot = pd.get_dummies(rome_venues[['Venue Category']], prefix="", prefix_sep="")
```

We then add neighborhood column back to dataframe:

```
rome_onehot['Borough'] = rome_venues['Borough']
```

We also move neighborhood column to the first column:

```
list_column = rome_onehot.columns.tolist()\nnumber_column = int(list_column.index('Borough'))\nlist_column = [list_column[number_column]] +\nlist_column[:number_column] +\nlist_column[number_column+1:]\nrome_onehot = rome_onehot[list_column]\nrome_onehot.head(20)
```



```
In [25]: # one hot encoding
rome_onehot = pd.get_dummies(rome_venues[['Venue Category']], prefix="", prefix_sep="")

# add neighborhood column back to dataframe
rome_onehot['Borough'] = rome_venues['Borough']

# move neighborhood column to the first column
list_column = rome_onehot.columns.tolist()
number_column = int(list_column.index('Borough'))
list_column = [list_column[number_column]] + list_column[:number_column] + list_column[number_column+1:]
rome_onehot = rome_onehot[list_column]

rome_onehot.head(20)
```

Out[25]:

	Borough	Argentinian Restaurant	Art Gallery	Art Museum	Asian Restaurant	Bagel Shop	Bakery	Bar	Basketball Court	Bed & Breakfast	Beer Bar	Bistro	Bookstore	Boutique	Brazilian Restaurant	Br
0	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	CentroStorico	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Let's determine the frequency of occurrence of each venue category in the different boroughs:

```
rome_grouped = rome_onehot.groupby('Borough').mean().reset_index()
rome_grouped.head()
```

Out[26]:

	Borough	Argentinian Restaurant	Art Gallery	Art Museum	Asian Restaurant	Bagel Shop	Bakery	Bar	Basketball Court	Bed & Breakfast	Beer Bar	Bistro	Bookstore	Boutique	Bra Resta
0	Balduina	0.00	0.00	0.00	0.045455	0.0	0.000000	0.000000	0.000000	0.0	0.000000	0.00	0.00	0.00	0.0
1	Bologna	0.00	0.00	0.00	0.022222	0.0	0.044444	0.088889	0.022222	0.0	0.044444	0.00	0.00	0.00	0.0
2	Camilliccia	0.00	0.00	0.00	0.000000	0.0	0.000000	0.000000	0.000000	0.0	0.000000	0.00	0.00	0.00	0.0
3	Caracalla	0.00	0.00	0.00	0.000000	0.0	0.000000	0.000000	0.000000	0.0	0.000000	0.00	0.00	0.00	0.0
4	CentroStorico	0.01	0.01	0.01	0.000000	0.0	0.010000	0.000000	0.000000	0.0	0.000000	0.01	0.01	0.01	0.0

Creating a function to sort the venues in descending order:

```
def return_most_common_venues(row, num_top_venues):
    row_categories = row.iloc[1:]
    row_categories_sorted = row_categories.sort_values(ascending=False)
    return row_categories_sorted.index.values[0:num_top_venues]
```

```
In [28]: M def return_most_common_venues(row, num_top_venues):
          row_categories = row.iloc[1:]
          row_categories_sorted = row_categories.sort_values(ascending=False)
          return row_categories_sorted.index.values[0:num_top_venues]
```

Now let's create the new data-frame and display the top 10 venues for each neighborhood:

```
num_top_venues = 10
indicators = ['st', 'nd', 'rd']
```

We also create columns according to number of top venues:

```
columns = ['Borough']
for ind in np.arange(num_top_venues):
    try:
        columns.append('{}{}{} Most Common Venue'.format(ind+1, indicators[ind]))
    except:
        columns.append('{}th Most Common Venue'.format(ind+1))
```

We create a new dataframe with venues and boroughs sorted:

```

boroughs_venues_sorted =
pd.DataFrame(columns=columns)
boroughs_venues_sorted['Borough'] =
rome_grouped['Borough']
for ind in np.arange(rome_grouped.shape[0]):
boroughs_venues_sorted.iloc[ind, 1:] =
return_most_common_venues(rome_grouped.iloc[ind, :],
num_top_venues)
boroughs_venues_sorted.head(12)

```

```

In [113]: num_top_venues = 10

indicators = ['st', 'nd', 'rd']

# create columns according to number of top venues
columns = ['Borough']
for ind in np.arange(num_top_venues):
    try:
        columns.append('{} Most Common Venue'.format(ind+1, indicators[ind]))
    except:
        columns.append('{}th Most Common Venue'.format(ind+1))

# create a new dataframe
boroughs_venues_sorted = pd.DataFrame(columns=columns)
boroughs_venues_sorted['Borough'] = rome_grouped['Borough']

for ind in np.arange(rome_grouped.shape[0]):
    boroughs_venues_sorted.iloc[ind, 1:] = return_most_common_venues(rome_grouped.iloc[ind, :], num_top_venues)

boroughs_venues_sorted.head(12)

```

Out[113]:

	Borough	1st Most Common Venue	2nd Most Common Venue	3rd Most Common Venue	4th Most Common Venue	5th Most Common Venue	6th Most Common Venue	7th Most Common Venue	8th Most Common Venue	9th Most Common Venue	10th Most Common Venue
0	Balduina	Italian Restaurant	Hotel	Spa	Coffee Shop	Chinese Restaurant	Department Store	Pizza Place	Plaza	Pool	Lounge
1	Bologna	Italian Restaurant	Bar	Pizza Place	Restaurant	Beer Bar	Bakery	Ice Cream Shop	Supermarket	Café	Trattoria/Osteria
2	Camiliuccia	Nightclub	Wine Shop	Flea Market	Coffee Shop	Convenience Store	Deli / Bodega	Department Store	Dessert Shop	Diner	Eastern European Restaurant
3	Caracalla	Park	Performing Arts Venue	Plaza	Garden	Italian Restaurant	Festival	Nightclub	Opera House	Historic Site	Track Stadium
4	CentroStorico	Italian Restaurant	Plaza	Ice Cream Shop	Hotel	Sandwich Place	Fountain	Restaurant	Monument / Landmark	Pizza Place	Church
5	CorsoFrancia	Italian Restaurant	Café	Pizza Place	Plaza	Sushi Restaurant	Piedmontese Restaurant	Hotel	Piadineria	Cafeteria	Campanian Restaurant
6	Della Vittoria	Italian Restaurant	Café	Pizza Place	Plaza	Breakfast Spot	Brazilian Restaurant	Hotel	Food Court	Ice Cream Shop	Convenience Store
7	GregorioVII	Pizza Place	Hotel	Café	Breakfast Spot	Italian Restaurant	Seafood Restaurant	Flower Shop	Dessert Shop	Plaza	Fast Food Restaurant
8	Termini	Hotel	Italian Restaurant	Plaza	Ice Cream Shop	History Museum	Roman Restaurant	Pizza Place	Church	Trattoria/Osteria	Department Store
9	Trastevere	Italian Restaurant	Pizza Place	Café	Ice Cream Shop	Restaurant	Cocktail Bar	Plaza	Hotel Bar	Indie Movie Theater	Coffee Shop
10	Trieste	Café	Italian Restaurant	Plaza	Hotel	Pizza Place	Bar	Clothing Store	Bookstore	Cocktail Bar	Diner

As you can see from above analysis, using one hot encoding, we categorized the venue types further and sorted them based on occurrence by each borough. That gave us the following results showing the top 3 most common venues by each borough. This will help us further determine the best investment opportunity in each borough depending on the types of venues that exist currently.

	Borough	1st Most Common Venue	2nd Most Common Venue	3rd Most Common Venue
0	Balduina	Italian Restaurant	Hotel	Spa
1	Bologna	Italian Restaurant	Bar	Pizza Place
2	Camilluccia	Nightclub	Wine Shop	Flea Market
3	Caracalla	Park	Performing Arts Venue	Plaza
4	CentroStorico	Italian Restaurant	Plaza	Ice Cream Shop
5	CorsoFrancia	Italian Restaurant	Café	Pizza Place
6	Della Vittoria	Italian Restaurant	Café	Pizza Place
7	GregorioVII	Pizza Place	Hotel	Café
8	Termini	Hotel	Italian Restaurant	Plaza
9	Trastevere	Italian Restaurant	Pizza Place	Café
10	Trieste	Café	Italian Restaurant	Plaza

We will now use the previously imported K-means algorithm to find out which types of venues fall in the same category and their corresponding spatial distance:

We first set number of clusters to 2 as that will give us the optimal results, using the 'Elbow' method, explained in more detail below:

```
kclusters = 2
```

When run the k-means clustering algorithm:

```
kmeans = KMeans(n_clusters=kclusters,  
random_state=0).fit(rome_grouped_clustering)
```

We then check cluster labels generated for each row in the data frame:

```
labels = kmeans.labels_[0:11]
```

```
labels
```

```
In [67]: # set number of clusters  
kclusters = 2  
  
#rome_grouped_clustering = rome_grouped.drop('Borough', 1)  
  
# run k-means clustering  
kmeans = KMeans(n_clusters=kclusters, random_state=0).fit(rome_grouped_clustering)  
  
# check cluster labels generated for each row in the dataframe  
labels = kmeans.labels_[0:11]  
labels  
  
Out[67]: array([1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1])
```

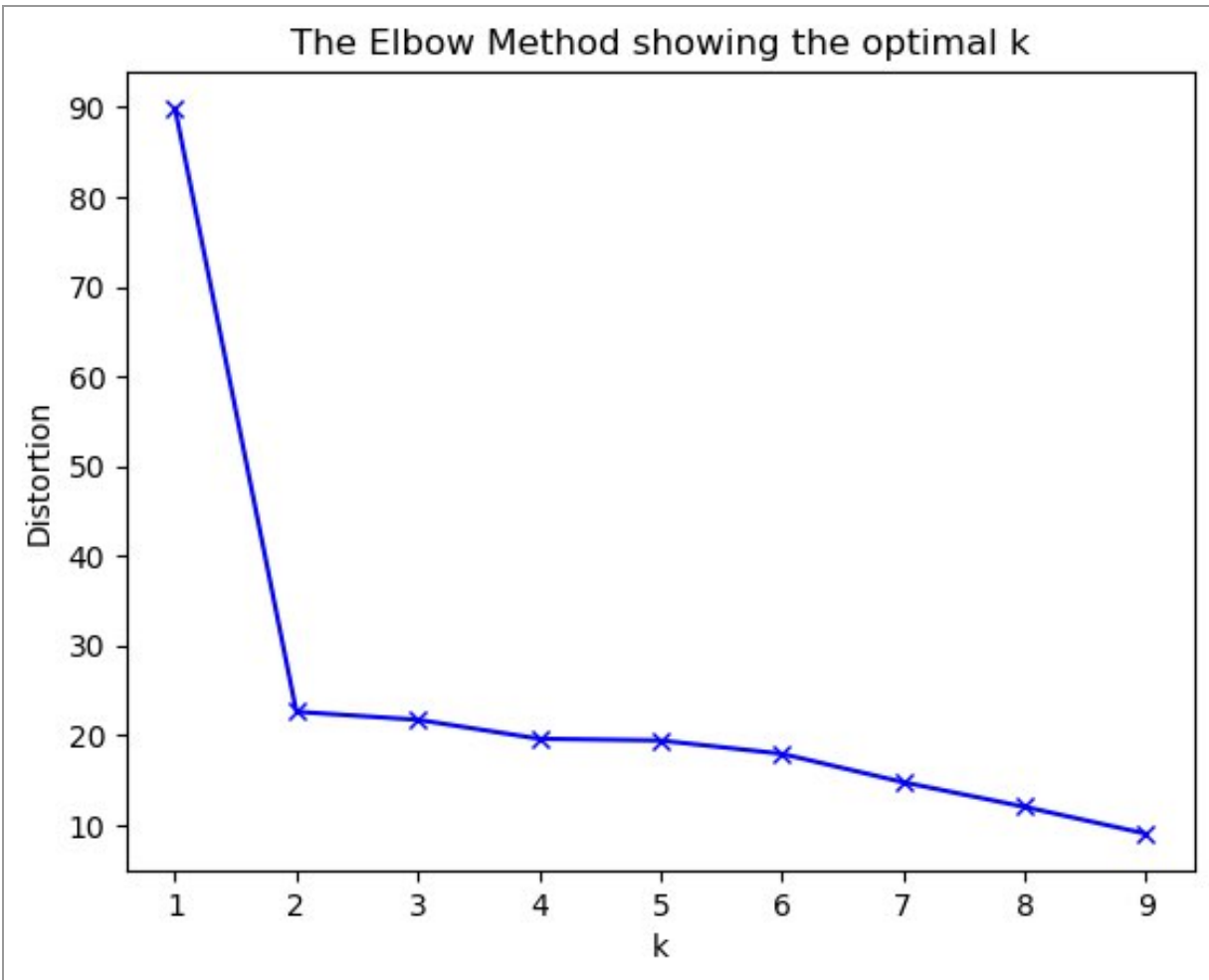
We finally visualize the results and find the optimal K-means value:

```
from scipy.spatial.distance import cdist  
distortions = []  
K = range(1,10)  
for k in K:  
kmeanModel = KMeans(n_clusters=k,  
random_state=0).fit(rome_grouped_clustering)  
distortions.append(sum(np.min(cdist(rome_grouped_c  
lustering, kmeanModel.cluster_centers_, 'canberra'),  
axis=1)) / rome_grouped_clustering.shape[0])
```

There are different metric distance functions for spatial distance. We chose Correlation instead of Euclidean method in this case because the Canberra function gives us a better view of elbow break point. This can be seen when we plot the K-distortion on a plot:

```
plt.plot(K, distortions, 'bx-')  
plt.xlabel('k')  
plt.ylabel('Distortion')  
plt.title('The Elbow Method showing the optimal k')  
plt.show()
```

```
In [68]: from scipy.spatial.distance import cdist  
  
distortions = []  
K = range(1,10)  
for k in K:  
    kmeanModel = KMeans(n_clusters=k, random_state=0).fit(rome_grouped_clustering)  
  
    distortions.append(sum(np.min(cdist(rome_grouped_clustering, kmeanModel.cluster_centers_, 'canberra'), axis=1)) / rome_g  
|  
#There are different metric distance function for spatial distance.  
#I choose correlation instaed of euclidean because the canberra function gives me more clear view of elbow break point.  
  
# Plot the elbow  
plt.plot(K, distortions, 'bx-')  
plt.xlabel('k')  
plt.ylabel('Distortion')  
plt.title('The Elbow Method showing the optimal k')  
plt.show()
```



Based on the above analysis, Optimal K value is 2 – which means we should group the venues into two clusters overall: 0 and 1. We will now create a new data-frame that includes the clusters as well as the top 10 venues for each neighborhood.

```
rome_merged = rome_data
```

We then add clustering labels:

```
boroughs_venues_sorted.insert(0, 'Cluster Labels',  
kmeans.labels_)
```

and merge rome\_grouped with rome\_data to add latitude/longitude for each neighborhood:

```

rome_merged
rome_merged.join(boroughs_venues_sorted.set_index('Borough'), on='Borough')
rome_merged.head(20)

```

```

In [73]: rome_merged = rome_data

# add clustering labels
#rome_merged['Cluster Labels'] = kmeans.Labels_
boroughs_venues_sorted.insert(0, 'Cluster Labels', kmeans.Labels_)

# merge rome_grouped with rome_data to add Latitude/Longitude for each neighborhood
rome_merged = rome_merged.join(boroughs_venues_sorted.set_index('Borough'), on='Borough')

rome_merged.head(20) # check the last columns!

```

Out[73]:

	Borough	Avg-HousePrice	Latitude	Longitude	Cluster Labels	Cluster	1st Most Common Venue	2nd Most Common Venue	3rd Most Common Venue	4th Most Common Venue	5th Most Common Venue	6th Most Common Venue	7th Most Common Venue
0	CentroStorico	7817	41.898200	12.477300	1.0	1.0	Italian Restaurant	Plaza	Ice Cream Shop	Hotel	Sandwich Place	Fountain	Restaurant
1	Caracalla	6910	41.879400	12.493100	1.0	1.0	Park	Performing Arts Venue	Plaza	Garden	Italian Restaurant	Festival	Nightclub
2	Flaminio	5622	42.191900	12.472500	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	Trastevere	5435	41.884800	12.470400	1.0	1.0	Italian Restaurant	Pizza Place	Café	Ice Cream Shop	Restaurant	Cocktail Bar	Plaza
4	Della Vittoria	5137	41.918200	12.463900	1.0	1.0	Italian Restaurant	Café	Pizza Place	Plaza	Breakfast Spot	Brazilian Restaurant	Hotel
5	Trieste	4884	45.649500	13.776800	1.0	1.0	Café	Italian Restaurant	Plaza	Hotel	Pizza Place	Bar	Clothing Store
6	Bologna	4531	44.498955	11.327500	1.0	1.0	Italian Restaurant	Bar	Pizza Place	Restaurant	Beer Bar	Bakery	Ice Cream Shop
7	CorsoFrancia	4289	45.076734	7.667100	1.0	1.0	Italian Restaurant	Café	Pizza Place	Plaza	Sushi Restaurant	Piedmontese Restaurant	Hotel
8	Termini	4107	41.901100	12.501200	1.0	1.0	Hotel	Italian Restaurant	Plaza	Ice Cream Shop	History Museum	Roman Restaurant	Pizza Place
9	Camilluccia	4099	43.968130	12.662350	0.0	0.0	Nightclub	Wine Shop	Flea Market	Coffee Shop	Convenience Store	Deli / Bodega	Department Store
10	Balduina	4000	41.920000	12.442100	1.0	1.0	Italian Restaurant	Hotel	Spa	Coffee Shop	Chinese Restaurant	Department Store	Pizza Place
11	GregorioVII	3972	41.893348	12.435962	1.0	1.0	Pizza Place	Hotel	Café	Breakfast Spot	Italian Restaurant	Seafood Restaurant	Flower Shop

We estimate the number of 1st Most Common Venue in each cluster. Then, we can create a bar chart which may help us to find proper label names for each cluster.

```

count_venue = rome_merged
count_venue = count_venue.drop(['Borough', 'Avg-HousePrice', 'Latitude', 'Longitude'], axis=1)
count_venue = count_venue.groupby(['Cluster Labels', '1st Most Common

```



```

Venue'])).size().reset_index(name='Counts')
    cv_cluster = count_venue.pivot(index='Cluster Labels',
columns='1st Most Common Venue', values='Counts')
    cv_cluster
    cv_cluster.fillna(0).astype(int).reset_index(drop=True)
    cv_cluster

```

```

In [74]: count_venue = rome_merged
count_venue = count_venue.drop(['Borough', 'Avg-HousePrice', 'Latitude', 'Longitude'], axis=1)
count_venue = count_venue.groupby(['Cluster Labels', '1st Most Common Venue']).size().reset_index(name='Counts')

#we can transpose it to plot bar chart
cv_cluster = count_venue.pivot(index='Cluster Labels', columns='1st Most Common Venue', values='Counts')
cv_cluster = cv_cluster.fillna(0).astype(int).reset_index(drop=True)
cv_cluster

Out[74]:

```

	1st Most Common Venue	Café	Hotel	Italian Restaurant	Nightclub	Park	Pizza Place
		0	0	0	0	1	0
		1	1	1	6	0	1

Creating a bar chart of "Number of Venues in Each Cluster":

```

frame=cv_cluster.plot(kind='bar',figsize=(20,8),width =
0.8)
plt.legend(labels=cv_cluster.columns,fontsize= 14)
plt.title("Number of Venues in Each Cluster",fontsize=
16)
plt.xticks(fontsize=14)
plt.xticks(rotation=0)
plt.xlabel('Number of Venue', fontsize=14)
plt.ylabel('Clusters', fontsize=14)

```

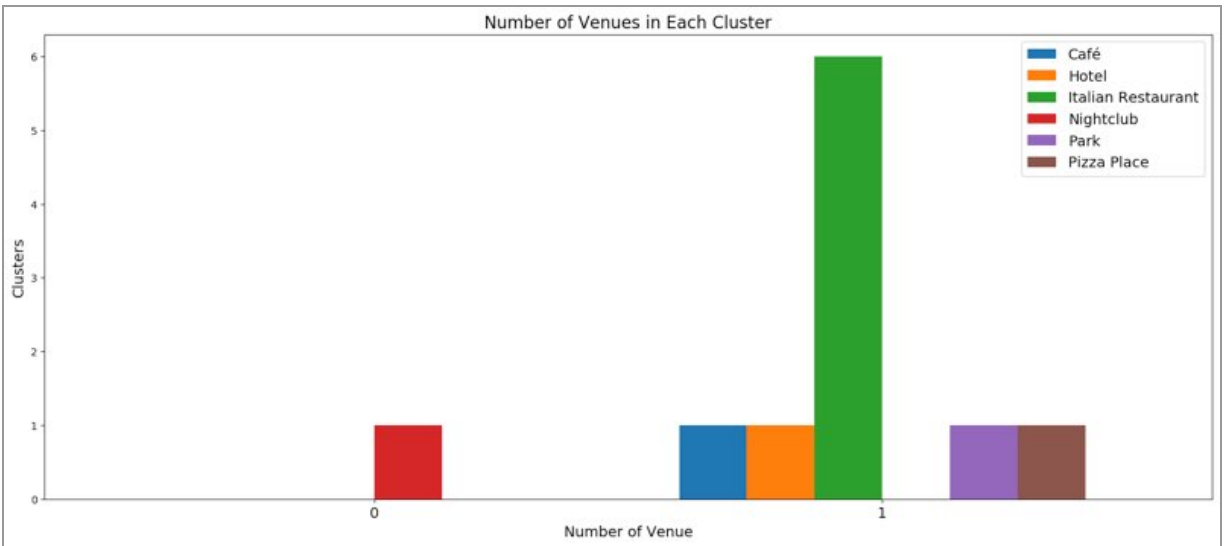
```

In [75]: #creating a bar chart of "Number of Venues in Each Cluster"
frame=cv_cluster.plot(kind='bar',figsize=(20,8),width = 0.8)

plt.legend(labels=cv_cluster.columns,fontsize= 14)
plt.title("Number of Venues in Each Cluster",fontsize= 16)
plt.xticks(fontsize=14)
plt.xticks(rotation=0)
plt.xlabel('Number of Venue', fontsize=14)
plt.ylabel('Clusters', fontsize=14)

Out[75]: Text(0, 0.5, 'Clusters')

```



When we examine above graph, we can label each cluster as follows:

Cluster 0: "Night Club"

Cluster 1: "Multiple Social Venues"

We can now assign those new labels to existing label of clusters:

```
Cluster_labels = {'Clusters': [0,1], 'Labels': ["Night Club", "Multiple Social Venues"]}
```

```
Cluster_labels = pd.DataFrame(data=Cluster_labels)
```

```
Cluster_labels
```

```
In [76]: Cluster_labels = {'Clusters': [0,1], 'Labels': ["Night Club", "Multiple Social Venues"]}
Cluster_labels = pd.DataFrame(data=Cluster_labels)
Cluster_labels

Out[76]:
```

	Clusters	Labels
0	0	Night Club
1	1	Multiple Social Venues

We can also present top 3 counts of different venue types in each neighborhood as follows:

```
top3 = rome_venues.groupby(['Borough', 'Venue Category']).size().reset_index(name='Counts')
```

```

top3
top3.sort_values(['Borough','Counts'],ascending=False).groupby('Borough').head(3).reset_index(drop=True)
top3['Join'] = top3['Counts'].map(str) + " " + top3['Venue Category']
top3 = top3.groupby(['Borough'])['Join'].apply(", ".join).reset_index()
top3.head(12)

```

```

In [86]: top3 = rome_venues.groupby(['Borough', 'Venue Category']).size().reset_index(name='Counts')
top3 = top3.sort_values(['Borough', 'Counts'], ascending=False).groupby('Borough').head(3).reset_index(drop=True)

top3['Join'] = top3['Counts'].map(str) + " " + top3['Venue Category']
top3 = top3.groupby(['Borough'])['Join'].apply(", ".join).reset_index()

top3.head(12)

```

	Borough	Join
0	Balduina	6 Italian Restaurant, 2 Hotel, 1 Asian Restaurant
1	Bologna	7 Italian Restaurant, 4 Bar, 3 Pizza Place
2	Camilluccia	1 Nightclub
3	Caracalla	2 Park, 1 Festival, 1 Garden
4	CentroStorico	16 Italian Restaurant, 14 Plaza, 9 Ice Cream Shop
5	CorsoFrancia	6 Italian Restaurant, 4 Café, 4 Pizza Place
6	Della Vittoria	7 Italian Restaurant, 4 Café, 2 Breakfast Spot
7	GregorioVII	3 Café, 3 Hotel, 3 Pizza Place
8	Termini	16 Hotel, 12 Italian Restaurant, 4 Plaza
9	Trastevere	12 Italian Restaurant, 5 Pizza Place, 4 Café
10	Trieste	11 Café, 8 Italian Restaurant, 8 Plaza

Since we have analyzed the venue groupings and most common occurrences by neighborhoods, we can now switch our attention to house prices – as that will help us narrow down the best investment opportunities among the different boroughs.

We will analyze the housing sales prices for per square meter in specific ranges. Then we can create new labels which involve pricing features, as well.

```
data_process = df.sort_values('Avg-HousePrice').reset_index(drop=True)
```

```

data_process = data_process.drop(['Latitude',
'Longitude'], axis=1)
data_process.head(12)

```

```

In [88]: data_process = df.sort_values('Avg-HousePrice').reset_index(drop=True)
data_process = data_process.drop(['Latitude', 'Longitude'], axis=1)
data_process.head(12)

Out[88]:

```

	Borough	Avg-HousePrice
0	GregorioVII	3972
1	Balduina	4000
2	Camillicia	4099
3	Termini	4107
4	CorsoFrancia	4289
5	Bologna	4531
6	Trieste	4884
7	Della Vittoria	5137
8	Trastevere	5435
9	Flaminio	5622
10	Caracalla	6910
11	CentroStorico	7817

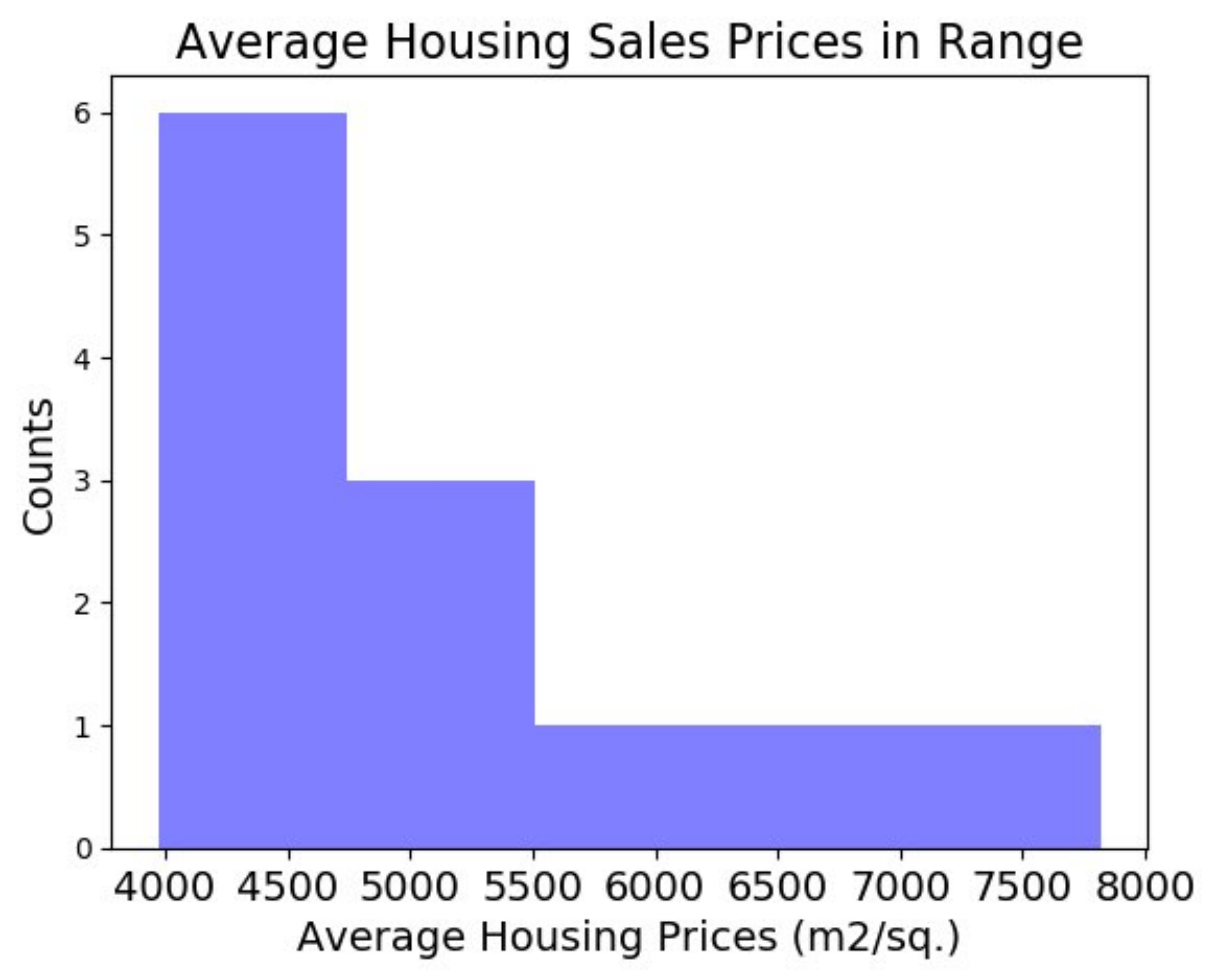
We will now examine the frequency of housing sales prices in different ranges using a histogram and for that we have to put them in 'bins'. Rest of the code below is to format the histogram, so you can adjust the color, font, and size to your liking.

```

num_bins = 5
n, bins, patches = plt.hist(data_process['Avg-
HousePrice'], num_bins, facecolor='blue', alpha=0.5)
plt.title("Average Housing Sales Prices in
Range",fontsize= 16)
plt.xticks(fontsize=14)
plt.xticks(rotation=0)
plt.xlabel('Average Housing Prices (m2/sq.)',
fontsize=14)
plt.ylabel('Counts', fontsize=14)
plt.show()

```

```
In [78]: num_bins = 5
n, bins, patches = plt.hist(data_process['Avg-HousePrice'], num_bins, facecolor='blue', alpha=0.5)
plt.title("Average Housing Sales Prices in Range", fontsize=16)
plt.xticks(fontsize=14)
plt.xticks(rotation=0)
plt.xlabel('Average Housing Prices (m2/sq.)', fontsize=14)
plt.ylabel('Counts', fontsize=14)
plt.show()
```



Based on the above histogram, house sales price (HSP) ranges can be defined as follows:

- 4000 AHP : "Low Level HSP"
- 4000-6000 AHP : "Mid Level HSP"
- 6000-8000 AHP : "High Level HSP"

In this case, we can create "Level\_labels" with those levels. Mid to High sale prices indicate good income for people residing in the area, and therefore, a good investment

opportunity if we are opening a new venue in the neighborhood.

```
level = []
for i in range(0,len(data_process)):
if (data_process['Avg-HousePrice'][i] < 4000):
level.append("Low Level HSP")
elif (data_process['Avg-HousePrice'][i] >= 4000 and
data_process['Avg-HousePrice'][i] < 6000):
level.append("Mid Level HSP")
elif (data_process['Avg-HousePrice'][i] >= 6000 and
data_process['Avg-HousePrice'][i] < 8000):
level.append("High Level HSP")
data_process['Level_labels'] = level
data_process.head(12)
```

```
In [87]: M level = []
for i in range(0,len(data_process)):
if (data_process['Avg-HousePrice'][i] < 4000):
level.append("Low Level HSP")
elif (data_process['Avg-HousePrice'][i] >= 4000 and data_process['Avg-HousePrice'][i] < 6000):
level.append("Mid Level HSP")
elif (data_process['Avg-HousePrice'][i] >= 6000 and data_process['Avg-HousePrice'][i] < 8000):
level.append("High Level HSP")

data_process['Level_labels'] = level
data_process.head(12)
```

Out[87]:

	Borough	Avg-HousePrice	Level_labels
0	GregorioVII	3972	Low Level HSP
1	Balduina	4000	Mid Level HSP
2	Camilluccia	4099	Mid Level HSP
3	Termini	4107	Mid Level HSP
4	CorsoFrancia	4289	Mid Level HSP
5	Bologna	4531	Mid Level HSP
6	Trieste	4884	Mid Level HSP
7	Della Vittoria	5137	Mid Level HSP
8	Trastevere	5435	Mid Level HSP
9	Flaminio	5622	Mid Level HSP
10	Caracalla	6910	High Level HSP
11	CentroStorico	7817	High Level HSP

We can now add house sales price details to the cluster table that also include the top venue list by neighborhood:

```
import numpy as np
```

```
result = pd.merge(rome_merged,
top3[['Borough', 'Join']],
left_on = 'Borough',
right_on = 'Borough',
how = 'left')
result= pd.merge(result,
Cluster_labels[['Clusters', 'Labels']],
left_on = 'Cluster Labels',
right_on = 'Clusters',
how = 'left')
result = pd.merge(result,
data_process[['Borough', 'Level_labels']],
left_on = 'Borough',
right_on = 'Borough',
how = 'left')
result = result.drop(['Clusters'], axis=1)
result.head(12)
```

```
In [85]: import numpy as np

result = pd.merge(rome_merged,
                 top3[['Borough', 'Join']],
                 left_on = 'Borough',
                 right_on = 'Borough',
                 how = 'left')
result= pd.merge(result, |
                 Cluster_labels[['Clusters', 'Labels']],
                 left_on = 'Cluster Labels',
                 right_on = 'Clusters',
                 how = 'left')
result = pd.merge(result,
                 data_process[['Borough', 'Level_labels']],
                 left_on = 'Borough',
                 right_on = 'Borough',
                 how = 'left')

result = result.drop(['Clusters'], axis=1)
result.head(12)
```



	Borough	Avg-HousePrice	Latitude	Longitude	Cluster Labels	Cluster	1st Most Common Venue	2nd Most Common Venue	3rd Most Common Venue
0	CentroStorico	7817	41.898200	12.477300	1.0	1.0	Italian Restaurant	Plaza	Ice Cream Shop
1	Caracalla	6910	41.879400	12.493100	1.0	1.0	Park	Performing Arts Venue	Plaza
2	Flaminio	5622	42.191900	12.472500	NaN	NaN	NaN	NaN	NaN
3	Trastevere	5435	41.884800	12.470400	1.0	1.0	Italian Restaurant	Pizza Place	Café
4	Della Vittoria	5137	41.918200	12.463900	1.0	1.0	Italian Restaurant	Café	Pizza Place
5	Trieste	4884	45.649500	13.776800	1.0	1.0	Café	Italian Restaurant	Plaza
6	Bologna	4531	44.498955	11.327500	1.0	1.0	Italian Restaurant	Bar	Pizza Place
7	CorsoFrancia	4289	45.076734	7.667100	1.0	1.0	Italian Restaurant	Café	Pizza Place
8	Termini	4107	41.901100	12.501200	1.0	1.0	Hotel	Italian Restaurant	Plaza

9	Camilluccia	4099	43.968130	12.662350	0.0	0.0	Nightclub	Wine Shop	Flea Market	Coffee Shop
10	Balduina	4000	41.920000	12.442100	1.0	1.0	Italian Restaurant	Hotel	Spa	Coffee Shop
11	GregorioVII	3972	41.893348	12.435962	1.0	1.0	Pizza Place	Hotel	Café	Breakfast Spot

Looks like Cluster 0, Camilluccia, is still looking very attractive since the house prices are in the medium range

and there is very little competition from other restaurant type venues in the neighborhood!

Finally, let's visualize the resulting clusters using the Folium library which is very good for visualizing geospatial maps – as mentioned earlier in this chapter.

```
import math  
map_clusters = folium.Map(location=[latitude,  
longitude], zoom_start=9.5)
```

We set the color scheme for the maps in the code below. You can adjust the colors and format to your liking – as mentioned previously when plotting other visualizations using Python.

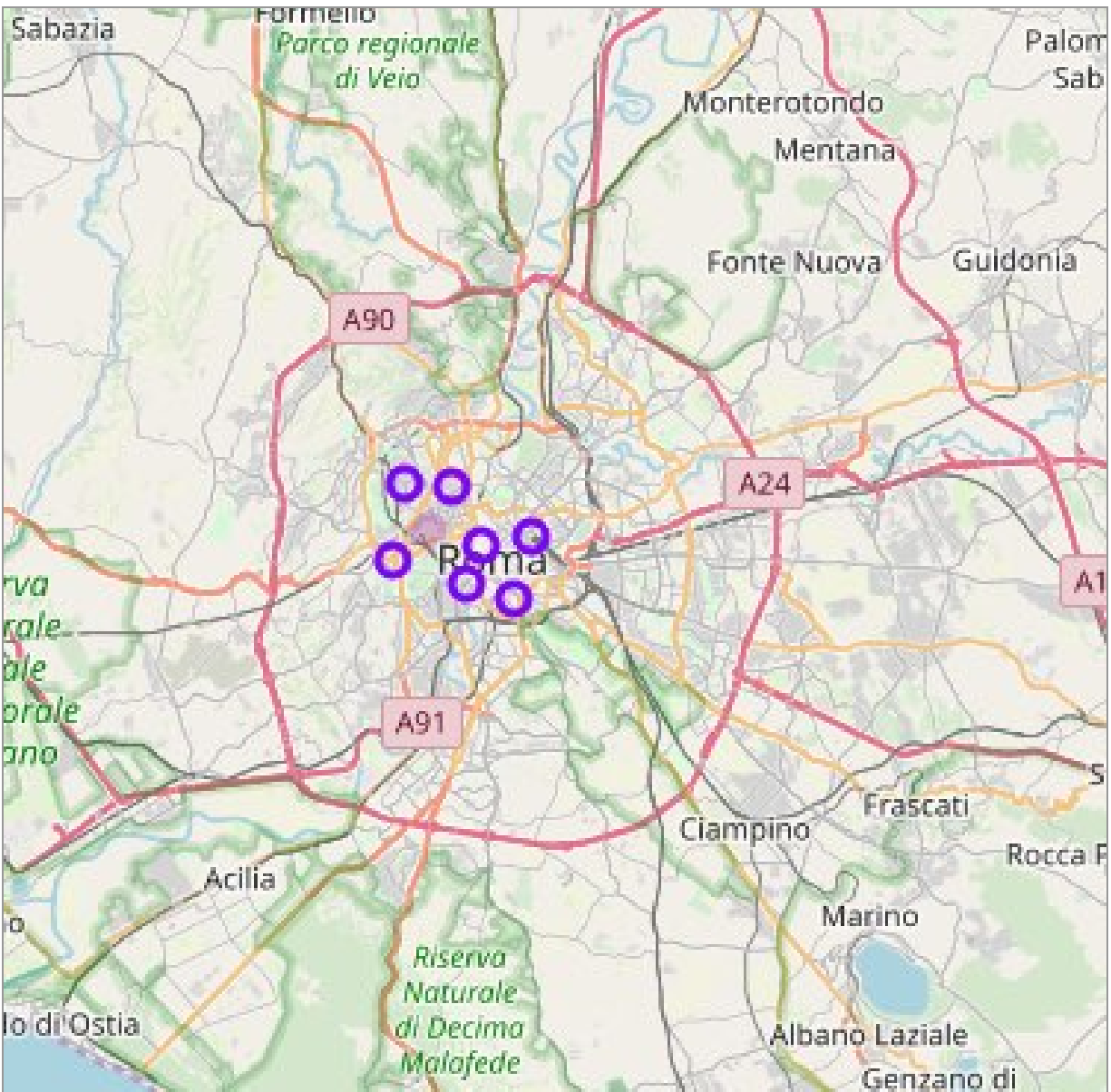
```
x = np.arange(kclusters)  
ys = [i+x+(i*x)**2 for i in range(kclusters)]  
colors_array = cm.rainbow(np.linspace(0, 1, len(ys)))  
rainbow = [colors.rgb2hex(i) for i in colors_array]
```

You can also add markers to the map to mark each borough:

```
markers_colors = []  
for lat, lon, poi, cluster, join, cluster_number, label in  
zip(result['Latitude'], result['Longitude'],  
result['Borough'], result['Labels'], result['Join'],  
result['Cluster Labels'], result['Level_labels']):  
label = folium.Popup(str(poi) + " / " + str(cluster) + "-"  
+ str(label) + " / " + str(join), parse_html=True)  
if (math.isnan(cluster_number)== False):  
folium.CircleMarker(  
[lat, lon],  
radius=5,  
color= rainbow[int(cluster_number-1)],
```

```
popup=label,  
fill_color = rainbow[int(cluster_number-1)],  
fill_opacity=1).add_to(map_clusters)  
map_clusters
```

```
In [100]: M import math  
# create map  
map_clusters = folium.Map(location=[latitude, longitude], zoom_start=9.5)  
  
# set color scheme for the clusters  
x = np.arange(kclusters)  
ys = [i+x+(i*x)**2 for i in range(kclusters)]  
colors_array = cm.rainbow(np.linspace(0, 1, len(ys)))  
rainbow = [colors.rgb2hex(i) for i in colors_array]  
  
# add markers to the map  
markers_colors = []  
for lat, lon, poi, cluster, join, cluster_number, label in zip(result['Latitude'], result['Longitude'], result['Borough'], re  
label = folium.Popup(str(poi) + " / " + str(cluster) + "-" + str(label) + " / " + str(join), parse_html=True)  
if (math.isnan(cluster_number)== False):  
    folium.CircleMarker(  
        [lat, lon],  
        radius=5,  
        color= rainbow[int(cluster_number-1)],  
        popup=label,  
        fill_color = rainbow[int(cluster_number-1)],  
        fill_opacity=1).add_to(map_clusters)  
  
map_clusters
```



As a summary of this analysis, we used a dataset containing the names of different neighborhoods for Rome as well as the average house price and longitude and latitude coordinates of those neighborhoods. We further augmented this dataset by using Foursquare API, to bring in details of the most common venues in each neighborhood.

We then used K-mean algorithm and elbow method that recommended segmenting the data into 2 clusters overall

where Camillucia neighborhood ended up in Cluster 0 and remaining neighborhoods ended up in Cluster 1. When we analyzed the most common venues in each neighborhood, it also became quite apparent that while in Cluster 1, there are several Italian restaurants and other socialization venues like cafes and hotels, Cluster 0 mainly had night clubs and wine bars.

When we further visualized the data by overlaying the real estate prices, Cluster 0 looks even more attractive from a business investment perspective as the real estate prices fall in the mid-level sales price range.

Based on the above analysis, we reach the recommendation that Cluster 0 (Camillucia neighborhood) as a good option for business investment and more specifically opening an Italian restaurant or Pizza joint as there is minimum to no competition for food locations for night club goers and the real estate price is in the mid-range.



## USE CASE 9 – CREATING RECOMMENDER SYSTEMS

**R**ecommender systems is one of the most widely used applications of Machine Learning in today's world. Here, technology meets marketing and psychology to find the best matches between consumers and products.

All of us have experience with recommender systems:

- **YouTube** proposes videos that we like
- **Spotify** makes lists with songs based on what songs we have listened to before
- **Netflix** recommends movies based on what we have watched before or what other similar users like
- **Facebook** displays advertisements based on our previous search history
- **Amazon** displays products that we would probably buy based on previous purchases, or products that other users frequently buy together with our current purchase
- **Instagram** chooses to show us popular photos based on our interests (monitored by likes and search history)

Each application that utilizes recommender systems uses a different approach to form the recommendations.

A recommender system needs the following:

1. **Users**
2. **Items** (videos on YouTube, songs on Spotify, photos on Instagram etc.)
3. Users' **activities** (likes, purchases etc.)

In this chapter, we will use a Kaggle public dataset from Netflix 2019 line up to determine how we can recommend movies based on user viewing patterns and available content on their platform.

As in previous chapters, this dataset along with the code used in this chapter will be available to you via a download link later in this book.

We will start with importing the necessary libraries:

```
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
import matplotlib.pyplot as plt
```

We will then load the dataset:

```
netflix_overall=pd.read_csv("netflix_titles.csv")  
netflix_overall.head()
```

```
In [2]: #We will start with importing the necessary libraries  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import seaborn as sns  
import matplotlib.pyplot as plt  
  
#We will then load the dataset  
  
netflix_overall=pd.read_csv("netflix_titles.csv")  
netflix_overall.head()
```

Out[2]:

	show_id	type	title	director	cast	country	date_added	release_year	rating	duration	listed_in	description
0	s1	TV Show	3%	NaN	João Miguel, Bianca Comparato, Michel Gomes, R...	Brazil	August 14, 2020	2020	TV-MA	4 Seasons	International TV Shows, TV Dramas, TV Sci-Fi &...	In a future where the elite inhabit an island ...
1	s2	Movie	7:19	Jorge Michel Grau	Demián Bichir, Héctor Bonilla, Oscar Serrano, ...	Mexico	December 23, 2016	2016	TV-MA	93 min	Dramas, International Movies	After a devastating earthquake hits Mexico Cit...
2	s3	Movie	23:59	Gilbert Chan	Tedd Chan, Stella Chung, Henley Hill, Lawrence ...	Singapore	December 20, 2018	2011	R	78 min	Horror Movies, International Movies	When an army recruit is found dead, his fellow...
3	s4	Movie	9	Shane Acker	Elijah Wood, John C. Reilly, Jennifer Connelly...	United States	November 16, 2017	2009	PG-13	80 min	Action & Adventure, Independent Movies, Sci-Fi...	In a postapocalyptic world, rag-doll robots hi...
4	s5	Movie	21	Robert Luketic	Jim Sturgess, Kevin Spacey, Kate Bosworth, Aar...	United States	January 1, 2020	2008	PG-13	123 min	Dramas	A brilliant group of students become card-coun...



Let's do a count of the overall variables in the Netflix dataset:

```
netflix_overall.count()
```

```
In [3]: #Lets do a count of the overall variables in the netflix dataset
netflix_overall.count()

Out[3]: show_id      7787
        type         7787
        title        7787
        director     5398
        cast         7069
        country      7280
        date_added   7777
        release_year 7787
        rating       7780
        duration     7787
        listed_in    7787
        description  7787
        dtype: int64
```

Let's create specific variables for tv shows and movies for further exploration

```
netflix_shows=netflix_overall[netflix_overall['type']=='TV Show']
```

```
netflix_movies=netflix_overall[netflix_overall['type']=='Movie']
```

We can now do a visual comparison by plotting tv shows vs movies as a bar chart using seaborn library:

```
sns.set(style="darkgrid")
ax = sns.countplot(x="type", data=netflix_overall,
palette="Set1")
```



It is obvious from above analysis that there are more movies than tv shows on the Netflix platform.

Now suppose we wanted to know when the best time is to release new content on the Netflix platform. To determine that, we will create a heatmap that shows releases of shows by year on x-axis and month of release on y-axis to determine the busiest and lightest months for new content. NOTE: Most of the code below is to format the charts and to set the color, font, and size of the plot. You can adjust to your liking by changing the parameters in the code.

```
netflix_date = netflix_shows[['date_added']].dropna()
netflix_date['year']
netflix_date['date_added'].apply(lambda x : x.split(' ', ' )
[-1])
netflix_date['month']
netflix_date['date_added'].apply(lambda x :
x.lstrip().split(' ')[0])
month_order = ['January', 'February', 'March', 'April',
'May', 'June', 'July', 'August', 'September', 'October',
'November', 'December'][::-1]
```

```

df = netflix_date.groupby('year')
['month'].value_counts().unstack().fillna(0)
[month_order].T
plt.figure(figsize=(9, 8), dpi=250)
plt.pcolor(df, cmap='YlGnBu_r', edgecolors='white',
linewidths=2)
plt.xticks(np.arange(0.5, len(df.columns), 1),
df.columns, fontsize=7, fontfamily='serif')
plt.yticks(np.arange(0.5, len(df.index), 1), df.index,
fontsize=7, fontfamily='serif')
plt.title('Netflix Launching New Content', fontsize=12,
fontfamily='calibri', fontweight='bold', position=(0.20,
1.0+0.02))
cbar = plt.colorbar()
cbar.ax.tick_params(labelsize=8)
cbar.ax.minorticks_on()
plt.show()

```

```

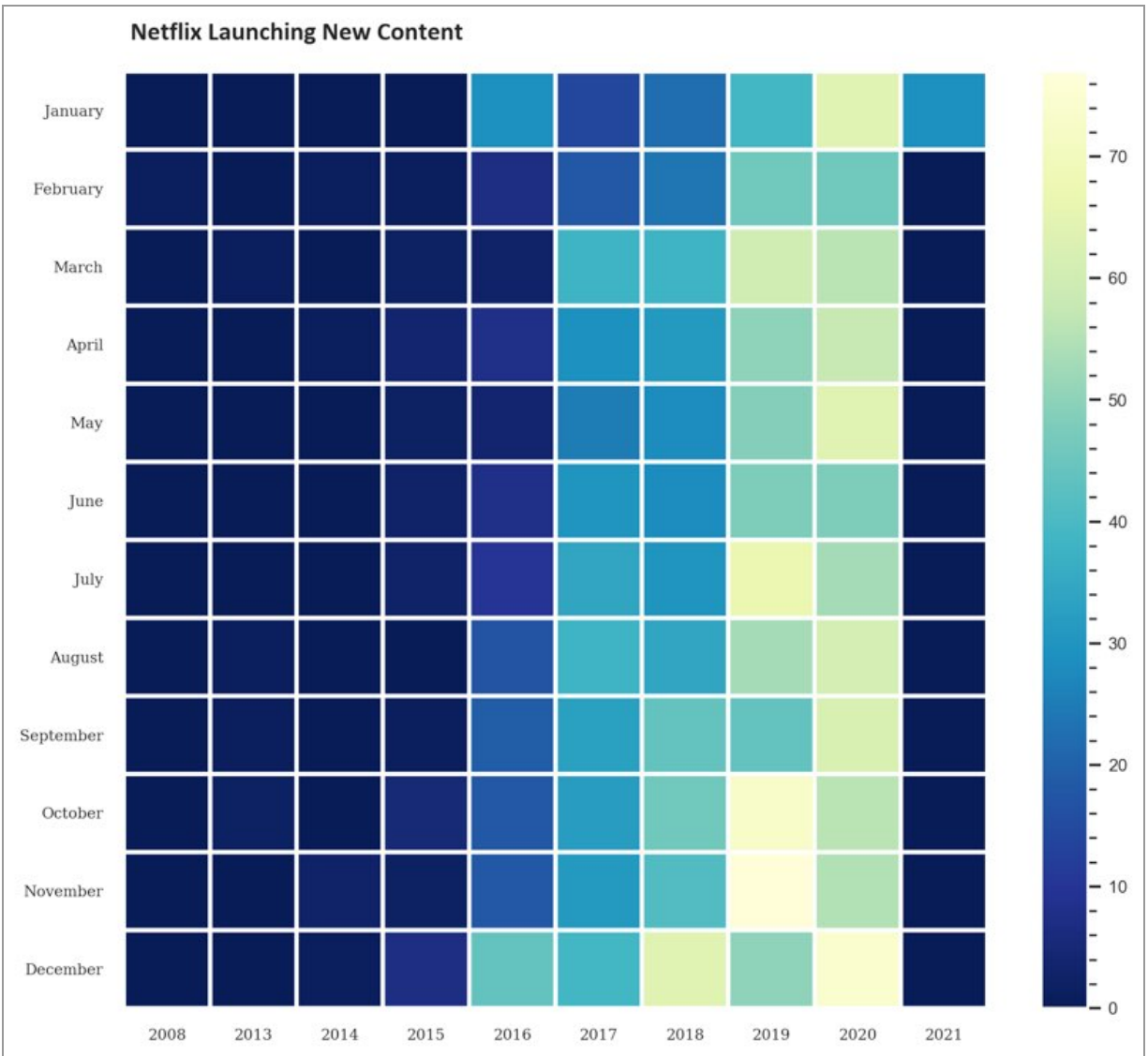
In [9]: # It is obvious from above analysis that there are movies than tv shows on the netflix platform.
# Now suppose we wanted to know when is the best timing to release new content on the netflix platform

netflix_date = netflix_shows[['date_added']].dropna()
netflix_date['year'] = netflix_date['date_added'].apply(lambda x : x.split('-')[1])
netflix_date['month'] = netflix_date['date_added'].apply(lambda x : x.lstrip().split('-')[0])

month_order = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November',
df = netflix_date.groupby('year')['month'].value_counts().unstack().fillna(0)[month_order].T
plt.figure(figsize=(9, 8), dpi=250)
plt.pcolor(df, cmap='YlGnBu_r', edgecolors='white', linewidths=2)
plt.xticks(np.arange(0.5, len(df.columns), 1), df.columns, fontsize=7, fontfamily='serif')
plt.yticks(np.arange(0.5, len(df.index), 1), df.index, fontsize=7, fontfamily='serif')

plt.title('Netflix Launching New Content', fontsize=12, fontfamily='calibri', fontweight='bold', position=(0.20, 1.0+0.02))
cbar = plt.colorbar()
|
cbar.ax.tick_params(labelsize=8)
cbar.ax.minorticks_on()
plt.show()

```



From the looks of the heatmap above, 2019 January, April to June and December were relatively lighter months for content and therefore likely good timing to release new content on the Netflix platform. Now let's analyze the movie ratings and their relative distribution.

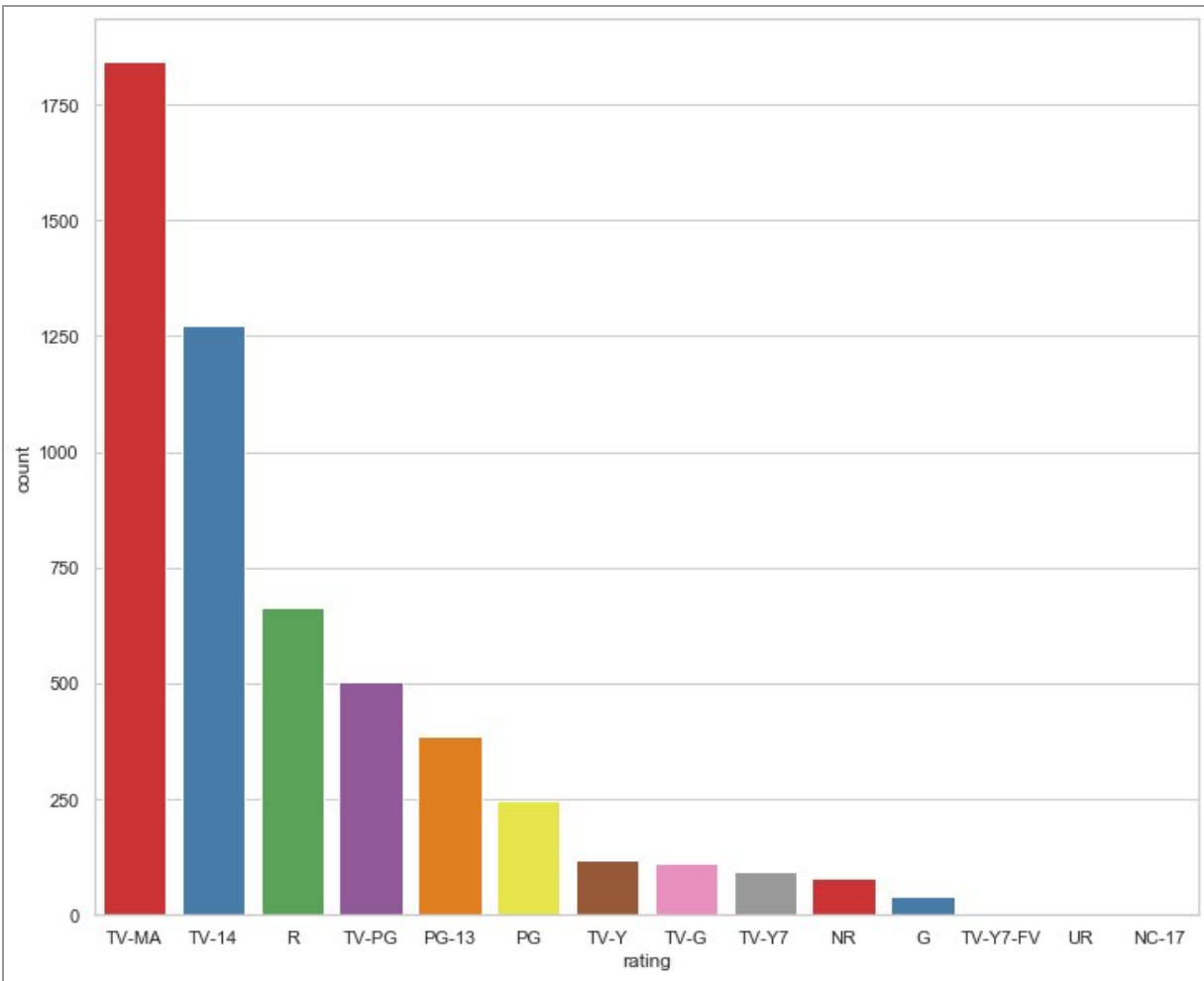
```
plt.figure(figsize=(12,10))
sns.set(style="whitegrid")
ax = sns.countplot(x="rating", data=netflix_movies,
palette="Set1", \
```

```
order=netflix_movies['rating'].value_counts().index[0:15])
```

```
In [13]: # From the Looks of the heatmap above, 2019 January, April to June and December were relatively lighter months for content
# and therefore likely good timing to release new content on the Netflix platform

# Now let's analyze the movie ratings and their relative distribution

plt.figure(figsize=(12,10))
sns.set(style="whitegrid")
ax = sns.countplot(x="rating", data=netflix_movies, palette="Set1", \
                  order=netflix_movies['rating'].value_counts().index[0:15])
```

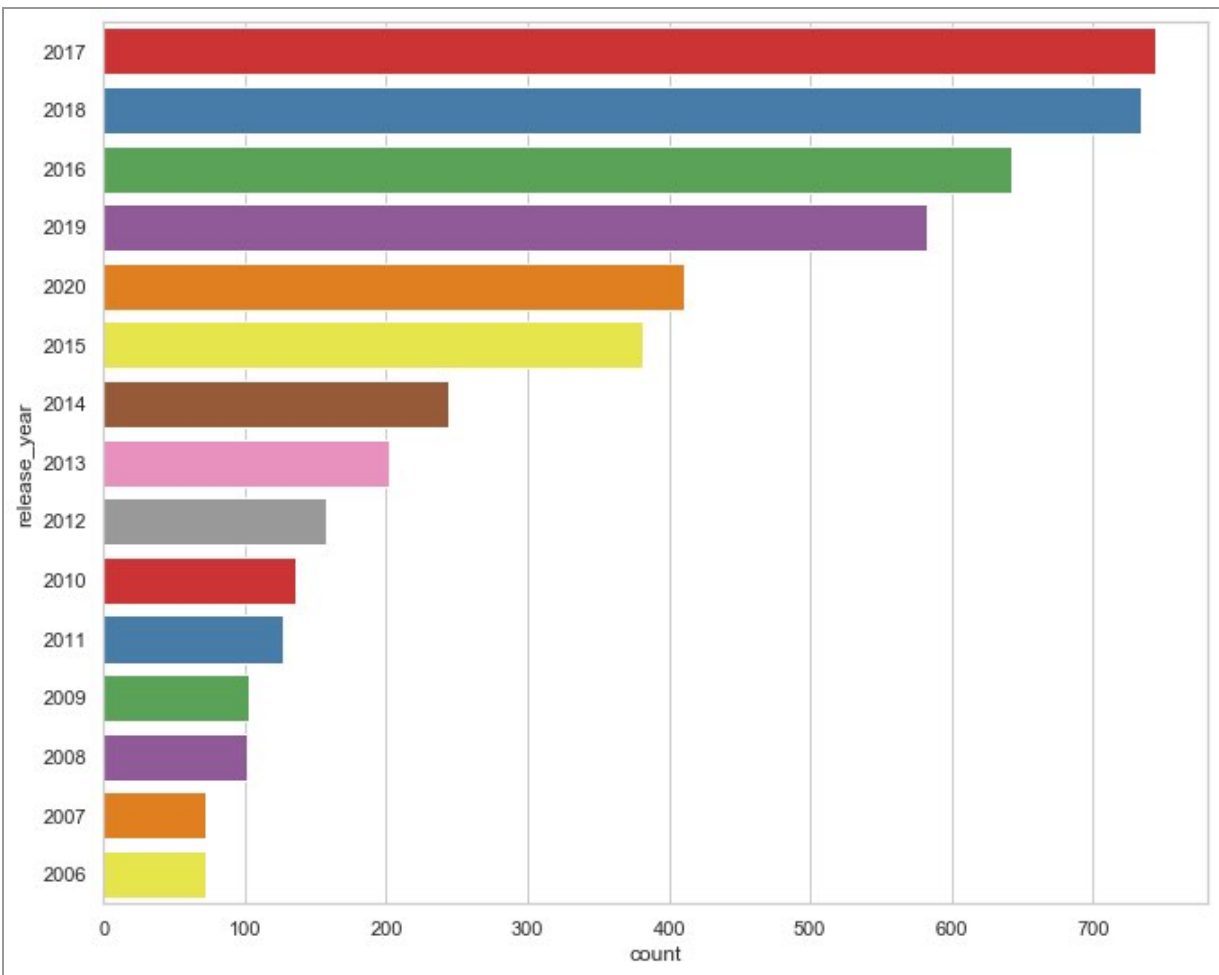


Based on above comparison, mature content that is rated MA, 14+ or R-rated outpaces more family friendly and PG content. Now let's look at which years the most content was released.

```
plt.figure(figsize=(11,9))
sns.set(style="whitegrid")
ax = sns.countplot(y="release_year",
data=netflix_movies, palette="Set1", \
order=netflix_movies['release_year'].value_counts().index[0:15])
```

```
In [12]: #Based on above comparison, mature content that is rated MA, 14+ or R-rated outpaces more family friendly and PG content
#Now Let's Look at which years the most content was Released

plt.figure(figsize=(11,9))
sns.set(style="whitegrid")
ax = sns.countplot(y="release_year", data=netflix_movies, palette="Set1", \
order=netflix_movies['release_year'].value_counts().index[0:15])
```



Based on above comparison, 2017 and 2018 had the most content released. Let's also analyze average duration of movies on Netflix.

```
netflix_movies['duration']=netflix_movies['duration'].  
str.replace(' min','')
```

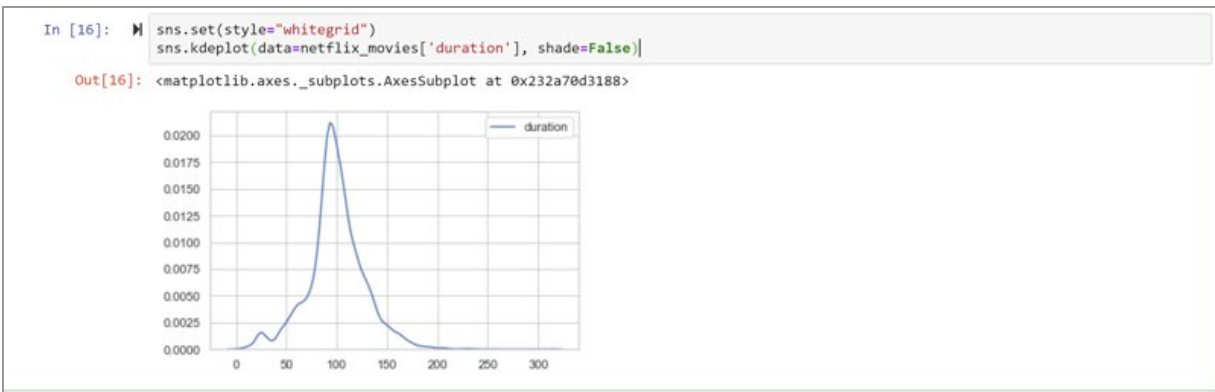
```
netflix_movies['duration']=netflix_movies['duration'].  
astype(str).astype(int)
```

```
netflix_movies['duration']
```

```
In [15]: #Based on above comparison, 2017 and 2018 had the most content released  
#Now Let's analyze average duration of movies on netflix  
netflix_movies['duration']=netflix_movies['duration'].str.replace(' min','')  
netflix_movies['duration']=netflix_movies['duration'].astype(str).astype(int)  
netflix_movies['duration']
```

```
Out[15]: 1      93  
        2      78  
        3      80  
        4     123  
        5      95  
        ..  
       7781     88  
       7782     99  
       7783    111  
       7784     44  
       7786     90  
Name: duration, Length: 5377, dtype: int32
```

```
sns.set(style="whitegrid")  
sns.kdeplot(data=netflix_movies['duration'],  
shade=False)
```



Based on the above chart, the average duration of movies on Netflix is between 80 to 150 minutes. Now let's generate a word cloud of the most common genres for movies by using the word cloud library.

```
from wordcloud import WordCloud, STOPWORDS,  
ImageColorGenerator
```

```
from PIL import Image
```

```
from collections import Counter
```

```
genres=list(netflix_movies['listed_in'])
```

```
gen=[]
```

```
for i in genres:
```

```
i=list(i.split(','))
```

```
for j in i:
```

```
gen.append(j.replace(' ',''))
```

```
g=Counter(gen)
```

```
text = list(set(gen))
```

```
plt.rcParams['figure.figsize'] = (13, 13)
```

In the code below, we format the word cloud by selecting the background color and max number of words that show up in the word cloud:

```
wordcloud =  
WordCloud(max_words=1000000,background_color="white").generate(str(text))  
plt.imshow(wordcloud,interpolation="bilinear")  
plt.axis("off")  
plt.show()
```



```

In [19]: #Based on the above chart, the average duration of movies on Netflix is between 80 to 150 minutes
#Now Let's generate a word cloud of the most common genres for movies by using the word cloud library
from wordcloud import WordCloud, STOPWORDS, ImageColorGenerator
from PIL import Image

from collections import Counter

genres=list(netflix_movies['listed_in'])
gen=[]

for i in genres:
    i=list(i.split(','))
    for j in i:
        gen.append(j.replace(' ',''))
g=Counter(gen)
|
text = list(set(gen))
plt.rcParams['figure.figsize'] = (13, 13)

#assigning shape to the word cloud
wordcloud = WordCloud(max_words=1000000,background_color="white").generate(str(text))

plt.imshow(wordcloud,interpolation="bilinear")
plt.axis("off")
plt.show()

```



Based on above word cloud, it appears that dramas, cult movies and musicals are the most common movie genres on Netflix. For full coverage, let's generate a similar word cloud for tv shows.

```

genres=list(netflix_shows['listed_in'])
gen=[]
for i in genres:
    i=list(i.split(','))

```

```

for j in i:
    gen.append(j.replace(' ', ""))
g=Counter(gen)
from wordcloud import WordCloud, STOPWORDS,
ImageColorGenerator
text = list(set(gen))
wordcloud =
WordCloud(max_words=1000000,background_color="white").generate(str(text))
plt.rcParams['figure.figsize'] = (13, 13)
plt.imshow(wordcloud,interpolation="bilinear")
plt.axis("off")
plt.show()

```

```

In [20]: M #Based on above word cloud, it appears that dramas, cult movies and musicals are the most common movie genres on Netflix
#Now let's generate a similar word cloud for tv shows

genres=list(netflix_shows['listed_in'])
gen=[]

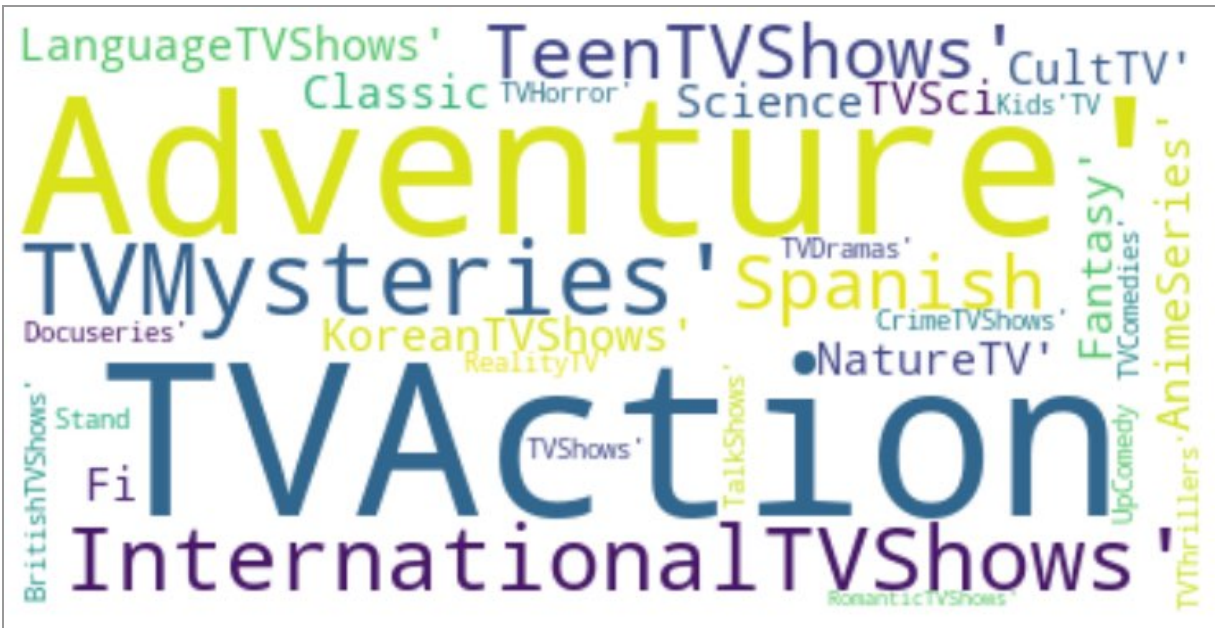
for i in genres:
    i=list(i.split(','))
    for j in i:
        gen.append(j.replace(' ', ""))
g=Counter(gen)

from wordcloud import WordCloud, STOPWORDS, ImageColorGenerator

text = list(set(gen))

wordcloud = WordCloud(max_words=1000000,background_color="white").generate(str(text))
plt.rcParams['figure.figsize'] = (13, 13)
plt.imshow(wordcloud,interpolation="bilinear")
plt.axis("off")
plt.show()

```



Based on the above word cloud, Action, Adventure, International and Mysteries are the most common tv show genres on the platform. Now that we have analyzed the Netflix movies and tv show genres, duration, and content length, let's create our own recommender system to recommend Netflix content to the users.

For that we will use the TfidfVectorizer library from SKLearn that comes built-in with recommender system algorithms. The TF-IDF(Term Frequency-Inverse Document Frequency) score is the frequency of a word occurring in a document, down-weighted by the number of documents in which it occurs. This is done to reduce the importance of words that occur frequently in plot overviews and therefore, their significance in computing the final similarity score.

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

We apply some data cleaning to the text by removing stop words and null values. Stop words are generally filtered out

before processing a natural language. These are the most common words in any language like articles, prepositions, pronouns, and conjunctions and do not add much information to the text that is being analyzed.

```
tfidf = TfidfVectorizer(stop_words='english')  
netflix_overall['description'] =  
netflix_overall['description'].fillna('')
```

We then construct the required TF-IDF matrix by fitting and transforming the data:

```
tfidf_matrix =  
tfidf.fit_transform(netflix_overall['description'])
```

And finally output the shape of tfidf\_matrix:

```
tfidf_matrix.shape
```

```
In [22]: #removing stopwords  
tfidf = TfidfVectorizer(stop_words='english')  
  
#Replace NaN with an empty string  
netflix_overall['description'] = netflix_overall['description'].fillna('')  
  
#Construct the required TF-IDF matrix by fitting and transforming the data  
tfidf_matrix = tfidf.fit_transform(netflix_overall['description'])  
  
#Output the shape of tfidf_matrix  
tfidf_matrix.shape  
  
Out[22]: (7787, 17905)
```

To generate our recommendations, we will use the cosine similarity score. Cosine similarity measures the similarity between two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines whether two vectors are pointing in roughly the same direction. It uses the Euclidean dot product formula (as illustrated below) to find similarities between components of two vectors A and B. It is often used to measure document similarity in text analysis and is relatively straightforward and computationally efficient to calculate.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}},$$

Here is the code to compute the cosine similarity matrix and a function called 'get\_recommendations' that we can use to generate recommendations based on the user's previous viewing habits.

We first import the linear\_kernel to compute the cosine similarity matrix:

```
from sklearn.metrics.pairwise import linear_kernel
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
indices = pd.Series(netflix_overall.index,
index=netflix_overall['title']).drop_duplicates()
```

We then define the function to get recommendations:

```
def get_recommendations(title,
cosine_sim=cosine_sim):
idx = indices[title]
```

First, we get the pairwise similarity scores of all movies with that movie:

```
sim_scores = list(enumerate(cosine_sim[idx]))
```

We then sort the movies based on the similarity scores:

```
sim_scores = sorted(sim_scores, key=lambda x: x[1],
reverse=True)
```

We narrow it down to get the scores of the 10 most similar movies:

```
sim_scores = sim_scores[1:11]
```

and get the movie indices:

```
movie_indices = [i[0] for i in sim_scores]
Finally, we return the top 10 most similar movies:
return netflix_overall['title'].iloc[movie_indices]
```

```
In [23]: # To generate our recommendations, we will use the cosine similarity score. Cosine similarity measures the similarity between
# two vectors of an inner product space. It is measured by the cosine of the angle between two vectors and determines whether
# two vectors are pointing in roughly the same direction. It is often used to measure document similarity in text analysis
# and is relatively straightforward and computationally efficient to calculate

# Import linear_kernel
from sklearn.metrics.pairwise import linear_kernel

# Compute the cosine similarity matrix
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)

indices = pd.Series(netflix_overall.index, index=netflix_overall['title']).drop_duplicates()

def get_recommendations(title, cosine_sim=cosine_sim):
    idx = indices[title]

    # Get the pairwise similarity scores of all movies with that movie
    sim_scores = list(enumerate(cosine_sim[idx]))

    # Sort the movies based on the similarity scores
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    # Get the scores of the 10 most similar movies
    sim_scores = sim_scores[1:11]

    # Get the movie indices
    movie_indices = [i[0] for i in sim_scores]

    # Return the top 10 most similar movies
    return netflix_overall['title'].iloc[movie_indices]
```

Now that we have defined our recommendation methodology, let's see the type of recommendations we will get if our favorite movie was 'The Matrix'.

```
get_recommendations('The Matrix')
```

```
In [24]: # Now that we have defined our recommendation methodology, let's see the type of recommendations we will get if our favorite
# movie was 'The Matrix'

get_recommendations('The Matrix')

Out[24]: 6396    The Girl with the Dragon Tattoo
          3          9
          4927    Power Rangers Beast Morphers
          5537    Shakti: The Power
          2651    Haunted
          5527    Sextuplets
          566    Arès
          7094    Time Please
          5554    She-Ra and the Princesses of Power
          2512    Grand Army
          Name: title, dtype: object
```

Interesting...and now what about if our favorite TV show was 'Breaking Bad'.

```
get_recommendations('Breaking Bad')
```

```
In [29]: #Interesting...and now what about if our favorite TV show was 'Breaking Bad'
         get_recommendations('Breaking Bad')

Out[29]: 6156          The Book of Sun
         3056          Iron Ladies
         394           Alice Junior
         1782         Dismissed
         380           Alexa & Katie
         6357         The Five Venoms
         6850         The School of Mischief
         6659         The Mess You Leave Behind
         4700         Ouran High School Host Club
         5954         Taare Zameen Par
         Name: title, dtype: object
```

And there you have it folks! A simple way to create our very own recommender system that we created from scratch! Congratulations you have now gone through all common data science use cases in the real world and should feel extremely confident tackling this brave new world empowered with mastering Artificial Intelligence!





## AFTERWORD

Wow! What a journey! Thank you for reading through the data science use cases in this book and how you can apply them in the real world!

You learned the most common day to day Python and predictive analytics applications including:

- Use Case 1 – Web scrapping to get the data you need from relevant websites and format it into a data frame for further analysis
- Use Case 2 – Image processing to analyze and format pictures for pattern recognition
- Use Case 3 – Different file type processing to be able to handle any data wrangling challenges
- Use Case 4 – Sending and receiving emails for campaign automation
- Use Case 5 – Time warping to compare data with different time scale
- Use Case 6 – Time series analysis and forecasting to predict the future based on history!
- Use Case 7 – Financial fraud analysis and determine the features required to identify these type of cases
- Use Case 8 – Processing geospatial data for opportunity analysis
- Use Case 9 – Recommender system development to see how Netflix works

And you did all of that by using hands on code examples that are readily available to you for download (see the Free

Gift chapter for more details).

Also, if you haven't already, please make sure to check out the first two best-selling books in this series:

The [first best selling book in this series, Ultimate Step by Step Guide to Machine Learning using Python](http://mybook.to/MachineLearningPython), gets you started on your journey by including step by step instructions to set up Python, introduces you to basic syntax, data structures and data cleaning. It then takes you through a real-life use case where you create a machine learning predictive model from scratch! To purchase this book, follow this link to get redirected to your local Amazon site: <http://mybook.to/MachineLearningPython>.

The [second best selling book in this series, Ultimate Step by Step Guide to Deep Learning using Python](http://mybook.to/DeepLearningPython) gets into neural networks concepts. It further differentiates machine learning models from deep learning models and as a bonus, shows you how you can deploy and optimize your models at scale in the cloud! To purchase this book, follow this link to get redirected to your local Amazon site: <http://mybook.to/DeepLearningPython>.

If you have gone through all three of these books, there is no stopping you in mastering the machine learning, deep learning, and data science world! All the best to you in your career progression!



## POST YOUR REVIEW

**Y**our review will help me improve this book and future content and will also help other readers find this book!

Thanks again for purchasing this book and your continued support!



**WEBSITE AND FREE GIFT (CODE TO DOWNLOAD)!**

**D**on't be a stranger and please check out my website:  
<https://daneyalauthor.com/datascience>  
You will be able to download the code and datasets used in this book by using the above link.



## REFERENCES

- 1. One of the richest in content websites about machine learning:**  
Machine Learning, Medium.  
<https://medium.com/topic/machine-learning>
- 2. Detailed tutorials and articles:**  
Learn Machine learning, artificial intelligence, business analytics, data science, big data, data visualizations tools and techniques, Analytics Vidhya.  
<https://www.analyticsvidhya.com/>
- 3. Machine learning blog & code: Machine Learning Mastery.** <https://machinelearningmastery.com/>
- 4. Start coding with Jupyter notebook:**  
Running the Jupyter Notebook, Running the Jupyter Notebook - Jupyter/IPython Notebook Quick Start Guide 0.1 documentation  
<https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/execute.html>
- 5. Check Python documentation when coding:**  
Latest Python documentation:  
<https://www.python.org/doc/>
- 6. When stuck, google your question, and always check Stack Overflow:**  
Where Developers Learn, Share, & Build Careers, Stack overflow.  
<https://stackoverflow.com/>
- 7. A machine learning community with coding challenges and public datasets:**



Your Machine Learning and Data Science Community,  
Kaggle.

<https://www.kaggle.com/>

Public datasets used in this book under Creative  
Commons license:

- **Time Series Forecasting: Ecommerce Daily Orders Data**
- [https://www.kaggle.com/jyesawtellrickson/ecommerce-bookings-data?select=ecommerce\\_data.csv](https://www.kaggle.com/jyesawtellrickson/ecommerce-bookings-data?select=ecommerce_data.csv)
- **Fraud Analysis: Synthetic Financial Datasets for Fraud Detection**
- <https://www.kaggle.com/ealaxi/paysim1>
- **Netflix Movies and TV Shows**
- [https://www.kaggle.com/shivamb/netflix-shows?select=netflix\\_titles.csv](https://www.kaggle.com/shivamb/netflix-shows?select=netflix_titles.csv)