# Week 1 - Retreiving and Preparing Text for Machines

This week, we begin by "begging, borrowing and stealing" text from several contexts of human communication (e.g., PDFs, HTML, Word) and preparing it for machines to "read" and analyze. This notebook outlines scraping text from the web, from images, PDF and Word documents. Then we detail "spidering" or walking through hyperlinks to build samples of online content, and using APIs, Application Programming Interfaces, provided by webservices to access their content. Along the way, we will use regular expressions, outlined in the reading, to remove unwanted formatting and ornamentation. Finally, we discuss various text encodings, filtering and data structures in which text can be placed for analysis.

For this notebook we will be using the following packages:

#All these packages need to be installed from pip  
import requests #for http requests  
import bs4 #called `beautifulsoup4`, an html parser  
import pandas #gives us DataFrames  
import docx #reading MS doc files, install as `python-docx`  
import PIL.Image #called `pillow`, an image processing library  
import pytesseract #an interface to Tesseract, you will need Tesseract  
  
#Stuff for pdfs  
#Install as `pdfminer2`  
import pdfminer.pdfinterp  
import pdfminer.converter  
import pdfminer.layout  
import pdfminer.pdfpage  
  
#These come with Python  
import re #for regexs  
import urllib.parse #For joining urls  
import io #for making http requests look like files  
import json #For Tumblr API responses  
import os.path #For checking if files exist  
import os #For making directories

We will also be working on the following files/urls

wikipedia\_base\_url = 'https://en.wikipedia.org'  
wikipedia\_content\_analysis = 'https://en.wikipedia.org/wiki/Content\_analysis'  
content\_analysis\_save = 'wikipedia\_content\_analysis.html'  
example\_text\_file = 'sometextfile.txt'  
information\_extraction\_pdf = 'https://github.com/KnowledgeLab/content\_analysis/raw/data/21.pdf'  
  
ocr\_file = 'data/tesseractTest.png'

# Scraping

Before we can start analyzing content we need to obtain it. Sometimes it will be provided to us from a pre-curated text archive, but sometimes we will need to download it. As a starting example we will attempt to download the wikipedia page on content analysis. The page is located at [https://en.wikipedia.org/wiki/ Content\_analysis](https://en.wikipedia.org/wiki/Content_analysis) so lets start with that.

We can do this by making an HTTP GET request to that url, a GET request is simply a request to the server to provide the contents given by some url. The other request we will be using in this class is called a POST request and requests the server to take some content we provide. While the Python standard library does have the ability do make GET requests we will be using the [*requests*](http://docs.python-requests.org/en/master/) package as it is *'the only Non-GMO HTTP library for Python'*...also it provides a nicer interface.

#wikipedia\_content\_analysis = 'https://en.wikipedia.org/wiki/Content\_analysis'  
requests.get(wikipedia\_content\_analysis)

'Response [200]' means the server responded with what we asked for. If you get another number (e.g. 404) it likely means there was some kind of error, these codes are called HTTP response codes and a list of them can be found [here](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes). The response object contains all the data the server sent including the website's contents and the HTTP header. We are interested in the contents which we can access with the .text attribute.

wikiContentRequest = requests.get(wikipedia\_content\_analysis)  
print(wikiContentRequest.text[:1000])

This is not what we were looking for, because it is the start of the HTML that makes up the website. This is HTML and is meant to be read by computers. Luckily we have a computer to parse it for us. To do the parsing we will use [*Beautiful Soup*](https://www.crummy.com/software/BeautifulSoup/) which is a better parser than the one in the standard library.

wikiContentSoup = bs4.BeautifulSoup(wikiContentRequest.text, 'html.parser')  
print(wikiContentSoup.text[:200])

This is better but there's still random whitespace and we have more than just the text of the article. This is because what we requested is the whole webpage, not just the text for the article.

We want to extract only the text we care about, and in order to do this we will need to inspect the html. One way to do this is simply to go to the website with a browser and use its inspection or view source tool. If javascript or other dynamic loading occurs on the page, however, it is likely that what Python receives is not what you will see, so we will need to inspect what Python receives. To do this we can save the html requests obtained.

#content\_analysis\_save = 'wikipedia\_content\_analysis.html'  
  
with open(content\_analysis\_save, mode='w', encoding='utf-8') as f:  
 f.write(wikiContentRequest.text)

Now lets open the file (wikipedia\_content\_analysis.html) we just created with a web browser. It should look sort of like the original but without the images and formatting.

As there is very little standardization on structuring webpages, figuring out how best to extract what you want is an art. Looking at this page it looks like all the main textual content is inside <p>(paragraph) tags within the <body> tag.

contentPTags = wikiContentSoup.body.findAll('p')  
for pTag in contentPTags[:3]:  
 print(pTag.text)

We now have all the text from the page, split up by paragraph. If we wanted to get the section headers or references as well it would require a bit more work, but is doable.

There is one more thing we might want to do before sending this text to be processed, remove the references indicators ([2], [3] , etc). To do this we can use a short regular expression (regex).

contentParagraphs = []  
for pTag in contentPTags:  
 #strings starting with r are raw so their \'s are not modifier characters  
 #If we didn't start with r the string would be: '\\[\\d+\\]'  
 contentParagraphs.append(re.sub(r'\[\d+\]', '', pTag.text))  
  
#convert to a DataFrame  
contentParagraphsDF = pandas.DataFrame({'paragraph-text' : contentParagraphs})  
print(contentParagraphsDF)

Now we have a DataFrame containing all relevant text from the page ready to be processed

If you are not familiar with regex, it is a way of specifying searches in text. A regex engine takes in the search pattern, in the above case '\[\d+\]' and some string, the paragraph texts. Then it reads the input string one character at a time checking if it matches the search. Here the regex '\d' matches number characters (while '\[' and '\]' capture the braces on either side).

findNumber = r'\d'  
regexResults = re.search(findNumber, 'not a number, not a number, numbers 2134567890, not a number')  
regexResults

In Python the regex package (re) usually returns Match objects (you can have multiple pattern hits in a a single Match), to get the string that matched our pattern we can use the .group() method, and as we want the first one we will ask for the 0'th group.

print(regexResults.group(0))

That gives us the first number, if we wanted the whole block of numbers we can add a wildcard '+' which requests 1 or more instances of the preceding character.

findNumbers = r'\d+'  
regexResults = re.search(findNumbers, 'not a number, not a number, numbers 2134567890, not a number')  
print(regexResults.group(0))

Now we have the whole block of numbers, there are a huge number of special characters in regex, for the full description of Python's implementation look at the [re docs](https://docs.python.org/3/library/re.html) there is also a short [tutorial](https://docs.python.org/3/howto/regex.html#regex-howto).

## *Your Turn*

Construct cells immediately below this that describe and download webcontent relating to your anticipated final project. Use beautiful soup and at least five regular expressions to extract relevant, nontrivial *chunks* of that content (e.g., cleaned sentences, paragraphs, etc.) to a pandas Dataframe.

# Spidering

What if we want to to get a bunch of different pages from wikipedia. We would need to get the url for each of the pages we want. Typically, we want pages that are linked to by other pages and so we will need to parse pages and identify the links. Right now we will be retrieving all links in the body of the content analysis page.

To do this we will need to find all the <a> (anchor) tags with hrefs (hyperlink references) inside of <p> tags. href can have many [different](http://stackoverflow.com/questions/4855168/what-is-href-and-why-is-%20it-used) [forms](https://en.wikipedia.org/wiki/Hyperlink#Hyperlinks_in_HTML) so dealing with them can be tricky, but generally, you will want to extract absolute or relative links. An absolute link is one you can follow without modification, while a relative link requires a base url that you will then append. Wikipedia uses relative urls for its internal links: below is an example for dealing with them.

#wikipedia\_base\_url = 'https://en.wikipedia.org'  
  
otherPAgeURLS = []  
#We also want to know where the links come from so we also will get:  
#the paragraph number  
#the word the link is in  
for paragraphNum, pTag in enumerate(contentPTags):  
 #we only want hrefs that link to wiki pages  
 tagLinks = pTag.findAll('a', href=re.compile('/wiki/'), class\_=False)  
 for aTag in tagLinks:  
 #We need to extract the url from the <a> tag  
 relurl = aTag.get('href')  
 linkText = aTag.text  
 #wikipedia\_base\_url is the base we can use the urllib joining function to merge them  
 #Giving a nice structured tupe like this means we can use tuple expansion later  
 otherPAgeURLS.append((  
 urllib.parse.urljoin(wikipedia\_base\_url, relurl),  
 paragraphNum,  
 linkText,  
 ))  
print(otherPAgeURLS[:10])

We will be adding these new texts to our DataFrame contentParagraphsDF so we will need to add 2 more columns to keep track of paragraph numbers and sources.

contentParagraphsDF['source'] = [wikipedia\_content\_analysis] \* len(contentParagraphsDF['paragraph-text'])  
contentParagraphsDF['paragraph-number'] = range(len(contentParagraphsDF['paragraph-text']))  
  
contentParagraphsDF

Then we can add two more columns to our Dataframe and define a function to parse each linked page and add its text to our DataFrame.

contentParagraphsDF['source-paragraph-number'] = [None] \* len(contentParagraphsDF['paragraph-text'])  
contentParagraphsDF['source-paragraph-text'] = [None] \* len(contentParagraphsDF['paragraph-text'])  
  
def getTextFromWikiPage(targetURL, sourceParNum, sourceText):  
 #Make a dict to store data before adding it to the DataFrame  
 parsDict = {'source' : [], 'paragraph-number' : [], 'paragraph-text' : [], 'source-paragraph-number' : [], 'source-paragraph-text' : []}  
 #Now we get the page  
 r = requests.get(targetURL)  
 soup = bs4.BeautifulSoup(r.text, 'html.parser')  
 #enumerating gives use the paragraph number  
 for parNum, pTag in enumerate(soup.body.findAll('p')):  
 #same regex as before  
 parsDict['paragraph-text'].append(re.sub(r'\[\d+\]', '', pTag.text))  
 parsDict['paragraph-number'].append(parNum)  
 parsDict['source'].append(targetURL)  
 parsDict['source-paragraph-number'].append(sourceParNum)  
 parsDict['source-paragraph-text'].append(sourceText)  
 return pandas.DataFrame(parsDict)

And run it on our list of link tags

for urlTuple in otherPAgeURLS[:3]:  
 #ignore\_index means the indices will not be reset after each append  
 contentParagraphsDF = contentParagraphsDF.append(getTextFromWikiPage(\*urlTuple),ignore\_index=True)  
contentParagraphsDF

## *Your Turn*

Construct cells immediately below this that spider webcontent from another site with content relating to your anticipated final project. Specifically, identify urls on a core page, then follow and extract content from them into a pandas Dataframe. In addition, demonstrate a *recursive* spider, which follows more than one level of links (i.e., follows links from a site, then follows links on followed sites to new sites, etc.), making sure to define a reasonable endpoint so that you do not wander the web forever :-).

## API (Tumblr)

Generally website owners do not like you scraping their sites. If done badly, scarping can act like a DOS attack so you should be careful how often you make calls to a site. Some sites want automated tools to access their data, so they create [application programming interface (APIs)](https://en.wikipedia.org/wiki/Application_programming_interface). An API specifies a procedure for an application (or script) to access their data. Often this is though a [representational state transfer (REST)](https://en.wikipedia.org/wiki/Representational_state_transfer) web service, which just means if you make correctly formatted HTTP requests they will return nicely formatted data.

A nice example for us to study is [Tumblr](https://www.tumblr.com), they have a [simple RESTful API](https://www.tumblr.com/docs/en/api/v1) that allows you to read posts without any complicated html parsing.

We can get the first 20 posts from a blog by making an http GET request to 'http://{blog}.tumblr.com/api/read/json', were {blog} is the name of the target blog. Lets try and get the posts from [http://lolcats-lol- cat.tumblr.com/](http://lolcats-lol-cat.tumblr.com/) (Note the blog says at the top 'One hour one pic lolcats', but the canonical name that Tumblr uses is in the URL 'lolcats-lol-cat').

tumblrAPItarget = 'http://{}.tumblr.com/api/read/json'  
  
r = requests.get(tumblrAPItarget.format('lolcats-lol-cat'))  
  
print(r.text[:1000])

This might not look very good on first inspection, but it has far fewer angle braces than html, which makes it easier to parse. What we have is [JSON](https://en.wikipedia.org/wiki/JSON) a 'human readable' text based data transmission format based on javascript. Luckily, we can readily convert it to a python dict.

#We need to load only the stuff between the curly braces  
d = json.loads(r.text[len('var tumblr\_api\_read = '):-2])  
print(d.keys())  
print(len(d['posts']))

If we read the [API specification](https://www.tumblr.com/docs/en/api/v1), we will see there are a lot of things we can get if we add things to our GET request. First we can retrieve posts by their id number. Let's first get post 146020177084.

r = requests.get(tumblrAPItarget.format('lolcats-lol-cat'), params = {'id' : 146020177084})  
d = json.loads(r.text[len('var tumblr\_api\_read = '):-2])  
d['posts'][0].keys()  
d['posts'][0]['photo-url-1280']  
  
with open('lolcat.gif', 'wb') as f:  
 gifRequest = requests.get(d['posts'][0]['photo-url-1280'], stream = True)  
 f.write(gifRequest.content)

Such beauty; such vigor (If you can't see it you have to refresh the page). Now we could retrieve the text from all posts as well as related metadata, like the post date, caption or tags. We could also get links to all the images.

#Putting a max in case the blog has millions of images  
#The given max will be rounded up to the nearest multiple of 50  
def tumblrImageScrape(blogName, maxImages = 200):  
 #Restating this here so the function isn't dependent on any external variables  
 tumblrAPItarget = 'http://{}.tumblr.com/api/read/json'  
  
 #There are a bunch of possible locations for the photo url  
 possiblePhotoSuffixes = [1280, 500, 400, 250, 100]  
  
 #These are the pieces of information we will be gathering,  
 #at the end we will convert this to a DataFrame.  
 #There are a few other datums we could gather like the captions  
 #you can read the Tumblr documentation to learn how to get them  
 #https://www.tumblr.com/docs/en/api/v1  
 postsData = {  
 'id' : [],  
 'photo-url' : [],  
 'date' : [],  
 'tags' : [],  
 'photo-type' : []  
 }  
  
 #Tumblr limits us to a max of 50 posts per request  
 for requestNum in range(maxImages // 50):  
 requestParams = {  
 'start' : requestNum \* 50,  
 'num' : 50,  
 'type' : 'photo'  
 }  
 r = requests.get(tumblrAPItarget.format(blogName), params = requestParams)  
 requestDict = json.loads(r.text[len('var tumblr\_api\_read = '):-2])  
 for postDict in requestDict['posts']:  
 #We are dealing with uncleaned data, we can't trust it.  
 #Specifically, not all posts are guaranteed to have the fields we want  
 try:  
 postsData['id'].append(postDict['id'])  
 postsData['date'].append(postDict['date'])  
 postsData['tags'].append(postDict['tags'])  
 except KeyError as e:  
 raise KeyError("Post {} from {} is missing: {}".format(postDict['id'], blogName, e))  
  
 foundSuffix = False  
 for suffix in possiblePhotoSuffixes:  
 try:  
 photoURL = postDict['photo-url-{}'.format(suffix)]  
 postsData['photo-url'].append(photoURL)  
 postsData['photo-type'].append(photoURL.split('.')[-1])  
 foundSuffix = True  
 break  
 except KeyError:  
 pass  
 if not foundSuffix:  
 #Make sure your error messages are useful  
 #You will be one of the users  
 raise KeyError("Post {} from {} is missing a photo url".format(postDict['id'], blogName))  
  
 return pandas.DataFrame(postsData)  
tumblrImageScrape('lolcats-lol-cat', 50)

Now we have the urls of a bunch of images and can run OCR on them to gather compelling meme narratives, accompanied by cats.

## *Your Turn*

Construct cells immediately below this that either gather and organize alternative content from the Tumblr API into a pandas Dataframe, or, *preferably*, draw content from another site with content related to your anticipated final project. There are APIs associated with tens of thousands of sites, including Twitter, Facebook, Google, Bing, and a wider range of archives.

# Files

What if the text we want isn't on a webpage? There are a many other sources of text available, typically organized into *files*.

## Raw text (and encoding)

The most basic form of storing text is as a *raw text* document. Source code (.py, .r, etc) is usually raw text as are text files (.txt) and those with many other extension (e.g., .csv, .dat, etc.). Opening an unknown file with a text editor is often a great way of learning what the file is.

We can create a text file in python with the open() function

#example\_text\_file = 'sometextfile.txt'  
#stringToWrite = 'A line\nAnother line\nA line with a few unusual symbols \u2421 \u241B \u20A0 \u20A1 \u20A2 \u20A3 \u0D60\n'  
stringToWrite = 'A line\nAnother line\nA line with a few unusual symbols ␡ ␛ ₠ ₡ ₢ ₣ ൠ\n'  
  
with open(example\_text\_file, mode = 'w', encoding='utf-8') as f:  
 f.write(stringToWrite)

Notice the encoding='utf-8' argument, which specifies how we map the bits from the file to the glyphs (and whitespace characters like tab ('\t') or newline ('\n')) on the screen. When dealing only with latin letters, arabic numerals and the other symbols on America keyboards you usually do not have to worry about encodings as the ones used today are backwards compatible with [ASCII](https://en.wikipedia.org/wiki/ASCII), which gives the binary representation of 128 characters.

Some of you, however, will want to use other characters (e.g., Chinese characters). To solve this there is [Unicode](https://en.wikipedia.org/wiki/Unicode) which assigns numbers to symbols, e.g., 041 is 'A' and 03A3 is 'Σ' (numbers starting with 0 are hexadecimal). Often non/beyond-ASCII characters are called Unicode characters. Unicode contains 1,114,112 characters, about 10% of which have been assigned. Unfortunately there are many ways used to map combinations of bits to Unicode symbols. The ones you are likely to encounter are called by Python *utf-8*, *utf-16* and *latin-1*. *utf-8* is the standard for Linux and Mac OS while both *utf-16* and *latin-1* are used by windows. If you use the wrong encoding, characters can appear wrong, sometimes change in number or Python could raise an exception. Lets see what happens when we open the file we just created with different encodings.

with open(example\_text\_file, encoding='utf-8') as f:  
 print("This is with the correct encoding:")  
 print(f.read())  
  
with open(example\_text\_file, encoding='latin-1') as f:  
 print("This is with the wrong encoding:")  
 print(f.read())

Notice that with *latin-1* the unicode characters are mixed up and there are too many of them. You need to keep in mind encoding when obtaining text files. Determining the encoding can sometime involve substantial work.

## PDF

Another common way text will be stored is in a PDF file. First we will download a pdf in Python. To do that lets grab a chapter from *Speech and Language Processing*, chapter 21 is on Information Extraction which seems apt. It is stored as a pdf at [https://web.stanford.edu/~jurafsky/slp3/21. pdf](https://web.stanford.edu/~jurafsky/slp3/21.pdf) although we are downloading from a copy just in case Jurafsky changes their website.

#information\_extraction\_pdf = 'https://github.com/KnowledgeLab/content\_analysis/raw/data/21.pdf'  
  
infoExtractionRequest = requests.get(information\_extraction\_pdf, stream=True)  
print(infoExtractionRequest.text[:1000])

It says 'pdf', so thats a good sign. The rest though looks like we are having issues with an encoding. The random characters are not caused by our encoding being wrong, however. They are cause by there not being an encoding for those parts at all. PDFs are nominally binary files, meaning there are sections of binary that are specific to pdf and nothing else so you need something that knows about pdf to read them. To do that we will be using [PyPDF2](https://github.com/mstamy2/PyPDF2), a PDF processing library for Python 3.

Because PDFs are a very complicated file format pdfminer requires a large amount of boilerplate code to extract text, we have written a function that takes in an open PDF file and returns the text so you don't have to.

def readPDF(pdfFile):  
 #Based on code from http://stackoverflow.com/a/20905381/4955164  
 #Using utf-8, if there are a bunch of random symbols try changing this  
 codec = 'utf-8'  
 rsrcmgr = pdfminer.pdfinterp.PDFResourceManager()  
 retstr = io.StringIO()  
 layoutParams = pdfminer.layout.LAParams()  
 device = pdfminer.converter.TextConverter(rsrcmgr, retstr, laparams = layoutParams, codec = codec)  
 #We need a device and an interpreter  
 interpreter = pdfminer.pdfinterp.PDFPageInterpreter(rsrcmgr, device)  
 password = ''  
 maxpages = 0  
 caching = True  
 pagenos=set()  
 for page in pdfminer.pdfpage.PDFPage.get\_pages(pdfFile, pagenos, maxpages=maxpages, password=password,caching=caching, check\_extractable=True):  
 interpreter.process\_page(page)  
 device.close()  
 returnedString = retstr.getvalue()  
 retstr.close()  
 return returnedString

First we need to take the response object and convert it into a 'file like' object so that pdfminer can read it. To do this we will use io's BytesIO.

infoExtractionBytes = io.BytesIO(infoExtractionRequest.content)

Now we can give it to pdfminer.

print(readPDF(infoExtractionBytes)[:1000])

From here we can either look at the full text or fiddle with our PDF reader and get more information about individual blocks of text.

## Word Docs

The other type of document you are likely to encounter is the .docx, these are actually a version of [XML](https://en.wikipedia.org/wiki/Office_Open_XML), just like HTML, and like HTML we will use a specialized parser.

For this class we will use [python-docx](https://python-%20docx.readthedocs.io/en/latest/) which provides a nice simple interface for reading .docx files

docxURL = 'https://github.com/xiaow2/persp-analysis/raw/02772bc5baf4044ba6410170ca740f14cd6155d5/assignments/short%20paper%201.docx'  
  
r = requests.get(docxURL, stream=True)  
d = docx.Document(io.BytesIO(r.content))  
for paragraph in d.paragraphs[:7]:  
 print(paragraph.text)

This procedure uses the io.BytesIO class again, since docx.Document expects a file. Another way to do it is to save the document to a file and then read it like any other file. If we do this we can either delete the file afterwords, or save it and avoid downloading the following time.

def downloadIfNeeded(targetURL, outputFile, \*\*openkwargs):  
 if not os.path.isfile(outputFile):  
 outputDir = os.path.dirname(outputFile)  
 #This function is a more general os.mkdir()  
 os.makedirs(outputDir, exist\_ok = True)  
 r = requests.get(targetURL, stream=True)  
 #Using a closure like this is generally better than having to  
 #remember to close the file. There are ways to make this function  
 #work as a closure too  
 with open(outputFile, 'wb') as f:  
 f.write(r.content)  
 return open(outputFile, \*\*openkwargs)

This function will download, save and open outputFile as outputFile or just open it if outputFile exists. By default open() will open the file as read only text with the local encoding, which may cause issues if its not a text file.

d = docx.Document(downloadIfNeeded(docxURL, 'data/temp.docx'))

We need to tell open() to read in binary mode ('rb'), this is why we added \*\*openkwargs, this allows us to pass any keyword arguments (kwargs) from downloadIfNeeded to open().

d = docx.Document(downloadIfNeeded(docxURL, 'data/temp.docx', mode = 'rb'))  
for paragraph in d.paragraphs[:7]:  
 print(paragraph.text)

Now we can read the file with docx.Document and not have to wait for it to be downloaded every time.

## Image Files (using OCR)

Optical Character Recognition or OCR is the process of identifying and extracting text from image files. The best available OCR programs, however, may not (are not) available in python. Here, we will use tesseract, Google code written in C/C++ as a subprocess.

A subprocess, child process or subtask in computing is a process invoked by another process. Spawning such processes occurs in multitasking operating systems, and enables us to launch precombined code binaries of any type from within this python notebook. We will control and launch tesseract from pytesseract.

Lets look at reading the text from the tesseractTest.png file in the data directory.

testimage

testimage

First we need to open the image

#ocr\_file = 'data/tesseractTest.png'  
  
testImage = PIL.Image.open(ocr\_file)

Then we can read from it

print(pytesseract.image\_to\_string(testImage))

## *Your Turn*

Construct cells immediately below this that extract and organize textual content from text, PDF or Word, and images into a pandas dataframe.