

CS 109A/STAT 121A/AC 209A/CSCI E-109A: Homework 1

Harvard University

Fall 2017

Instructors: Pavlos Protopapas, Kevin Rader, Rahul Dave, Margo Levine

INSTRUCTIONS

WARNING: There is web page scraping in this homework. It takes about 40 minutes. **Do not wait till the last minute** to do this homework.

- To submit your assignment follow the instructions given in canvas.
 - Restart the kernel and run the whole notebook again before you submit. There is an important CAVEAT to this. DO NOT run the web-page fetching cells again. (We have provided hints like `# DO NOT RERUN THIS CELL WHEN SUBMITTING` on some of the cells where we provide the code). Instead load your data structures from the JSON files we will ask you to save below. Otherwise you will be waiting for a long time. (Another reason to not wait until the last moment to submit.)
 - Do not include your name in the notebook.
-

Homework 1: Rihanna or Mariah?

Billboard Magazine puts out a top 100 list of "singles" every week. Information from this list, as well as that from music sales, radio, and other sources is used to determine a top-100 "singles" of the year list. A **single** is typically one song, but sometimes can be two songs which are on one "single" record.

In this homework you will:

1. Scrape Wikipedia to obtain information about the best singers and groups from each year (distinguishing between the two groups) as determined by the Billboard top 100 charts. You will have to clean this data. Along the way you will learn how to save data in json files to avoid repeated scraping.
2. Scrape Wikipedia to obtain information on these singers. You will have to scrape the web pages, this time using a cache to guard against network timeouts (or your laptop going to sleep). You will again clean the data, and save it to a json file.
3. Use pandas to represent these two datasets and merge them.
4. Use the individual and merged datasets to visualize the performance of the artists and their songs. We have kept the amount of analysis limited here for reasons of time; but you might enjoy exploring music genres and other aspects of the music business you can find on these wikipedia pages at your own leisure.

You should have worked through Lab0 and Lab 1, and Lecture 2. Lab 2 will help as well.

As usual, first we import the necessary libraries. In particular, we use Seaborn (<http://stanford.edu/~mwaskom/software/seaborn/>) to give us a nicer default color palette, with our plots being of large (poster) size and with a white-grid background.

```
In [2]: %matplotlib inline
import functools
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import pandas as pd
import time
pd.set_option('display.width', 500)
pd.set_option('display.max_columns', 100)
pd.set_option('display.notebook_repr_html', True)
import seaborn as sns
```

Q1. Scraping Wikipedia for Billboard Top 100.

In this question you will scrape Wikipedia for the Billboard's top 100 singles.

Scraping Wikipedia for Billboard singles

We'll be using [BeautifulSoup](http://www.crummy.com/software/BeautifulSoup/) (<http://www.crummy.com/software/BeautifulSoup/>), and suggest that you use Python's built in `requests` library to fetch the web page.

1.1 Parsing the Billboard Wikipedia page for 1970

Obtain the web page at http://en.wikipedia.org/wiki/Billboard_Year-End_Hot_100_singles_of_1970 (http://en.wikipedia.org/wiki/Billboard_Year-End_Hot_100_singles_of_1970) using a HTTP GET request. From this web page we'll extract the top 100 singles and their rankings. Create a list of dictionaries, 100 of them to be precise, with entries like

```
{ 'url': '/wiki/Sugarloaf_(band)', 'ranking': 30, 'band_singer':  
'Sugarloaf', 'title': 'Green-Eyed Lady' }.
```

If you look at that web page, you'll see a link for every song, from which you can get the `url` of the singer or band. We will use these links later to scrape information about the singer or band. From the listing we can also get the band or singer name `band_singer`, and `title` of the song.

HINT: look for a table with class `wikitable`.

You should get something similar to this (where songs is the aforementioned list):

```
songs[2:4]  
  
[{'band_singer': 'The Guess Who',  
  'ranking': 3,  
  'title': '"American Woman"',  
  'url': '/wiki/The_Guess_Who'},  
 {'band_singer': 'B.J. Thomas',  
  'ranking': 4,  
  'title': '"Raindrops Keep Fallin\' on My Head"',  
  'url': '/wiki/B.J._Thomas'}]
```

```
In [3]: import requests  
        from bs4 import BeautifulSoup  
        from IPython.display import IFrame, HTML  
        import time
```

```
In [4]: # Get Wiki page for 1970's top 100
req = requests.get(" http://en.wikipedia.org/wiki/Billboard_Year-End_Hot_
page = req.text
# Use BeautifulSoup to convert the HTML
soup = BeautifulSoup(page, 'html.parser')
# The classes of all tables that have a class attribute set on them
[t["class"] for t in soup.find_all("table") if t.get("class")]
# Find the table with class-types 'sortable' and 'wikitable'
dfinder = lambda tag: tag.name=='table' and tag.get('class') == ['wikitab
table_songs = soup.find_all(dfinder)
```

```
In [5]: # Extract rows from table_demographics
rows = [row for row in table_songs[0].find_all("tr")]

# Insert table data into dictionary
songs = []
for row in rows[1:]:
    entries = row.find_all("td")
    if entries[1].find("a"):
        songEntry = {'band_singer': entries[2].text, 'ranking': int(enti
                        'title': ('' + entries[1].find("a").get("title") + '')
        # if song doesn't have URL
    else:
        songEntry = {'band_singer': entries[2].text, 'ranking': int(enti
                        'title': ('' + entries[1].text + ''), 'url': entries[2
        songs.append(songEntry)
songs[2:4]
```

```
Out[5]: [{'band_singer': 'The Guess Who',
          'ranking': 3,
          'title': '"American Woman"',
          'url': '/wiki/The_Guess_Who'},
         {'band_singer': 'B.J. Thomas',
          'ranking': 4,
          'title': '"Raindrops Keep Fallin\' on My Head"',
          'url': '/wiki/B.J._Thomas'}]
```

1.2 Generalize the previous: scrape Wikipedia from 1992 to 2014

By visiting the urls similar to the ones for 1970, we can obtain the billboard top 100 for the years 1992 to 2014. (We choose these later years rather than 1970 as you might find music from this era more interesting.) Download these using Python's `requests` module and store the text from those requests in a dictionary called `yearstext`. This dictionary ought to have as its keys the years (as integers from 1992 to 2014), and as values corresponding to these keys the text of the page being fetched.

You ought to sleep a second (look up `time.sleep` in Python) at the very least in-between fetching each web page: you do not want Wikipedia to think you are a marauding bot attempting to mount a denial-of-service attack.

HINT: you might find `range` and `string-interpolation` useful to construct the URLs .

```
In [5]: years = range(1992, 2015)
text = []
for year in range(1992, 2015):
    req = requests.get(" http://en.wikipedia.org/wiki/Billboard_Year-End_
    text.append(req.text)
    time.sleep(1)
yearstext = dict(zip(years,text))
```

1.3 Parse and Clean data

Remember the code you wrote to get data from 1970 which produces a list of dictionaries, one corresponding to each single. Now write a function `parse_year(the_year, yeartext_dict)` which takes the year, prints it out, gets the text for the year from the just created `yearstext` dictionary, and return a list of dictionaries for that year, with one dictionary for each single. Store this list in the variable `yearinfo`.

The dictionaries **must** be of this form:

```
{'band_singer': ['Brandy', 'Monica'],
 'ranking': 2,
 'song': ['The Boy Is Mine'],
 'songurl': ['/wiki/The_Boy_Is_Mine_(song)'],
 'titletext': '" The Boy Is Mine "',
 'url': ['/wiki/Brandy_Norwood', '/wiki/Monica_(entertainer)']}
```

The spec of this function is provided below:

```
In [6]: """
function
-----
parse_year

Inputs
-----
the_year: the year you want the singles for
yeartext_dict: a dictionary with keys as integer years and values the downl.
               from wikipedia for that year.

Returns
-----

a list of dictionaries, each of which corresponds to a single and has the
following data:

eg:
```

```
'band_singer': ['Brandy', 'Monica'],
'ranking': 2,
'song': ['The Boy Is Mine'],
'songurl': ['/wiki/The_Boy_Is_Mine_(song)'],
'titletext': '" The Boy Is Mine "',
'url': ['/wiki/Brandy_Norwood', '/wiki/Monica_(entertainer)']}]}
```

A dictionary with the following data:

band_singer: a list of bands/singers who made this single

song: a list of the titles of songs on this single

songurl: a list of the same size as song which has urls for the songs (see point 3 above)

ranking: ranking of the single

titletext: the contents of the table cell

band_singer: a list of bands or singers on this single

url: a list of wikipedia singer/band urls on this single: only put in of the url from /wiki onwards

Notes

See description and example above.

"""

```
def parse_year(the_year, yeartext_dict):
```

```
    page = yeartext_dict[the_year]
```

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

```
    # Find the table with class-types 'sortable' and 'wikitable'
```

```
    table_songs = soup.find_all(dfinder)
```

```
    # Extract rows from table_demographics
```

```
    rows = [row for row in table_songs[0].find_all("tr")]
```

```
    # Insert table data into dictionary
```

```
    songs = []
```

```
    counter = 0
```

```
    for row in rows[1:]:
```

```
        counter += 1
```

```
        entries = row.find_all("td")
```

```
        # if song has URL
```

```
        if entries[0].find("a"):
```

```
            songurl = list(map(lambda x: x.get("href"), entries[0].find_all("a")))
```

```
            song = list(map(lambda x: x.text, entries[0].find_all("a")))
```

```
            titletext = functools.reduce((lambda x, y: x + " / " + y), (list
```

```
        else:
```

```
            songurl = [None]
```

```
            song = [entries[0].text.replace(' ', '')]
```

```
            titletext = [entries[0].text]
```

```
        # if artist has URL
```

```
        if entries[1].find("a"):
```

```
            url = list(map(lambda x: x.get("href"), entries[1].find_all("a")))
```

```
            band_singer = list(map(lambda x: x.text, entries[1].find_all("a")))
```

```
        else:
```

```
            url = [None]
```

```
            band_singer = [entries[1].text]
```

```
    songEntry = {'band singer': band_singer, 'ranking': counter, 'song':
```

```

        songs.append(songEntry)
    return songs

yearinfo = []

for year in years:
    info = parse_year(year, yearstext)
    yearinfo.append(info)

```

Helpful notes

Notice that some singles might have multiple songs:

```

{'band_singer': ['Jewel'],
 'ranking': 2,
 'song': ['Foolish Games', 'You Were Meant for Me'],
 'songurl': ['/wiki/Foolish_Games',
             '/wiki/You_Were_Meant_for_Me_(Jewel_song)'],
 'titletext': '" Foolish Games " / " You Were Meant for Me "',
 'url': ['/wiki/Jewel_(singer)']}

```

And some singles don't have a song URL:

```

{'band_singer': [u'Nu Flavor'],
 'ranking': 91,
 'song': [u'Heaven'],
 'songurl': [None],
 'titletext': u'"Heaven"',
 'url': [u'/wiki/Nu_Flavor']}

```

Thus there are some issues this function must handle:

1. There can be more than one `band_singer` as can be seen above (sometimes with a comma, sometimes with "featuring" in between). The best way to parse these is to look for the urls.
2. There can be two songs in a single, because of the way the industry works: there are two-sided singles. See https://en.wikipedia.org/wiki/Billboard_Year-End_Hot_100_singles_of_1997 (https://en.wikipedia.org/wiki/Billboard_Year-End_Hot_100_singles_of_1997) for an example. You can find other examples in 1998 and 1999.
3. The `titletext` is the contents of the table cell, and retains the quotes that Wikipedia puts on the single.
4. If no song anchor is found (see the 24th song in the above url), assume there is one song in the single, set `songurl` to `[None]` and the song name to the contents of the table cell with the quotes stripped (ie `song` is a one-element list with this the `titletext` stripped of its quotes).

As a check, we can do this for 1997. We'll print the first 5 outputs: `parse_year(1997, yearstext)[:5]`

This should give the following. Notice that the year 1997 exercises the edge cases we talked about earlier.


```

[{'band_singer': ['Elton John'],
  'ranking': 1,
  'song': ['Something About the Way You Look Tonight',
  'Candle in the Wind 1997'],
  'songurl': ['/wiki/Something_About_the_Way_You_Look_Tonight',
  '/wiki/Candle_in_the_Wind_1997'],
  'titletext': '" Something About the Way You Look Tonight " / "
Candle in the Wind 1997 "',
  'url': ['/wiki/Elton_John']},
{'band_singer': ['Jewel'],
  'ranking': 2,
  'song': ['Foolish Games', 'You Were Meant for Me'],
  'songurl': ['/wiki/Foolish_Games',
  '/wiki/You_Were_Meant_for_Me_(Jewel_song)'],
  'titletext': '" Foolish Games " / " You Were Meant for Me "',
  'url': ['/wiki/Jewel_(singer)']},
{'band_singer': ['Puff Daddy', 'Faith Evans', '112'],
  'ranking': 3,
  'song': ["I'll Be Missing You"],
  'songurl': ['/wiki/I%27ll_Be_Missing_You'],
  'titletext': '" I\'ll Be Missing You "',
  'url': ['/wiki/Sean_Combs', '/wiki/Faith_Evans', '/wiki/112_(ba
nd)']},
{'band_singer': ['Toni Braxton'],
  'ranking': 4,
  'song': ['Un-Break My Heart'],
  'songurl': ['/wiki/Un-Break_My_Heart'],
  'titletext': '" Un-Break My Heart "',
  'url': ['/wiki/Toni_Braxton']},
{'band_singer': ['Puff Daddy', 'Mase'],
  'ranking': 5,
  'song': ["Can't Nobody Hold Me Down"],
  'songurl': ['/wiki/Can%27t_Nobody_Hold_Me_Down'],
  'titletext': '" Can\'t Nobody Hold Me Down "',
  'url': ['/wiki/Sean_Combs', '/wiki/Mase']}]

```

```
In [7]: parse_year(1997, yearstext)[:5]
```

```
Out[7]: [{ 'band_singer': ['Elton John'],
  'ranking': 1,
  'song': ['Something About the Way You Look Tonight',
  'Candle in the Wind 1997'],
  'songurl': ['/wiki/Something_About_the_Way_You_Look_Tonight',
  '/wiki/Candle_in_the_Wind_1997'],
  'titletext': '"Something About the Way You Look Tonight" / "Candle i
n the Wind 1997"',
  'url': ['/wiki/Elton_John']},
{ 'band_singer': ['Jewel'],
  'ranking': 2,
  'song': ['Foolish Games', 'You Were Meant for Me'],
  'songurl': ['/wiki/Foolish_Games',
  '/wiki/You_Were_Meant_for_Me_(Jewel_song)'],
  'titletext': '"Foolish Games" / "You Were Meant for Me (Jewel song)"
',
  'url': ['/wiki/Jewel_(singer)']},
{ 'band_singer': ['Puff Daddy', 'Faith Evans', '112'],
  'ranking': 3,
  'song': ["I'll Be Missing You"],
  'songurl': ['/wiki/I%27ll_Be_Missing_You'],
  'titletext': '"I\'ll Be Missing You"',
  'url': ['/wiki/Sean_Combs', '/wiki/Faith_Evans', '/wiki/112_(band)']
},
{ 'band_singer': ['Toni Braxton'],
  'ranking': 4,
  'song': ['Un-Break My Heart'],
  'songurl': ['/wiki/Un-Break_My_Heart'],
  'titletext': '"Un-Break My Heart"',
  'url': ['/wiki/Toni_Braxton']},
{ 'band_singer': ['Puff Daddy', 'Mase'],
  'ranking': 5,
  'song': ["Can't Nobody Hold Me Down"],
  'songurl': ['/wiki/Can%27t_Nobody_Hold_Me_Down'],
  'titletext': '"Can\'t Nobody Hold Me Down"',
  'url': ['/wiki/Sean_Combs', '/wiki/Mase']]}
```

Save a json file of information from the scraped files

We do not want to lose all this work, so let's save the last data structure we created to disk. That way if you need to re-run from here, you don't need to redo all these requests and parsing.

DO NOT RERUN THE HTTP REQUESTS TO WIKIPEDIA WHEN SUBMITTING.

We **DO NOT** need to see these JSON files in your submission!

```
In [8]: import json
```

```
In [8]: # DO NOT RERUN THIS CELL WHEN SUBMITTING
# fd = open("yearinfo.json", "w")
# json.dump(yearinfo, fd)
# fd.close()
# del yearinfo
```

Now let's reload our JSON file into the yearinfo variable, just to be sure everything is working.

```
In [56]: # RERUN WHEN SUBMITTING
# Another way to deal with files. Has the advantage of closing the file f
with open("yearinfo.json", "r") as fd:
    yearinfo = json.load(fd)
```

1.4 Construct a year-song-singer dataframe from the yearly information

Let's construct a dataframe `flatframe` from the `yearinfo`. The frame should be similar to the frame below. Each row of the frame represents a song, and carries with it the chief properties of year, song, singer, and ranking.

	year	band_singer	ranking	song	songurl	url
0	1992	Boyz II Men	1.0	End of the Road	/wiki/End_of_the_Road	/wiki/Boyz_II_Men
1	1993	Whitney Houston	1.0	I Will Always Love You	/wiki/I_Will_Always_Love_You#Whitney_Houston_v...	/wiki/Whitney_Houston
2	1994	Ace of Base	1.0	The Sign (song)	/wiki/The_Sign_(song)	/wiki/Ace_of_Base
3	1995	Coolio	1.0	Gangsta's Paradise	/wiki/Gangsta%27s_Paradise	/wiki/Coolio
4	1996	Los del Río	1.0	Macarena (song)	/wiki/Macarena_(song)	/wiki/Los_del_R%C3%ADo
5	1997	Elton John	1.0	Something About the Way You Look Tonight	/wiki/Something_About_the_Way_You_Look_Tonight	/wiki/Elton_John
6	1998	Next (group)	1.0	Too Close (Next song)	/wiki/Too_Close_(Next_song)	/wiki/Next_(group)
7	1999	Cher	1.0	Believe (Cher song)	/wiki/Believe_(Cher_song)	/wiki/Cher

To construct the dataframe, we'll need to iterate over the years and the singles per year. Notice how, above, the dataframe is ordered by ranking and then year. While the exact order is up to you, note that you will have to come up with a scheme to order the information.

Check that the dataframe has sensible data types. You will also likely find that the year field has become an "object" (Pandas treats strings as generic objects): this is due to the conversion to and back from JSON. Such conversions need special care. Fix any data type issues with `flatframe`. (See Pandas [astype](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.astype.html) (<http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.astype.html>) function.) We will use this `flatframe` in the next question.

(As an aside, we used the name `flatframe` to indicate that this dataframe is flattened from a hierarchical dictionary structure with the keys being the years.)

```
In [10]: # Rebuild yearinfo as a list of list of dictionaries that can be easily r
new_yearinfo = []
# loop through every singer, for every song, of every year
for year in range(len(yearinfo)):
    yearEntry = []
    for rank in range(len(yearinfo[1])):
        counter = 0
        for singer in yearinfo[year][rank]['band_singer']:
            # rebuild dictionary entry in usable format for dataframe
            date = year + 1992
            ranking = rank + 1
            song = yearinfo[year][rank]['song'][0]
            url = yearinfo[year][rank]['url'][counter]
            counter += 1
            entry = {'year': date, 'band_singer': singer, 'song': song, '
            yearEntry.append(entry)
        new_yearinfo.append(yearEntry)

# build base dataframe from 1992
flatframe = pd.DataFrame.from_dict(new_yearinfo[0])
# append to dataframe additional years
for i in range(1, len(new_yearinfo)):
    flatframe = flatframe.append(pd.DataFrame.from_dict(new_yearinfo[i]),

# sort by ranking and year, and re-index and change column order
flatframe = flatframe.sort_values(by=['ranking', 'year']).reset_index(dro
    ['year', 'band_singer', 'ranking', 'song', 'url'])
```

Who are the highest quality singers?

Here we show the highest quality singers and plot them on a bar chart.

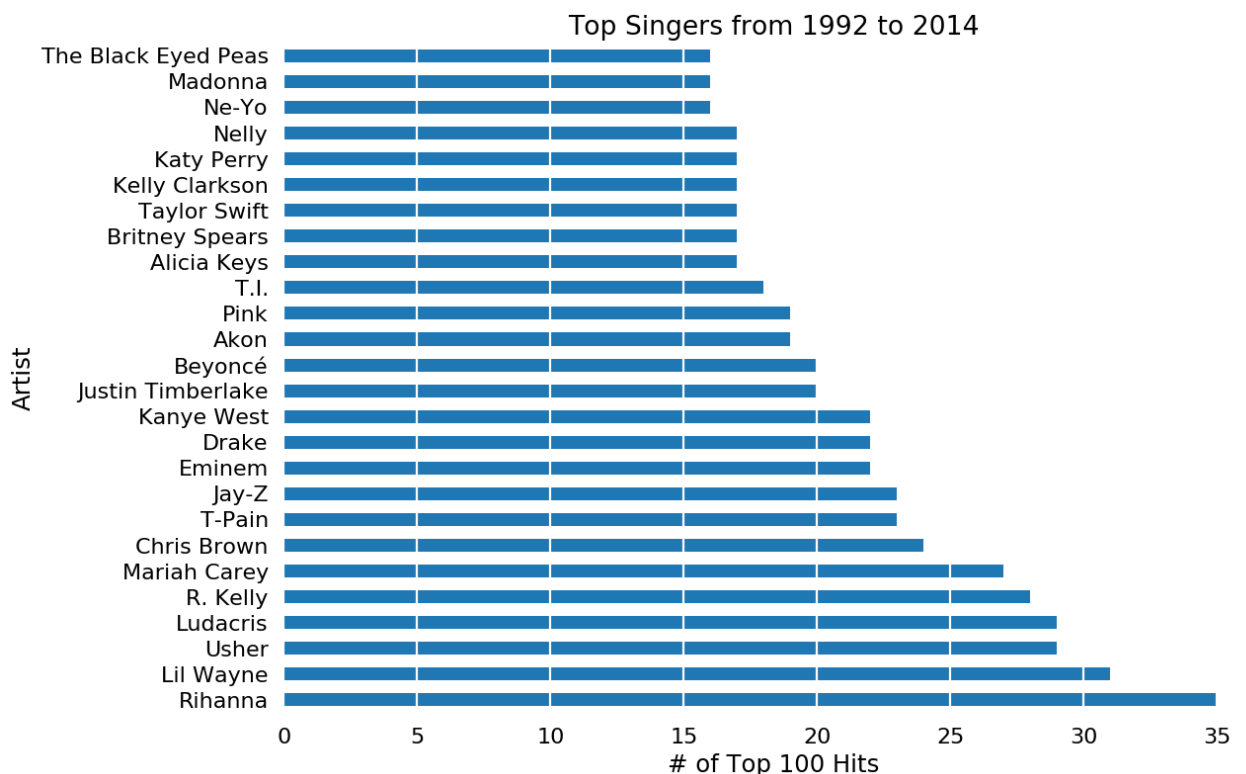
1.5 Find highest quality singers according to how prolific they are

What do we mean by highest quality? This is of course open to interpretation, but let's define "highest quality" here as the number of times a singer appears in the top 100 over this time period. If a singer appears twice in a year (for different songs), this is counted as two appearances, not one.

Make a bar-plot of the most prolific singers. Singers on this chart should have appeared at-least more than 15 times. (HINT: look at the docs for the pandas method `value_counts`.)

```
In [11]: #Creates a series of the top 15 ranked singers/bands
singers = flatframe.band_singer.value_counts()
best_singers = singers[singers > 15]

#Reset the defaults to be safe and plot graph
sns.reset_defaults()
with sns.plotting_context("poster"):
    ax = plt.gca()
    best_singers.plot(kind="barh")
    plt.grid(axis = 'x', color = 'white', linestyle='-')
    ax.tick_params(axis='both', which='both',length=0)
    sns.despine(left=True, bottom=True)
    plt.title("Top Singers from 1992 to 2014")
    plt.ylabel("Artist")
    plt.xlabel("# of Top 100 Hits")
    plt.show()
```



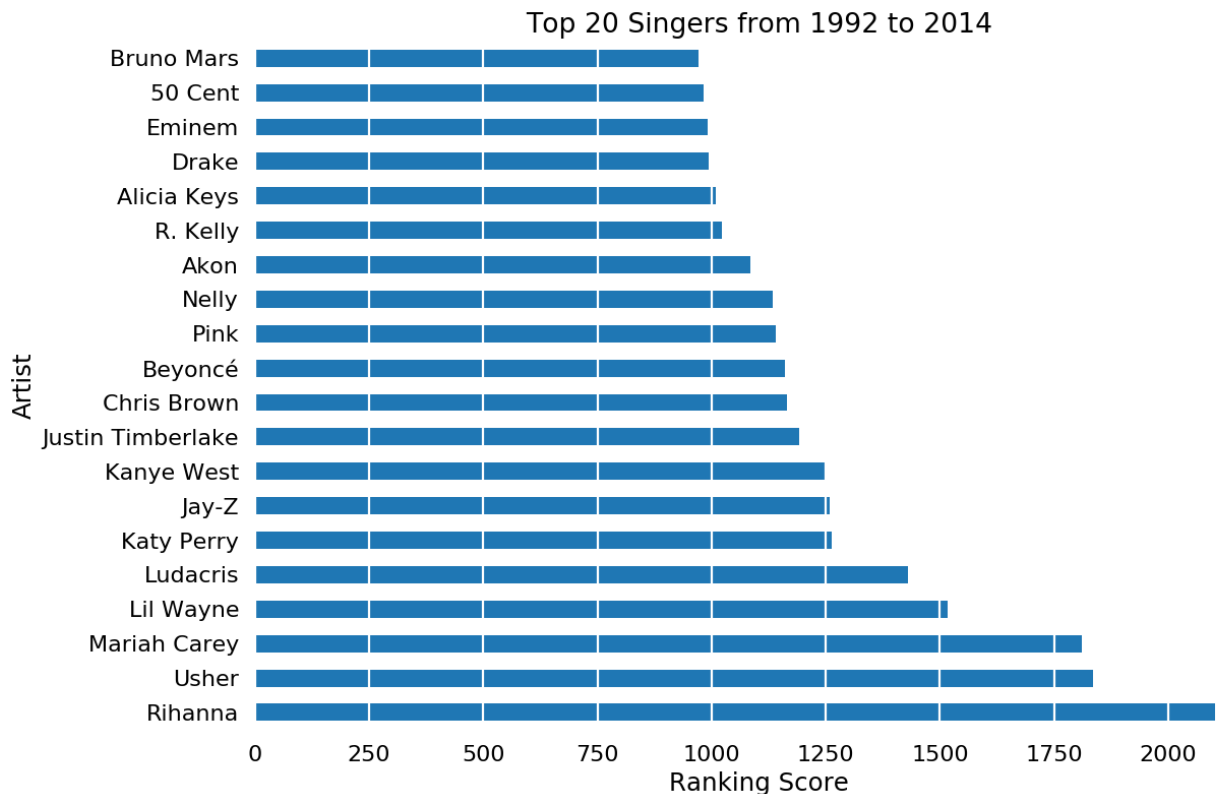
1.6 What if we used a different metric?

What we would like to capture is this: a singer should to be scored higher if the singer appears higher in the rankings. So we'd say that a singer who appeared once at a higher and once at a lower ranking is a "higher quality" singer than one who appeared twice at a lower ranking.

To do this, group all of a singers songs together and assign each song a score $101 - \text{ranking}$. Order the singers by their total score and make a bar chart for the top 20.

```
In [12]: #Create the rankings 101 column
flatframe['rankings101'] = flatframe['ranking'].map(lambda x: 101 -x)
#Series of each band/singers ranking sum
top20 = flatframe.groupby('band_singer').sum().sort_values('rankings101',

#Reset defaults to be safe and plot the graph
sns.reset_defaults()
with sns.plotting_context("poster"):
    ax = plt.gca()
    top20.plot(kind="barh")
    plt.grid(axis = 'x', color = 'white', linestyle='-')
    ax.tick_params(axis='both', which='both', length=0)
    sns.despine(left=True, bottom=True)
    plt.title("Top 20 Singers from 1992 to 2014")
    plt.ylabel("Artist")
    plt.xlabel("Ranking Score")
    plt.show()
```



1.7 Do you notice any major differences when you change the metric?

How have the singers at the top shifted places? Why do you think this happens?

Various artists have shifted ranking using the different methodologies. This happens because some artists produce a lot of good songs, but those songs are not necessarily the best songs for that year. Some artists like Rihanna who is at the top of both charts, produce many excellent songs that are at the top of the charts. On the other hand, R. Kelly and T.I produce a lot of songs on that make it on the top 100 list, but they are not all excellent songs at the top of the charts. Finally, artists like Bruno Mars have a few really excellent songs so they rank on the second methodology but they don't have many songs so they do not rank well on the first methodology.

Q2. Scraping and Constructing: Information about Artists, Bands and Genres from Wikipedia

Our next job is to use those band/singer urls we collected under `flatframe.url` and get information about singers and/or bands.

Scrape information about artists from wikipedia

We wish to fetch information about the singers or groups for all the winning songs in a list of years.

Here we show a function that fetches information about a singer or group from their url on wikipedia. We create a cache object `urlcache` that will avoid redundant HTTP requests (e.g. an artist might have multiple singles on a single year, or be on the list over a span of years). Once we have fetched information about an artist, we don't need to do it again. The caching also helps if the network goes down, or the target website is having some problems. You simply need to run the `get_page` function below again, and the `urlcache` dictionary will continue to be filled.

If the request gets an HTTP return code different from 200, (such as a 404 not found or 500 Internal Server Error) the cells for that URL will have a value of 1; and if the request completely fails (e.g. no network connection) the cell will have a value of 2. This will allow you to analyse the failed requests.

Notice that we have wrapped the call in whats called *an exception block*. We try to make the request. If it fails entirely, or returns a HTTP code thats not 200, we set the status to 2 and 1 respectively.

```
In [9]: urlcache={}
```

```
In [10]: def get_page(url):
# Check if URL has already been visited.
if (url not in urlcache) or (urlcache[url]==1) or (urlcache[url]==2):
time.sleep(1)
# try/except blocks are used whenever the code could generate an
# In this case we don't know if the page really exists, or even if
try:
r = requests.get("http://en.wikipedia.org%s" % url)

if r.status_code == 200:
urlcache[url] = r.text
else:
urlcache[url] = 1
except:
urlcache[url] = 2
return urlcache[url]
```

We sort the flatframe by year, ascending, first. Think why.

```
In [13]: flatframe=flatframe.sort_values('year')
flatframe.head()
```

Out[13]:

	year	band_singer	ranking	song	url	rankings101
0	1992	Boyz II Men	1	End of the Road	/wiki/Boyz_II_Men	100
1879	1992	Karyn White	63	The Way I Feel About You	/wiki/Karyn_White	38
1849	1992	Genesis	62	Hold on My Heart	/wiki/Genesis_(band)	39
1824	1992	CeCe Peniston	61	Keep on Walkin'	/wiki/CeCe_Peniston	40
187	1992	En Vogue	7	My Lovin' (You're Never Gonna Get It)	/wiki/En_Vogue	94

Pulling and saving the data

```
In [ ]: # DO NOT RERUN THIS CELL WHEN SUBMITTING
# Here we are populating the url cache
# subsequent calls to this cell should be very fast, since Python won't
# need to fetch the page from the web server.
# NOTE this function will take quite some time to run (about 30 mins for
# making a request. If you run it again it will be almost instantaneous,
# (you will need to run it again if requests fail..see cell below for how
# flatframe["url"].apply(get_page)
```


You may have to run this function again and again, in case there were network problems. Note that, because there is a "global" cache, it will take less time each time you run it. Also note that this function is designed to be run again and again: it attempts to make sure that there are no unresolved pages remaining. Let us make sure of this: *the sum below should be 0, and the boolean True.*

```
In [ ]: # DO NOT RERUN THIS CELL WHEN SUBMITTING
# print("Number of bad requests:", np.sum([(urlcache[k]==1) or (urlcache[k]
# print("Did we get all urls?", len(flatframe.url.unique())==len(urlcache
```

Let's save the urlcache to disk, just in case we need it again.

```
In [ ]: # DO NOT RERUN THIS CELL WHEN SUBMITTING
# with open("data/artistinfo.json", "w") as fd:
#     json.dump(urlcache, fd)
# del urlcache
```

```
In [14]: # RERUN WHEN SUBMITTING
with open("artistinfo.json") as json_file:
    urlcache = json.load(json_file)
```

2.1 Extract information about singers and bands

From each page we collected about a singer or a band, extract the following information:

1. If the page has the text "Born" in the sidebar on the right, extract the element with the class `.bday`. If the page doesn't contain "Born", store `False`. Store either of these into the variable `born`. We want to analyze the artist's age.
2. If the text "Years active" is found, but no "born", assume a band. Store into the variable `ya` the value of the next table cell corresponding to this, or `False` if the text is not found.


Put this all into a function `singer_band_info` which takes the singer/band url as argument and returns a dictionary `dict(url=url, born=born, ya=ya)`.

The information can be found on the sidebar on each such wikipedia page, as the example here shows:

ie most
and
nd "The
their
ums.
ian

, under
ards,
,
again
radio in
touring
was
l
cting
ar
ge over

Simon & Garfunkel



Simon (right) and Garfunkel performing in [Dublin](#) in 1982

Background information	
Origin	Forest Hills, Queens, New York City, U.S.
Genres	Folk rock ^[1]
Years active	1957–1965, 1966–1970 (breakup) (Reunions: 1975, 1981–83, 1993, 2003–04, 2009–10)
Labels	Columbia
Website	simonandgarfunkel.com 
Best	Paul Simon

Write the function `singer_band_info` according to the following specification:

```

In [15]: """
Function
-----
singer_band_info

Inputs
-----
url: the url
page_text: the text associated with the url

Returns
-----
A dictionary with the following data:
    url: copy the input argument url into this value
    born: the artist's birthday
    ya: years active variable

Notes
-----
See description above. Also note that some of the genres urls might requi
bit of care and special handling.
"""

def singer_band_info (url, page_text):
    if page_text == 2:
        return
    soup = BeautifulSoup(page_text, 'html.parser')
    bday = soup.find_all(lambda tag: tag.name=='span' and tag.get('class'
    if bday:
        born = bday[0].text
        ya = False
    else:
        table = soup.find_all(lambda tag: tag.name=='table' and tag.get('
        if table:
            yearsActive = [row.find('td').text for row in table[0].find_a
            if yearsActive:
                ya = yearsActive[0]
            else:
                ya = False
            born = False
        else:
            return
    return {'url':url, 'born':born,'ya': ya }

```

2.2 Merging this information in

Iterate over the items in the singer-group dictionary cache `urlcache`, run the above function, and create a dataframe from there with columns `url`, `born`, and `ya`. Merge this dataframe on the `url` key with `flatframe`, creating a rather wide dataframe that we shall call `largedf`. It should look something like this:

	year	band_singer	ranking	song	songurl	url	born	ya
0	1992	Boyz II Men	1.0	End of the Road	/wiki/End_of_the_Road	/wiki/Boyz_II_Men	False	1985- prese
1	1992	Boyz II Men	37.0	It's So Hard to Say Goodbye to Yesterday	/wiki/It%27s_So_Hard_to_Say_Goodbye_to_Yesterday	/wiki/Boyz_II_Men	False	1985- prese
2	1992	Boyz II Men	84.0	Uhh Ahh	/wiki/Uhh_Ahh	/wiki/Boyz_II_Men	False	1985- prese
3	1993	Boyz II Men	12.0	In the Still of the Night (1956 song)	/wiki/In_the_Still_of_the_Night_(1956_song)#Bo...	/wiki/Boyz_II_Men	False	1985- prese
4	1994	Boyz II Men	3.0	I'll Make Love to You	/wiki/I%27ll_Make_Love_to_You	/wiki/Boyz_II_Men	False	1985- prese

Notice how the `born` and `ya` and `url` are repeated every time a different song from a given band is represented in a row.

```
In [16]: #Makes a dataframe of the url, born, and ya to append to flatframe
dicCreated = False
for key in urlcache:
    if not dicCreated:
        df = pd.DataFrame([singer_band_info(key, urlcache[key])])
        dicCreated = True
    else:
        df = df.append([singer_band_info(key, urlcache[key])])
del df[0]
df = df.dropna(axis=0)
```

```
In [17]: #Join flatframe and df to create largedf
largedf = flatframe.join(df.set_index('url'), on='url')
largedf
```

Out[17]:

	year	band_singer	ranking	song	url	rankings101	born
0	1992	Boyz II Men	1	End of the Road	/wiki/Boyz_II_Men	100	False
1879	1992	Karyn White	63	The Way I Feel About You	/wiki/Karyn_White	38	1965-10-14
1849	1992	Genesis	62	Hold on My Heart	/wiki/Genesis_(band)	39	False
1824	1992	CeCe Peniston	61	Keep on Walkin'	/wiki/CeCe_Peniston	40	1969-09-06
187	1992	En Vogue	7	My Lovin' (You're Never Gonna Get It)	/wiki/En_Vogue	94	False

2.3 What is the age at which singers achieve their top ranking?

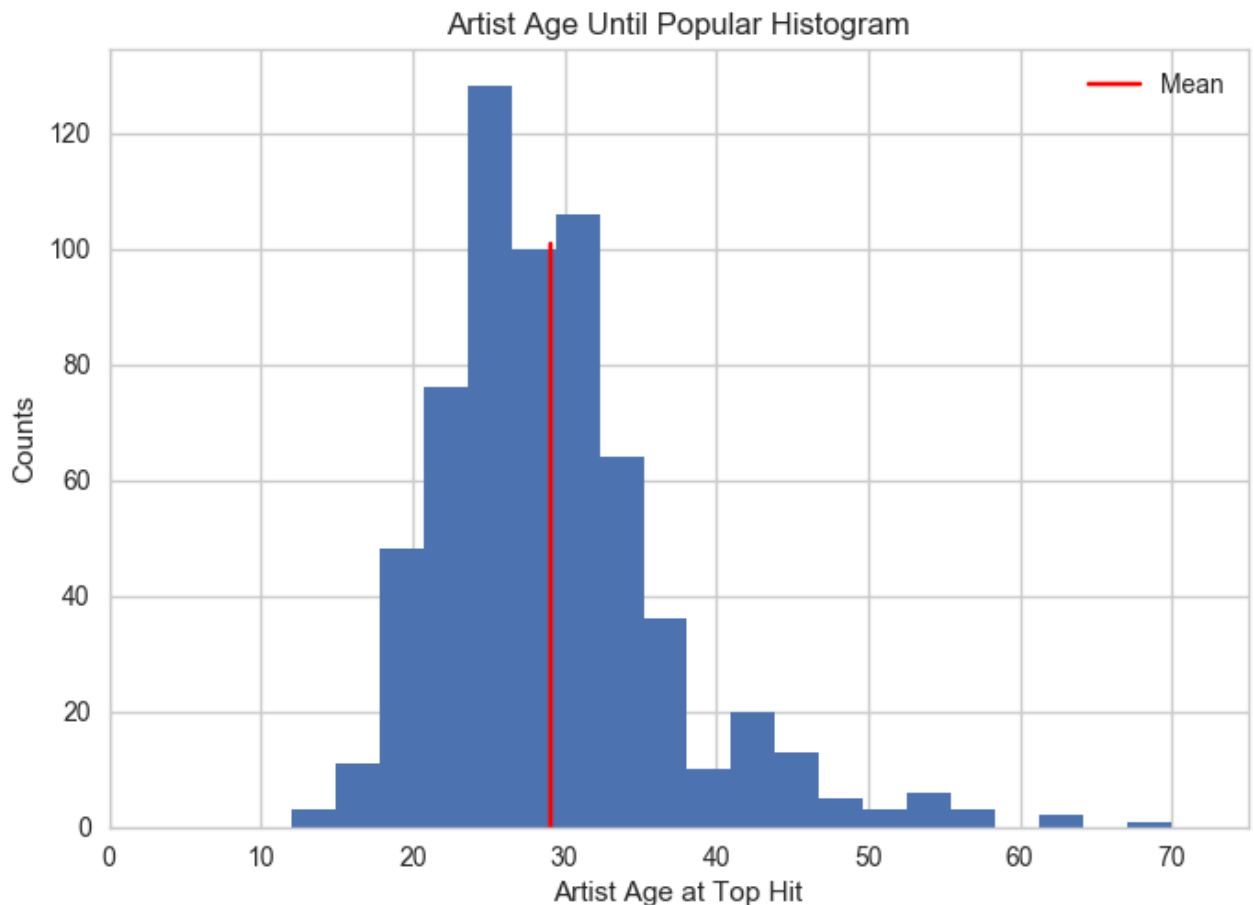
Plot a histogram of the age at which singers achieve their top ranking. What conclusions can you draw from this distribution of ages?

HINT: You will need to do some manipulation of the born column, and find the song for which a band or an artist achieves their top ranking. You will then need to put these rows together into another dataframe or array to make the plot.

```
In [19]: #Get each artists top hit in a list topHits
singerGroups = largedf.groupby('band_singer')
topHitsFrame = False
for group, data in singerGroups:
    if not topHitsFrame:
        topHits = data.sort_values('ranking').head(1)
        topHitsFrame = True
    else:
        topHits = topHits.append(data.sort_values('ranking').head(1))
```

```
In [52]: #clear NaN's and calculate age of artist at top hit
artistAge = topHits[['year', 'born']]
artistAge = artistAge[artistAge['born'] != False].dropna(axis=0)
artistAge['born'] = artistAge['born'].map(lambda x: int(x[:4]))
artistAge['age'] = artistAge['year'].astype(int) - artistAge['born']

#Plot graph
with sns.axes_style("whitegrid"):
    plt.hist(artistAge['age'], bins=20)
    plt.xlim(0, 75)
    plt.axvline(artistAge['age'].mean(), 0, 0.75, color='r', label='Mean')
    plt.xlabel("Artist Age at Top Hit")
    plt.ylabel("Counts")
    plt.title("Artist Age Until Popular Histogram")
    plt.legend()
    plt.show()
```



From the distribution, it is quite apparent that artists tend to have their first hit at 30 years old, on average. It also shows that the majority of popular artists are fairly young (in their late teens to 30's), and as an artist gets older, they are much less statistically likely to have a popular song.

2.4 At what year since inception do bands reach their top rankings?

Make a similar calculation to plot a histogram of the years since inception at which bands reach their top ranking. What conclusions can you draw?

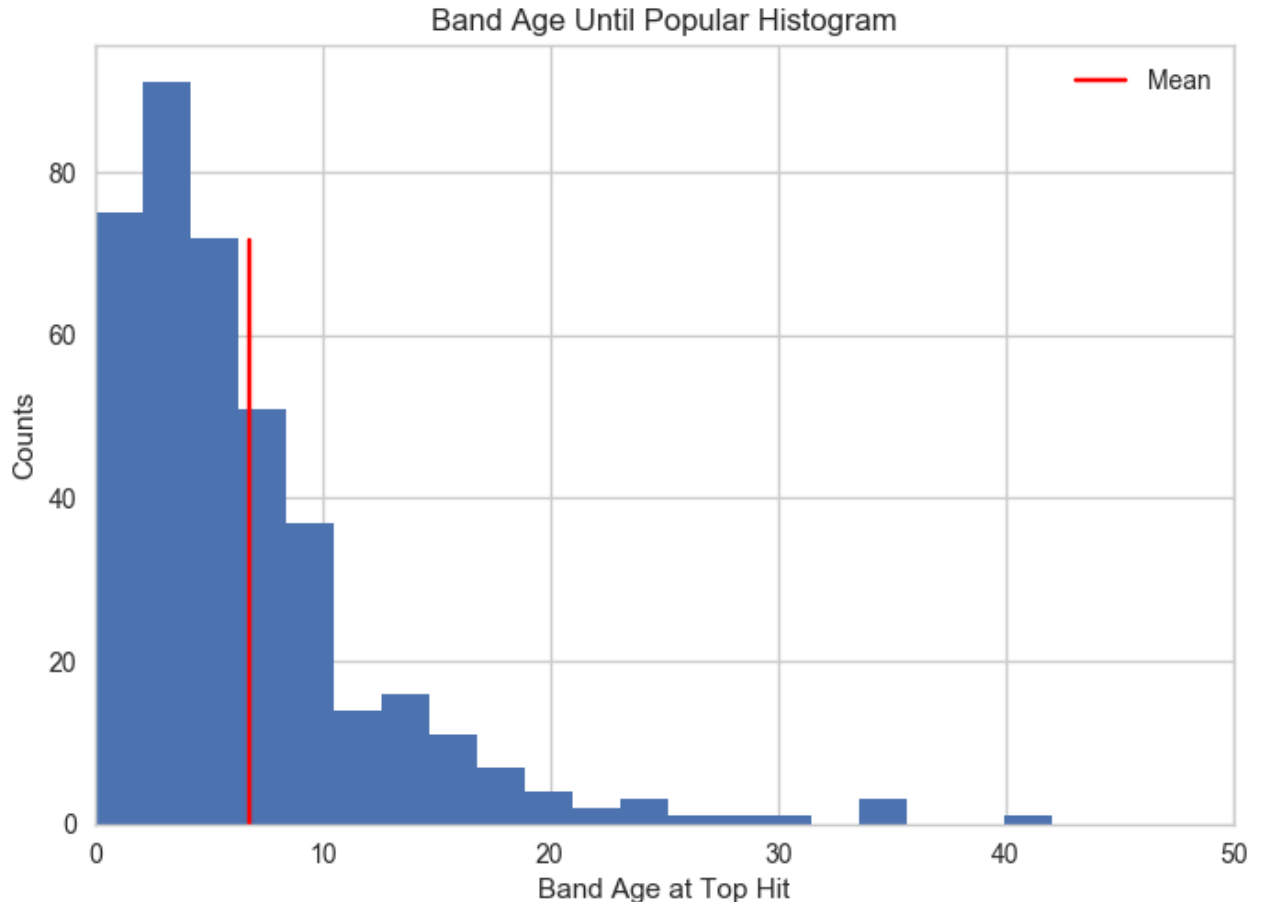
```

In [51]: # function to parse starting year of bands
def yearFinder(string):
    index = string.find('19')
    # if pre 2000s
    if index != -1:
        return int(string[index:(index+4)])
    else:
        index = string.find('20')
        return int(string[index:(index+4)])

#clear NaN's and calculate age of artist at top hit
bandAge = topHits[['year','ya']]
bandAge = bandAge[bandAge['ya'] != False].dropna(axis=0)
bandAge['ya'] = bandAge['ya'].map(yearFinder)
bandAge['age'] = bandAge['year'].astype(int) - bandAge['ya']

#Plot graph
with sns.axes_style("whitegrid"):
    plt.hist(bandAge['age'], bins=20)
    plt.xlim(0, 50)
    plt.axvline(bandAge['age'].mean(), 0, .75, color='r', label='Mean')
    plt.xlabel("Band Age at Top Hit")
    plt.ylabel("Counts")
    plt.title("Band Age Until Popular Histogram")
    plt.legend()
    plt.show()

```



Contrary to what we found with the artists, bands tend to become popular (or at least have a popular song) at a much younger age. In fact, on average, bands become popular in about 7 years and are much less likely to become popular as they get older. So, in conclusion, bands become popular very quickly after creation.