CUNY MSDS DATA608 : Assignment 2

Ramnivas Singh

In [1]:

```
import datashader as ds
import datashader.transfer_functions as tf
import datashader.glyphs
from datashader import reductions
from datashader.core import bypixel
from datashader.utils import lnglat_to_meters as webm, export_image
from datashader.colors import colormap_select, Greys9, viridis, inferno
import copy


from pyproj import Proj, transform
import numpy as np
import pandas as pd
import urllib
import json
import datetime
import colorlover as cl

import plotly.offline as py
import plotly.graph_objs as go
import warnings
warnings.filterwarnings('ignore')
import plotly.offline as py
import plotly.graph_objs as go
from plotly import tools
from plotly.subplots import make_subplots

from shapely.geometry import Point, Polygon, shape

from functools import partial

from IPython.display import GeoJSON

py.init_notebook_mode()
```

For module 2 we'll be looking at techniques for dealing with big data. In particular binning strategies and the datashader library (which possibly proves we'll never need to bin large data for visualization ever again.)

To demonstrate these concepts we'll be looking at the PLUTO dataset put out by New York City's department of city planning. PLUTO contains data about every tax lot in New York City.

PLUTO data can be downloaded from here. Unzip them to the same directory as this notebook, and you should be able to read them in using this (or very similar) code. Also take note of the data dictionary, it'll come in handy for this assignment.

In [2]:
```
# Code to read in v17, column names have been updated (without upper case letters) for v18

# bk = pd.read_csv('PLUTO17v1.1/BK2017V11.csv')
# bx = pd.read_csv('PLUTO17v1.1/BX2017V11.csv')
# mn = pd.read_csv('PLUTO17v1.1/MN2017V11.csv')
# qn = pd.read_csv('PLUTO17v1.1/QN2017V11.csv')
# si = pd.read_csv('PLUTO17v1.1/SI2017V11.csv')

# ny = pd.concat([bk, bx, mn, qn, si], ignore_index=True)

ny = pd.read_csv('pluto_21v4.csv',low_memory=False)
print("Shape of Full Dataset : ",ny.shape)
print(ny.columns)
print(ny.head())

# Getting rid of some outliers
ny = ny[(ny['yearbuilt'] > 1850) & (ny['yearbuilt'] < 2020) & (ny['numfloors'] != 0)]
print("Dataset after outliers are removed : ",len(ny))
```

```
Shape of Full Dataset :  (859032, 92)
Index(['borough', 'block', 'lot', 'cd', 'bct2020', 'bctcb2020', 'ct2010',
       'cb2010', 'schooldist', 'council', 'zipcode', 'firecomp', 'policeprct',
       'healthcenterdistrict', 'healtharea', 'sanitboro', 'sanitdistrict',
       'sanitsub', 'address', 'zonedist1', 'zonedist2', 'zonedist3',
       'zonedist4', 'overlay1', 'overlay2', 'spdist1', 'spdist2', 'spdist3',
       'ltdheight', 'splitzone', 'bldgclass', 'landuse', 'easements',
       'ownertype', 'ownername', 'lotarea', 'bldgarea', 'comarea', 'resarea',
       'officearea', 'retailarea', 'garagearea', 'strgearea', 'factryarea',
       'otherarea', 'areasource', 'numbldgs', 'numfloors', 'unitsres',
       'unitstotal', 'lotfront', 'lotdepth', 'bldgfront', 'bldgdepth', 'ext',
       'proxcode', 'irrlotcode', 'lottype', 'bsmtcode', 'assessland',
       'assesstot', 'exempttot', 'yearbuilt', 'yearalter1', 'yearalter2',
       'histdist', 'landmark', 'builtfar', 'residfar', 'commfar', 'facilfar',
```

```
       'borocode', 'bbl', 'condono', 'tract2010', 'xcoord', 'ycoord',
       'zonemap', 'zmcode', 'sanborn', 'taxmap', 'edesignum', 'appbbl',
       'appdate', 'plutomapid', 'firm07_flag', 'pfirm15_flag', 'version',
       'dcpedited', 'latitude', 'longitude', 'notes'],
      dtype='object')
  borough  block   lot     cd    bct2020     bctcb2020  ct2010  cb2010  \
0      MN    563  7502  102.0  1006100.0  1.006100e+10    61.0  2000.0
1      BK   7133    65  315.0  3039200.0  3.039200e+10   392.0  1003.0
2      BK   7153    53  315.0  3038800.0  3.038800e+10   388.0  2002.0
3      QN   6865   402  408.0  4045000.0  4.045000e+10   450.0  1002.0
4      QN   8658    14  413.0  4161700.0  4.161700e+10  1617.0  1013.0

   schooldist  council  ...        appbbl     appdate  plutomapid  \
0         2.0      2.0  ...  1.005631e+09  08/25/1988           1
1        21.0     47.0  ...           NaN         NaN           1
2        21.0     47.0  ...           NaN         NaN           1
3        28.0     24.0  ...           NaN         NaN           1
4        26.0     23.0  ...           NaN         NaN           1

   firm07_flag  pfirm15_flag  version  dcpedited   latitude  longitude notes
0          NaN           NaN     21v4        NaN  40.733326 -73.991972   NaN
1          NaN           NaN     21v4        NaN  40.597676 -73.964833   NaN
2          NaN           NaN     21v4        NaN  40.593135 -73.969560   NaN
3          NaN           NaN     21v4        NaN  40.715699 -73.803864   NaN
4          NaN           NaN     21v4        NaN  40.725978 -73.723547   NaN

[5 rows x 92 columns]
Dataset after outliers are removed :  810476
```

I'll also do some prep for the geographic component of this data, which we'll be relying on for datashader.

You're not required to know how I'm retrieving the lattitude and longitude here, but for those interested: this dataset uses a flat x-y projection (assuming for a small enough area that the world is flat for easier calculations), and this needs to be projected back to traditional lattitude and longitude.

In [3]:

```
wgs84 = Proj("+proj=longlat +ellps=GRS80 +datum=NAD83 +no_defs")
nyli = Proj("+proj=lcc +lat_1=40.66666666666666 +lat_2=41.03333333333333 +lat_0=40.16666666666666 +lon_0=-74 +x_0=300000
ny['xcoord'] = 0.3048*ny['xcoord']
ny['ycoord'] = 0.3048*ny['ycoord']
ny['lon'], ny['lat'] = transform(nyli, wgs84, ny['xcoord'].values, ny['ycoord'].values)

ny = ny[(ny['lon'] < -60) & (ny['lon'] > -100) & (ny['lat'] < 60) & (ny['lat'] > 20)]
print(ny.shape)
print(ny.head())
```

```
#Defining some helper functions for DataShader
background = "black"
export = partial(export_image, background = background, export_path="export")
cm = partial(colormap_select, reverse=(background!="black"))
```

```
(810024, 94)
  borough  block   lot     cd    bct2020     bctcb2020  ct2010  cb2010  \
0      MN    563  7502  102.0  1006100.0  1.006100e+10    61.0  2000.0
1      BK   7133    65  315.0  3039200.0  3.039200e+10   392.0  1003.0
2      BK   7153    53  315.0  3038800.0  3.038800e+10   388.0  2002.0
3      QN   6865   402  408.0  4045000.0  4.045000e+10   450.0  1002.0
4      QN   8658    14  413.0  4161700.0  4.161700e+10  1617.0  1013.0

   schooldist  council  ...  plutomapid firm07_flag  pfirm15_flag  version  \
0         2.0      2.0  ...           1         NaN           NaN     21v4
1        21.0     47.0  ...           1         NaN           NaN     21v4
2        21.0     47.0  ...           1         NaN           NaN     21v4
3        28.0     24.0  ...           1         NaN           NaN     21v4
4        26.0     23.0  ...           1         NaN           NaN     21v4

   dcpedited   latitude  longitude notes        lon        lat
0        NaN  40.733326 -73.991972   NaN -76.451368  40.313130
1        NaN  40.597676 -73.964833   NaN -76.441624  40.271977
2        NaN  40.593135 -73.969560   NaN -76.443006  40.270563
3        NaN  40.715699 -73.803864   NaN -76.394208  40.309017
4        NaN  40.725978 -73.723547   NaN -76.370010  40.312706

[5 rows x 94 columns]
```

Part 1: Binning and Aggregation

Binning is a common strategy for visualizing large datasets. Binning is inherent to a few types of visualizations, such as histograms and 2D histograms (also check out their close relatives: 2D density plots and the more general form: heatmaps.

While these visualization types explicitly include binning, any type of visualization used with aggregated data can be looked at in the same way. For example, lets say we wanted to look at building construction over time. This would be best viewed as a line graph, but we can still think of our results as being binned by year:

In [4]:

```
trace = go.Scatter(
    # I'm choosing BBL here because I know it's a unique key.
```

```
    x = ny.groupby('yearbuilt').count()['bbl'].index,
    y = ny.groupby('yearbuilt').count()['bbl']
)

layout = go.Layout(
    xaxis = dict(title = 'Year Built'),
    yaxis = dict(title = 'Number of Lots Built')
)

fig = go.FigureWidget(data = [trace], layout = layout)

fig
```

Something looks off... You're going to have to deal with this imperfect data to answer this first question.

But first: some notes on pandas. Pandas dataframes are a different beast than R dataframes, here are some tips to help you get up to speed:

Hello all, here are some pandas tips to help you guys through this homework:

Indexing and Selecting: .loc and .iloc are the analogs for base R subsetting, or filter() in dplyr

Group By: This is the pandas analog to group_by() and the appended function the analog to summarize(). Try out a few examples of this, and display the results in Jupyter. Take note of what's happening to the indexes, you'll notice that they'll become hierarchical. I personally find this more of a burden than a help, and this sort of hierarchical indexing leads to a fundamentally different experience compared to R dataframes. Once you perform an aggregation, try running the resulting hierarchical datafrome through a reset_index().

Reset_index: I personally find the hierarchical indexes more of a burden than a help, and this sort of hierarchical indexing leads to a fundamentally different experience compared to R dataframes. reset_index() is a way of restoring a dataframe to a flatter index style. Grouping is where you'll notice it the most, but it's also useful when you filter data, and in a few other split-apply-combine workflows. With pandas indexes are more meaningful, so use this if you start getting unexpected results.

Indexes are more important in Pandas than in R. If you delve deeper into the using python for data science, you'll begin to see the benefits in many places (despite the personal gripes I highlighted above.) One place these indexes come in handy is with time series data. The pandas docs have a huge section on datetime indexing. In particular, check out resample, which provides time series specific aggregation.

Merging, joining, and concatenation: There's some overlap between these different types of merges, so use this as your guide. Concat is a single function that replaces cbind and rbind in R, and the results are driven by the indexes. Read through these examples to get a feel on how these are performed, but you will have to manage your indexes when you're using these functions. Merges are fairly similar to merges in R, similarly mapping to SQL joins.

Apply: This is explained in the "group by" section linked above. These are your analogs to the plyr library in R. Take note of the lambda syntax used here, these are anonymous functions in python. Rather than predefining a custom function, you can just define it inline using lambda.

Browse through the other sections for some other specifics, in particular reshaping and categorical data (pandas' answer to factors.) Pandas can take a while to get used to, but it is a pretty strong framework that makes more advanced functions easier once you get used to it. Rolling functions for example follow logically from the apply workflow (and led to the best google results ever when I first tried to find this out and googled "pandas rolling")

Google Wes Mckinney's book "Python for Data Analysis," which is a cookbook style intro to pandas. It's an O'Reilly book that should be pretty available out there.

Question

After a few building collapses, the City of New York is going to begin investigating older buildings for safety. The city is particularly worried about buildings that were unusually tall when they were built, since best-practices for safety hadn't yet been determined. Create a graph that shows how many buildings of a certain number of floors were built in each year (note: you may want to use a log scale for the number of buildings). Find a strategy to bin buildings (It should be clear 20-29-story buildings, 30-39-story buildings, and 40-49-story buildings were first built in large numbers, but does it make sense to continue in this way as you get taller?)

In [5]:
```
# Start your answer here, inserting more cells as you go along
df = ny[['numfloors','yearbuilt']]
floor_bins = [0, 1, 2, 3, 4, 5, 10, 20, 50, df.numfloors.max()]
df = (df
      .assign(totalbuilt=1,
              floorbin=pd.cut(df['numfloors'], bins=floor_bins),
              decade=(df['yearbuilt'] // 10 * 10).astype(int))
      .sort_values(by=['decade'], ascending=True)
      .drop(columns=['yearbuilt', 'numfloors'])
      .groupby(['decade','floorbin'])
      .count()
```

```
      .dropna()
      .reset_index()
     )
df.head(100)
```

Out[5]:

	decade	floorbin	totalbuilt
0	1850	(0.0, 1.0]	4
1	1850	(1.0, 2.0]	65
2	1850	(2.0, 3.0]	843
3	1850	(3.0, 4.0]	390
4	1850	(4.0, 5.0]	199
...	...	...	...
95	1950	(5.0, 10.0]	1098
96	1950	(10.0, 20.0]	296
97	1950	(20.0, 50.0]	65
98	1950	(50.0, 104.0]	2
99	1960	(0.0, 1.0]	13656

100 rows × 3 columns

In [6]:

```
ny_sub = ny[['bbl','yearbuilt','numfloors']]
rounded_floors = pd.DataFrame(ny_sub['numfloors'].apply(np.ceil))

ny_sub = ny_sub.join(rounded_floors,lsuffix="_orig")
ny_sub['floorbins'] = pd.cut(x=ny_sub['numfloors'],\
                             bins=[0,10,20,30,40,50,60,70,80,90,100,110],\
                             labels=["0-10","11-20","21-30","31-40", "41-50", "51-60","61-70",\
                                     "71-80","81-90","91-100","101-110"])

ny_sub['yearbins'] = pd.cut(x=ny_sub['yearbuilt'],\
                            bins = [1850, 1860, 1870, 1880, 1890, 1900, 1910, 1920, 1930, 1940, 1950,\
                                    1960, 1970, 1980, 1990, 2000, 2010, 2020],\
                            labels=["1850-1860", "1861-1870", "1871-1880", "1881-1890", "1891-1900",\
                                    "1901-1910", "1911-1920", "1921-1930", "1931-1940", "1941-1950",\
```

```
                                    "1951-1960", "1961-1970", "1971-1980",\
                                    "1981-1990","1991-2000", "2001-2010", "2011-2020"])


ny_sub2 = ny_sub[['yearbins','floorbins']]
ny_sub3 = ny_sub2.groupby(['yearbins','floorbins']).size()
ny_final = ny_sub3.unstack()
ny_final
```

Out[6]:

floorbins	0-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80	81-90	91-100	101-110
yearbins											
1850-1860	1778	2	0	0	0	0	0	0	0	0	0
1861-1870	1584	2	0	0	0	0	0	0	0	0	0
1871-1880	3022	2	0	0	0	0	0	0	0	0	0
1881-1890	5575	10	1	1	1	0	0	0	0	0	0
1891-1900	31326	83	8	0	0	0	0	0	0	0	0
1901-1910	76598	384	14	2	1	1	0	1	0	0	0
1911-1920	107772	488	36	3	2	0	1	0	0	0	0
1921-1930	162948	945	126	33	12	4	2	0	1	0	0
1931-1940	99367	156	18	11	7	2	2	0	0	0	1
1941-1950	74908	100	12	2	1	1	0	0	0	0	0
1951-1960	69213	329	56	12	5	1	0	0	0	0	0
1961-1970	41969	522	176	60	28	4	0	0	0	0	0
1971-1980	20531	148	71	74	22	7	0	0	0	0	0
1981-1990	24907	167	90	78	45	9	2	1	0	0	0
1991-2000	29499	70	31	35	13	4	2	0	0	0	0
2001-2010	39510	360	88	56	28	23	1	2	0	0	1
2011-2020	13698	435	136	66	40	20	16	4	1	1	0

By looking the data, the columns name "YearBuilt" is accurate for the decade, therefore, year of construction will be binned by decade.

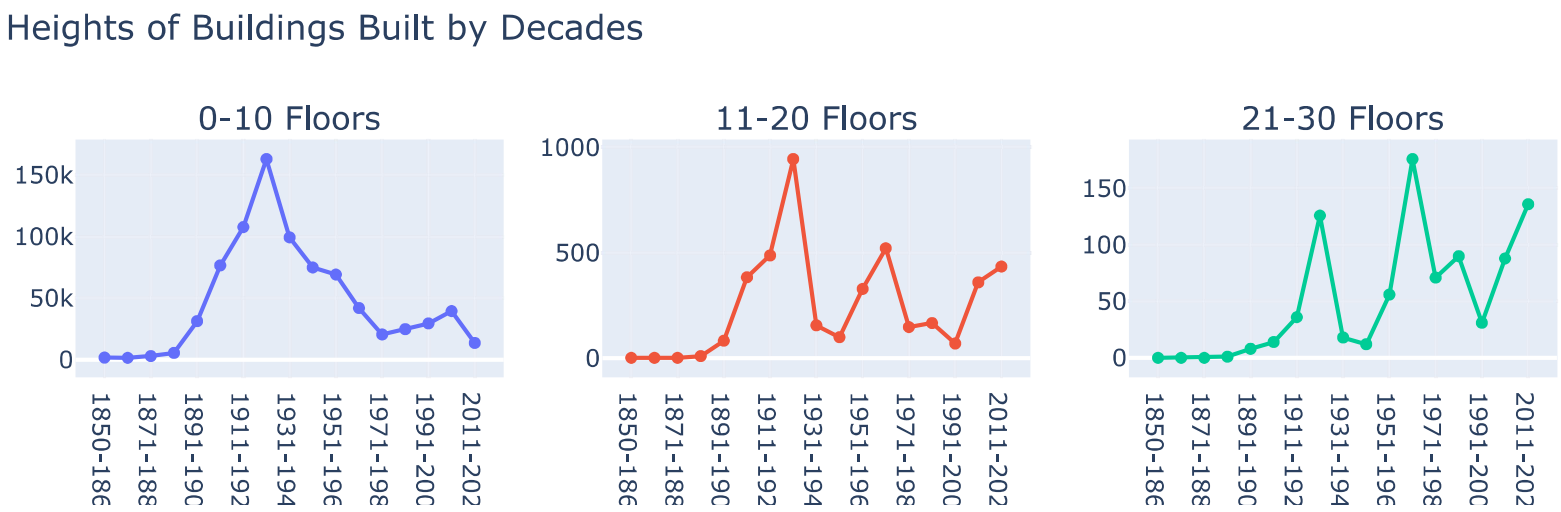
Construction year ranges from 1851 to 2017; however, values from the 19th century are unrealible. Anything built prior to 1910, is included in the bin for 1910.

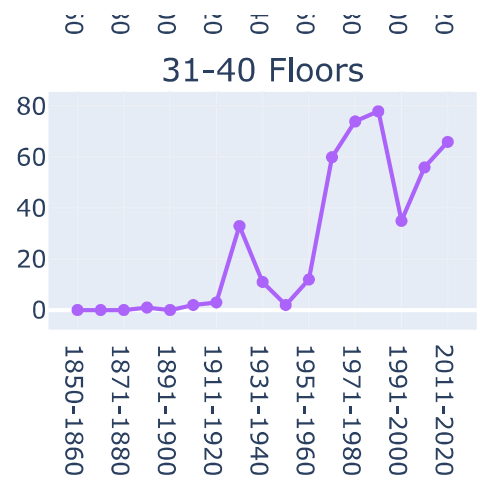
As such I have selected the following bins: individual floors up to 10, aggregated by 5 floors from 10 to 50, aggregated by 10 floors from 50 to 90 and then a single bin for anything over 90. Number of floors, NumFloors, ranges from 0 to 119. There is significantly more short buildings than tall ones.

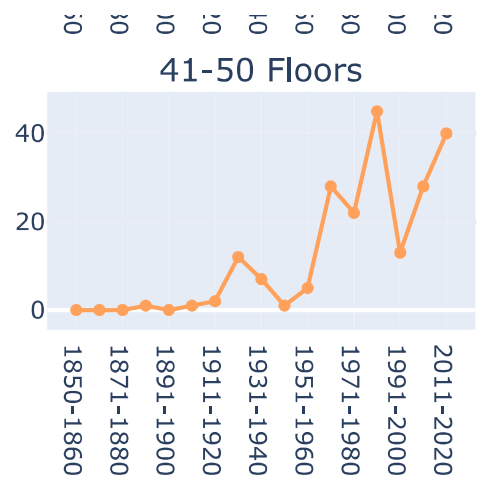
In [7]:

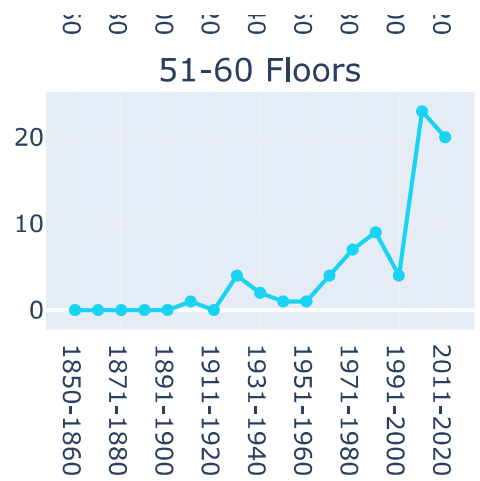
```
fig = make_subplots(rows=6, cols=3,
      subplot_titles=("0-10 Floors","11-20 Floors","21-30 Floors","31-40 Floors", "41-50 Floors",
                      "51-60 Floors","61-70 Floors","71-80 Floors","81-90 Floors",
                      "91-100 Floors","101-110 Floors"))

fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['0-10'],
), row=1, col=1)

fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['11-20'],
), row=1, col=2)

fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['21-30'],
), row=1, col=3)
###
fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['31-40'],
), row=2, col=1)

fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['41-50'],
), row=2, col=2)

fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['51-60'],
), row=2, col=3)
###
fig.append_trace(go.Scatter(
```

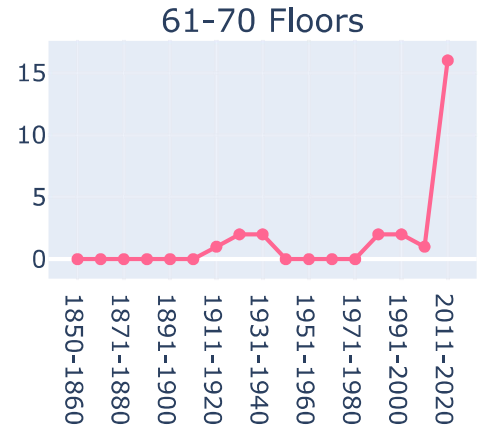
```
    x=ny_final.index,
    y=ny_final['61-70'],
), row=3, col=1)

fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['71-80'],
), row=3, col=2)

fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['81-90'],
), row=3, col=3)
###
fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['91-100'],
), row=4, col=1)

fig.append_trace(go.Scatter(
    x=ny_final.index,
    y=ny_final['101-110'],
), row=4, col=2)

fig.update_layout(height=1400, width=900, title_text="Heights of Buildings Built by Decades",
                  showlegend=False)
fig.show()
```

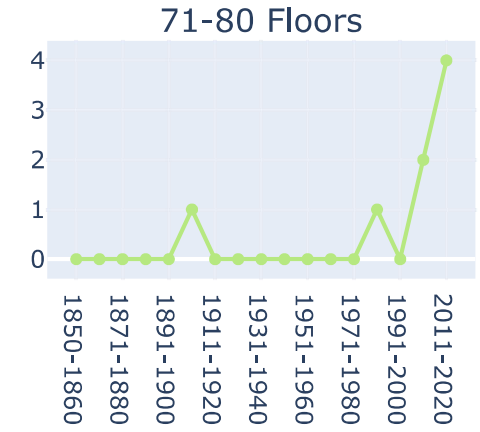


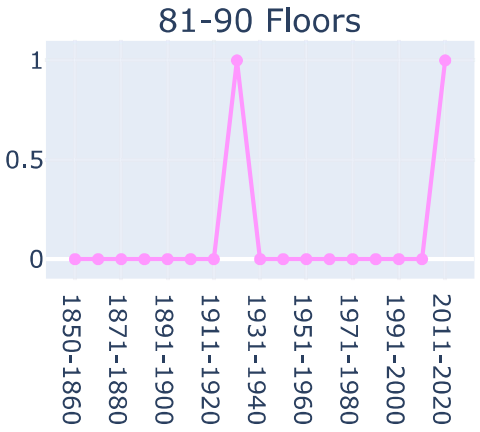

31-40 Floors
80
60
40
20
0
1850-1860
1871-1880
1891-1900
1911-1920
1931-1940
1951-1960
1971-1980
1991-2000
2011-2020

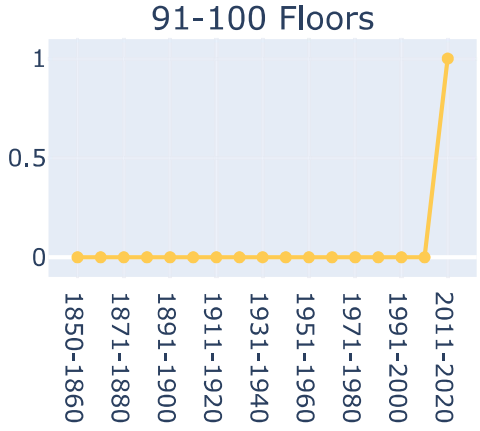

41-50 Floors
40
20
0
1850-1860
1871-1880
1891-1900
1911-1920
1931-1940
1951-1960
1971-1980
1991-2000
2011-2020

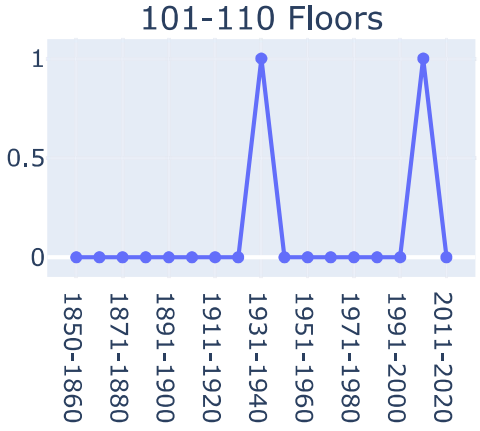

51-60 Floors
20
10
0
1850-1860
1871-1880
1891-1900
1911-1920
1931-1940
1951-1960
1971-1980
1991-2000
2011-2020


61-70 Floors
15
10
5
0
1850-1860
1871-1880
1891-1900
1911-1920
1931-1940
1951-1960
1971-1980
1991-2000
2011-2020


71-80 Floors
4
3
2
1
0
1850-1860
1871-1880
1891-1900
1911-1920
1931-1940
1951-1960
1971-1980
1991-2000
2011-2020


81-90 Floors
1
0.5
0
1850-1860
1871-1880
1891-1900
1911-1920
1931-1940
1951-1960
1971-1980
1991-2000
2011-2020


91-100 Floors
1
0.5
0
1850-1860
1871-1880
1891-1900
1911-1920
1931-1940
1951-1960
1971-1980
1991-2000
2011-2020


101-110 Floors
1
0.5
0
1850-1860
1871-1880
1891-1900
1911-1920
1931-1940
1951-1960
1971-1980
1991-2000
2011-2020

Part 2: Datashader

Datashader is a library from Anaconda that does away with the need for binning data. It takes in all of your datapoints, and based on the canvas and range returns a pixel-by-pixel calculations to come up with the best representation of the data. In short, this completely eliminates the need for binning your data.

As an example, lets continue with our question above and look at a 2D histogram of YearBuilt vs NumFloors:

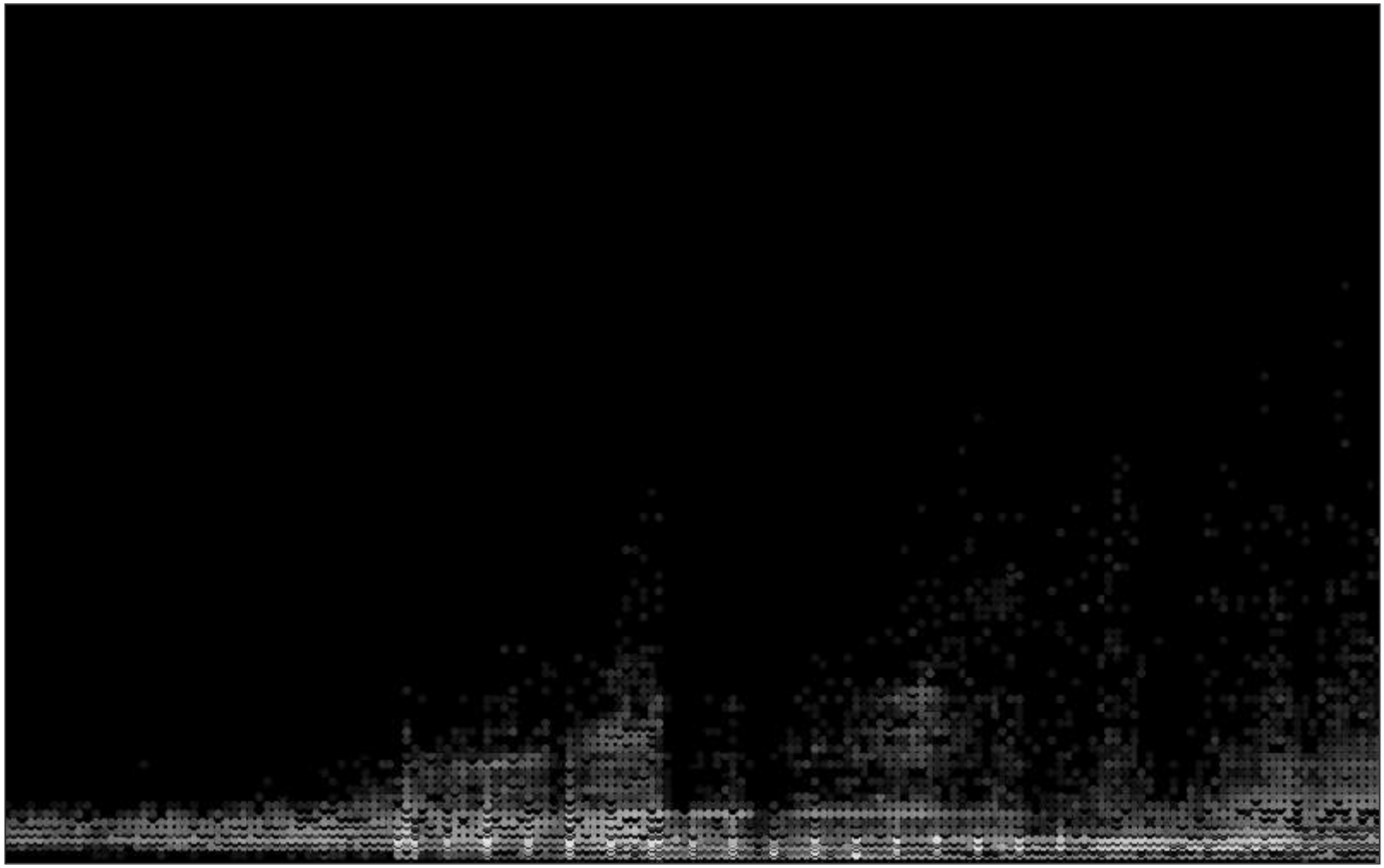
In [8]:
```
fig = go.FigureWidget(
    data = [
        go.Histogram2d(x=ny['yearbuilt'], y=ny['numfloors'], autobiny=False, ybins={'size': 1}, colorscale='Greens')
    ]
)

fig
```

This shows us the distribution, but it's subject to some biases discussed in the Anaconda notebook Plotting Perils.

Here is what the same plot would look like in datashader:

In [9]:
```
#Defining some helper functions for DataShader
background = "black"
export = partial(export_image, background = background, export_path="export")
cm = partial(colormap_select, reverse=(background!="black"))

cvs = ds.Canvas(800, 500, x_range = (ny['yearbuilt'].min(), ny['yearbuilt'].max()),
                                  y_range = (ny['numfloors'].min(), ny['numfloors'].max()))
agg = cvs.points(ny, 'yearbuilt', 'numfloors')
view = tf.shade(agg, cmap = cm(Greys9), how='log')
export(tf.spread(view, px=2), 'yearvsnumfloors')
```

Out[9]:



That's technically just a scatterplot, but the points are smartly placed and colored to mimic what one gets in a heatmap. Based on the pixel size, it will either display individual points, or will color the points of denser regions.

Datashader really shines when looking at geographic information. Here are the latitudes and longitudes of our dataset plotted out, giving us a map of the city colored by density of structures:

Interestingly, since we're looking at structures, the large buildings of Manhattan show up as less dense on the map. The densest areas measured by number of lots would be single or multi family townhomes.

Unfortunately, Datashader doesn't have the best documentation. Browse through the examples from their github repo. I would focus on the visualization pipeline and the US Census Example for the question below. Feel free to use my samples as templates as well when you

work on this problem.

Question

You work for a real estate developer and are researching underbuilt areas of the city. After looking in the Pluto data dictionary, you've discovered that all tax assessments consist of two parts: The assessment of the land and assessment of the structure. You reason that there should be a correlation between these two values: more valuable land will have more valuable structures on them (more valuable in this case refers not just to a mansion vs a bungalow, but an apartment tower vs a single family home). Deviations from the norm could represent underbuilt or overbuilt areas of the city. You also recently read a really cool blog post about bivariate choropleth maps, and think the technique could be used for this problem.

Datashader is really cool, but it's not that great at labeling your visualization. Don't worry about providing a legend, but provide a quick explanation as to which areas of the city are overbuilt, which areas are underbuilt, and which areas are built in a way that's properly correlated with their land value.

In [11]:
```
ny_sub = ny[["xcoord",'ycoord', "assessland", "assesstot"]]
ny_sub["assessstructure"] = ny['assesstot'] - ny['assessland']

ny_sub[["assessland","assessstructure"]].describe()

ny_sub["landclass"] = pd.qcut(x=ny_sub["assessland"], q=3, labels=["L1","L2","L3"])
ny_sub["structureclass"] = pd.qcut(x=ny_sub["assessstructure"],q=3, labels=["S1","S2","S3"])

ny_sub["builtclass"] = ny_sub["landclass"].astype(str) + ny_sub["structureclass"].astype(str)
ny_sub["builtclass"] = ny_sub["builtclass"].astype('category')

print(ny_sub)
```

```
            xcoord      ycoord  assessland  assesstot  assessstructure  \
0       300677.5800  62925.0456    292501.0  6494851.0        6202350.0
1       302976.0768  47862.1344     18000.0   178800.0         160800.0
2       302576.1792  47357.6904      6360.0    59580.0          53220.0
3       316571.9856  60986.2128     15960.0    39060.0          23100.0
4       323354.7000  62145.9768     26280.0    44220.0          17940.0
...             ...         ...         ...        ...              ...
858995  291586.6152  44198.1336      5760.0    12840.0           7080.0
859000  287836.9656  44688.2520     17760.0    44340.0          26580.0
859001  289147.3008  44979.9456      9420.0    33420.0          24000.0
859017  286522.3632  40558.5168     12180.0    58740.0          46560.0
859020  286558.3296  40905.3792     15060.0    58560.0          43500.0
```

```
       landclass structureclass builtclass
0             L3              S3       L3S3
1             L2              S3       L2S3
2             L1              S2       L1S2
3             L2              S1       L2S1
4             L3              S1       L3S1
...          ...             ...        ...
858995        L1              S1       L1S1
859000        L2              S1       L2S1
859001        L1              S1       L1S1
859017        L2              S2       L2S2
859020        L2              S2       L2S2

[810024 rows x 8 columns]
```

This second map confirmed my visual assumptions earlier. Mahanattan is overbuilt (red) along wth some areas on the west side of Brooklyn and Queens. The outskirts of Staten Island, eastern areas of Queens and Broooklyn are considered underbuilt (plum). The assessment value for areas of Queens, Brooklyn and Bronx close to Manhattan (aqua) are considered to be correlated with the value assigned to land.

In [ ]:

```
```

In [12]:

```
ny['AssessStruct'] = ny['assesstot'] - ny['assessland']

al33 = np.percentile(ny['assessland'].values, 33)
al66 = np.percentile(ny['assessland'].values, 66)
as33 = np.percentile(ny['AssessStruct'].values, 33)
as66 = np.percentile(ny['AssessStruct'].values, 66)

ny['AssessLand3tile'] = ny['assessland'].apply(lambda x: 1 if x < al33 else 2 if x < al66 else 3)
ny['AssessStruct3tile'] = ny['AssessStruct'].apply(lambda x: 1 if x < as33 else 2 if x < as66 else 3)
ny['AssessAll3tile'] = ny['AssessLand3tile'].apply(str) + ny['AssessStruct3tile'].apply(str) + 'c'
ny['AssessAll3tile'] = pd.Categorical(ny['AssessAll3tile'])

color_key = {'11c': '#e8e8e8', '12c': '#b0d5df','13c': '#64acbe','21c': '#e4acac', '22c': '#ad9ea5', '23c': '#627f8c',
    '31c': '#c85a5a','32c': '#985356', '33c': '#574249'}
```

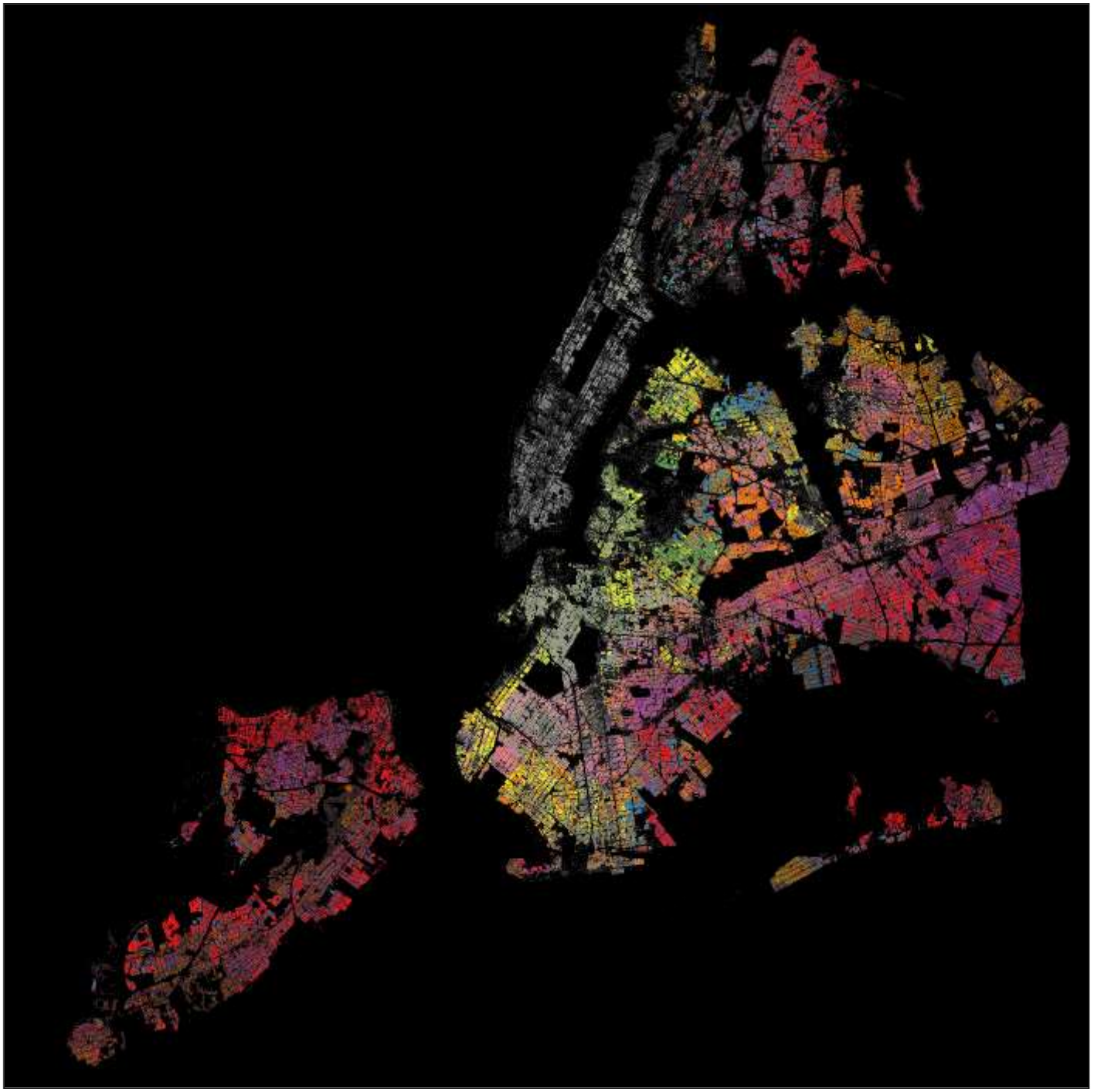
Greener is Assessed Land, Redder is Assessed Total, Maroon is both

In [13]:

```
nyc   = (( -74.29,  -73.69), (40.49, 40.92))
cvs = ds.Canvas(700, 700, *nyc)
```

```
agg = cvs.points(ny, 'longitude', 'latitude',ds.count_cat('AssessAll3tile'))
export(tf.shade(agg, cmap=['lightgreen', 'darkgreen']), 'tester')
```

Out[13]:



Plot for assessment land

In [14]:

```
# Let's create a dataframe for New York city taxes
ny_taxes = ny[['bbl', 'longitude', 'latitude', 'assessland','assesstot']]
tf.shade(cvs.points(ny_taxes,'longitude','latitude',agg=reductions.mean('assessland')))
```

Out[14]:

Conclusion

The brighter of the color means higher value of the land and structure. Apparently, Manhattan has the highest value with its bright yellow color. We also notice some water front areas also have a high land and structure values. These two values are correlated with each other. Those dark purple areas is undervalue with low land and structure value, such as Staten Island and the South side of Queens.

A few entries assessed over one billion (unrealistic even for NYC) or missing assessment values are removed. Then each group - land and structure - is divided into 3 categories based on the spread - high, medium and low. Because the spread is too wide, assessment values are log transformed.

In [ ]: