Assignment2

November 22, 2020

```
# setting up platform
library(reticulate)
py_run_string("import os as os")
py_run_string("os.environ['QT_QPA_PLATFORM_PLUGIN_PATH'] = 'C:/Users/NK/AppData/Local/r-miniconda/envs/
```

Using descriptive statistics to find out website performance.

Import nessecary libraries

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

Cleaning data

Before start let's first take a quick look at the some basic description from data.

Load the data into jupyter and examine the first few rows of the table

```
tripvn = pd.read_csv('tripvn.csv', index_col = 0)
tripvn.head()
```

```
##                    Test             Time  ...  % Availability  # Runs
## 0  [204709] - trip.com  11/01/2017 0:00  ...          91.667      12
## 1  [204709] - trip.com  11/01/2017 2:00  ...         100.000      12
## 2  [204709] - trip.com  11/01/2017 4:00  ...         100.000      11
## 3  [204709] - trip.com  11/01/2017 6:00  ...         100.000      13
## 4  [204709] - trip.com  11/01/2017 8:00  ...         100.000      11
##
## [5 rows x 12 columns]
```

Check number of rows and non-null object in each columns

```
tripvn.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## Int64Index: 360 entries, 0 to 359
## Data columns (total 12 columns):
##  #   Column                      Non-Null Count  Dtype  
## ---  ------                      --------------  -----  
##  0   Test                        360 non-null    object 
##  1   Time                        360 non-null    object 
##  2   Mdn DNS (ms)                360 non-null    float64
##  3   Mdn SSL (ms)                0 non-null      float64
##  4   Avg Time To First Byte (ms)  360 non-null    float64
##  5   Mdn Webpage Response (ms)    360 non-null    float64
##  6   Mdn Render Start (ms)        360 non-null    float64
##  7   Avg Image Bytes              360 non-null    float64
##  8   Avg Script Bytes             360 non-null    float64
##  9   Avg Css Bytes                360 non-null    float64
##  10  % Availability               360 non-null    float64
##  11  # Runs                       360 non-null    int64  
## dtypes: float64(9), int64(1), object(2)
## memory usage: 36.6+ KB
```

We can see that in the Mdn SSL (ms) column all the rows are NaN value, and so we have two options here:

- First option is dropping the hold column.
- Second one is replacing the NaN with 0 value, and it seems like there is no use from Mdn SSL because anyway SSL (ms) is an old obsolete standard security, TLS is now a far more secure version of SSL, but we don't have the numbers related to it here, so delete the Mdn SSl column is secure here.

```
tripvn.drop(axis = 1, inplace = True, columns = ['Mdn SSL (ms)'])
tripvn.head()
```

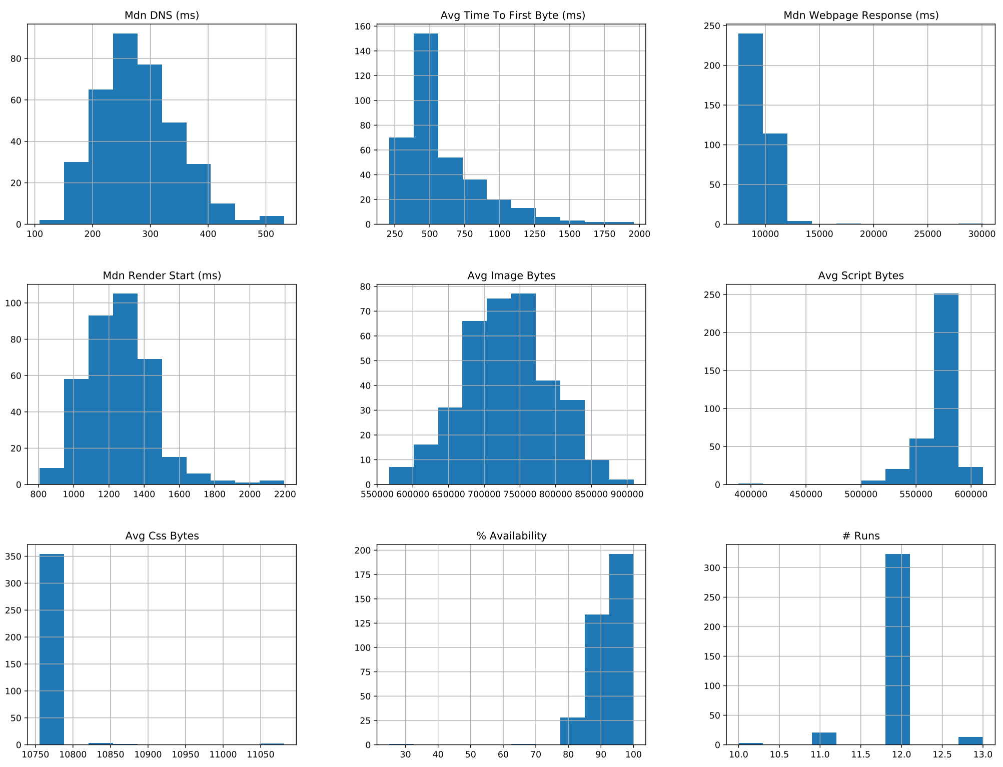
```
##                      Test             Time  ...  % Availability  # Runs
## 0  [204709] - trip.com  11/01/2017 0:00  ...          91.667      12
## 1  [204709] - trip.com  11/01/2017 2:00  ...         100.000      12
## 2  [204709] - trip.com  11/01/2017 4:00  ...         100.000      11
## 3  [204709] - trip.com  11/01/2017 6:00  ...         100.000      13
## 4  [204709] - trip.com  11/01/2017 8:00  ...         100.000      11
## 
## [5 rows x 11 columns]
```

Now, We could take a first look at the given value range through the histogram

```
tripvn.hist(figsize = (20,15))
```

```
## array([[<AxesSubplot:title={'center':'Mdn DNS (ms)'}>,
##         <AxesSubplot:title={'center':'Avg Time To First Byte (ms)'}>,
##         <AxesSubplot:title={'center':'Mdn Webpage Response (ms)'}>],
##        [<AxesSubplot:title={'center':'Mdn Render Start (ms)'}>,
##         <AxesSubplot:title={'center':'Avg Image Bytes'}>,
##         <AxesSubplot:title={'center':'Avg Script Bytes'}>],
##        [<AxesSubplot:title={'center':'Avg Css Bytes'}>,
##         <AxesSubplot:title={'center':'% Availability'}>,
##         <AxesSubplot:title={'center':'# Runs'}>]], dtype=object)
```

```
plt.show()
```



$Graph 0.1$

Take a look at some standard measurements to get more understanding about the data

```
tripvn.describe()
```

```
##          Mdn DNS (ms)  Avg Time To First Byte (ms)  ...  % Availability     # Runs
## count      360.000000                   360.000000  ...      360.000000 360.000000
## mean       280.076389                   593.206944  ...       95.262775  11.961111
## std         70.513539                   286.766903  ...        6.689858   0.355833
## min        108.500000                   209.750000  ...       25.000000  10.000000
## 25%        232.875000                   401.747500  ...       91.667000  12.000000
## 50%        274.750000                   484.170000  ...      100.000000  12.000000
## 75%        323.000000                   710.352500  ...      100.000000  12.000000
## max      531.500000                  1957.580000  ...      100.000000   13.000000
##
## [8 rows x 9 columns]
```

There are only 2 object in data frame is Test and Time. When we inspect the first 5 rows with repetitive value, so it can be a categorical attribute.

```
tripvn['Test'].value_counts()
```

```
## [204709] - trip.com    360
## Name: Test, dtype: int64
```

```
tripvn.head()
```

```
##                     Test             Time  ...  % Availability  # Runs
## 0  [204709] - trip.com  11/01/2017 0:00  ...          91.667      12
## 1  [204709] - trip.com  11/01/2017 2:00  ...         100.000      12
## 2  [204709] - trip.com  11/01/2017 4:00  ...         100.000      11
## 3  [204709] - trip.com  11/01/2017 6:00  ...         100.000      13
## 4  [204709] - trip.com  11/01/2017 8:00  ...         100.000      11
##
## [5 rows x 11 columns]
```

It's interesting that there is only 1 value in the hold Test column, so it's basically meaning less here to include in our dataset. So, we can just drop it like the Mdn SSL column

```
tripvn.drop(axis = 1, inplace = True, columns = ['Test'])
```

Let's check all the column and value once again to make sure everything alright.

```
tripvn.head()
```

```
##               Time  Mdn DNS (ms)  ...  % Availability  # Runs
## 0  11/01/2017 0:00         287.5  ...          91.667      12
## 1  11/01/2017 2:00         254.5  ...         100.000      12
## 2  11/01/2017 4:00         228.0  ...         100.000      11
## 3  11/01/2017 6:00         180.0  ...         100.000      13
## 4  11/01/2017 8:00         227.0  ...         100.000      11
##
## [5 rows x 10 columns]
```

```
tripvn.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## Int64Index: 360 entries, 0 to 359
## Data columns (total 10 columns):
##  #   Column                      Non-Null Count  Dtype
## ---  ------                      --------------  -----
##  0   Time                        360 non-null    object
##  1   Mdn DNS (ms)                360 non-null    float64
##  2   Avg Time To First Byte (ms)  360 non-null    float64
##  3   Mdn Webpage Response (ms)    360 non-null    float64
##  4   Mdn Render Start (ms)        360 non-null    float64
##  5   Avg Image Bytes              360 non-null    float64
##  6   Avg Script Bytes             360 non-null    float64
##  7   Avg Css Bytes                360 non-null    float64
##  8   % Availability               360 non-null    float64
##  9   # Runs                       360 non-null    int64
## dtypes: float64(8), int64(1), object(1)
## memory usage: 30.9+ KB
```

Q1. Overview of metrics

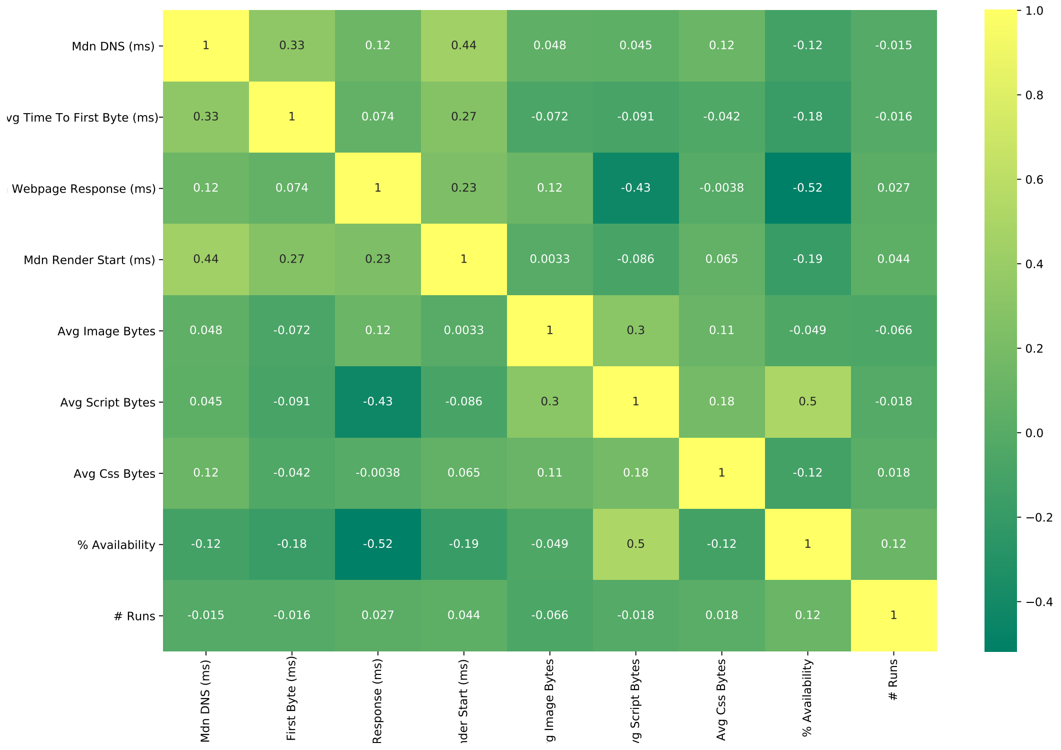
Take a quick look into the correlations between these metrics.

```
corr_metric = tripvn.corr()
corr_metric
```

```
##                              Mdn DNS (ms)  ...    # Runs
## Mdn DNS (ms)                     1.000000  ... -0.014535
## Avg Time To First Byte (ms)      0.330681  ... -0.016074
## Mdn Webpage Response (ms)        0.124822  ...  0.026945
## Mdn Render Start (ms)            0.436882  ...  0.043670
## Avg Image Bytes                  0.048118  ... -0.065512
## Avg Script Bytes                 0.044505  ... -0.017642
## Avg Css Bytes                    0.121324  ...  0.017672
## % Availability                  -0.115584  ...  0.115678
## # Runs                          -0.014535  ...  1.000000
##
## [9 rows x 9 columns]
```

Visualize correlation through graph

```
plt.figure(figsize = (15,10))
sns.heatmap(tripvn.corr(),annot=True,cmap='summer')
plt.show()
```



*Graph*1.1

Let's look at how much each attribute with Mdn DNS (ms)

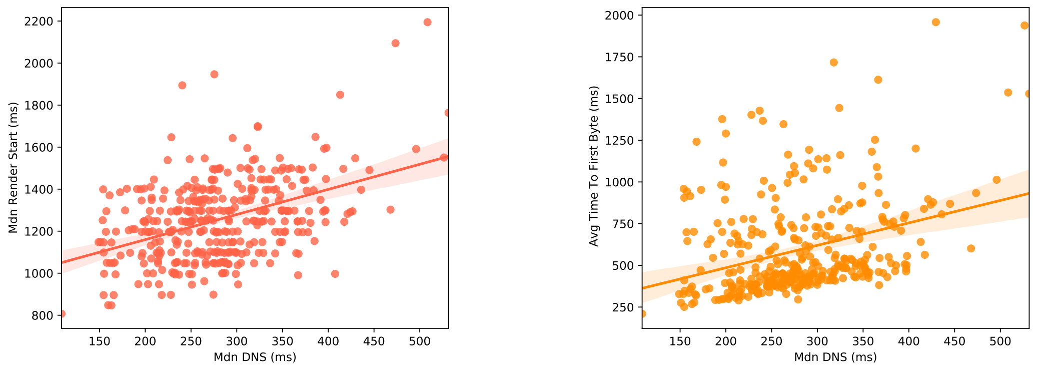
```
corr_metric['Mdn DNS (ms)'].sort_values(ascending = False)
```

```
## Mdn DNS (ms)                    1.000000
## Mdn Render Start (ms)           0.436882
## Avg Time To First Byte (ms)     0.330681
## Mdn Webpage Response (ms)       0.124822
## Avg Css Bytes                   0.121324
## Avg Image Bytes                 0.048118
## Avg Script Bytes                0.044505
## # Runs                         -0.014535
## % Availability                 -0.115584
## Name: Mdn DNS (ms), dtype: float64
```

We can see that 2 columns Mdn Render Start and Avg Time To First Byte have a pretty strong correlation with Mdn DNS in comparision with other columns. So let's take a closer in their look correlation scatterplot

```
fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (15,5))
plt.subplots_adjust(wspace = 0.5)
sns.regplot(data = tripvn, scatter = True, x = 'Mdn DNS (ms)',
            y = 'Mdn Render Start (ms)', ax = axes[0],
            label = 'Mdn Render Start', color = 'tomato')

sns.regplot(data = tripvn,  scatter = True, x = 'Mdn DNS (ms)',
            y = 'Avg Time To First Byte (ms)', ax = axes[1],
            label = 'Avg Time To First Byte', color = 'darkorange')
plt.show()
```



$Graph 1.2 - 1.3$

It's apparent that there is an upward trend in both plots and the point are not too dispersed, we can also notice that when Avg Time To First Byte (ms) below 500 (ms) there are much more concentrated points and the pattern is more visible.

With other attribute, we can think that the faster the connection established between your computer and your server, the more rapid components in website is loaded. But from the graph, we can assure to ignore this idea when the correlation is just to small.

We do it similarly with Avg Css Bytes columns

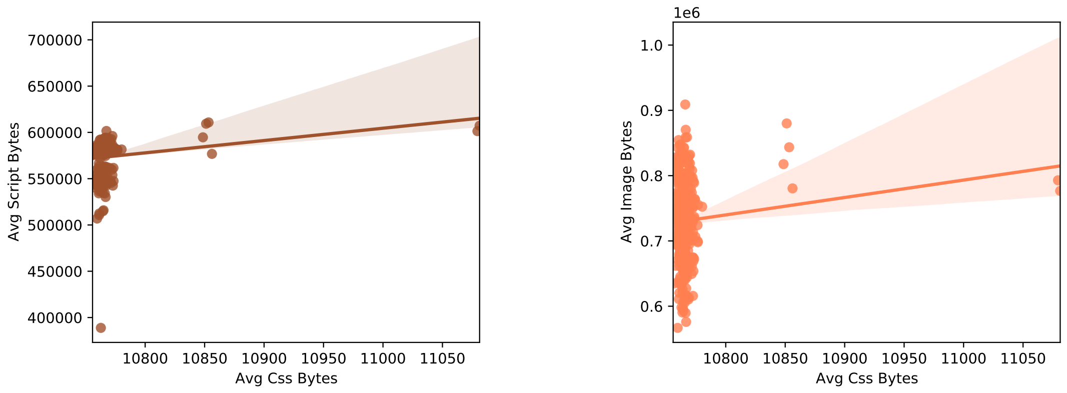
```
corr_metric['Avg Css Bytes'].sort_values(ascending = False)
```

```
## Avg Css Bytes                   1.000000
## Avg Script Bytes                0.176427
## Mdn DNS (ms)                    0.121324
## Avg Image Bytes                 0.110559
## Mdn Render Start (ms)           0.064856
## # Runs                          0.017672
## Mdn Webpage Response (ms)      -0.003841
## Avg Time To First Byte (ms)    -0.042084
## % Availability                 -0.117281
## Name: Avg Css Bytes, dtype: float64
```

Even though, CSS, Script and Image seems to have relation with each other when they are all the components which has been loaded by the site, it suprisingly has such a slight correlation between these aspects, even the Avg Image Bytes has smaller correlation than the Mdn DNS Lookup.

```
fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (12,4))
plt.subplots_adjust(wspace = 0.5)
sns.regplot(data = tripvn, scatter = True, x = 'Avg Css Bytes',
            y = 'Avg Script Bytes', ax = axes[0],
            label = 'Avg Script Bytes', color = 'sienna')

sns.regplot(data = tripvn,  scatter = True, x = 'Avg Css Bytes',
            y = 'Avg Image Bytes', ax = axes[1],
            label = 'Avg Image Bytes', color = 'coral')
plt.show()
```



$Graph 1.4 - 1.5$

We can see such a strange pattern when there is a vertical line in graphs above. It looks like the Avg Css Bytes doesn't change too much and constantly vary around 10750 throughout the whole data set.

Q2. Website speed metrics analysis

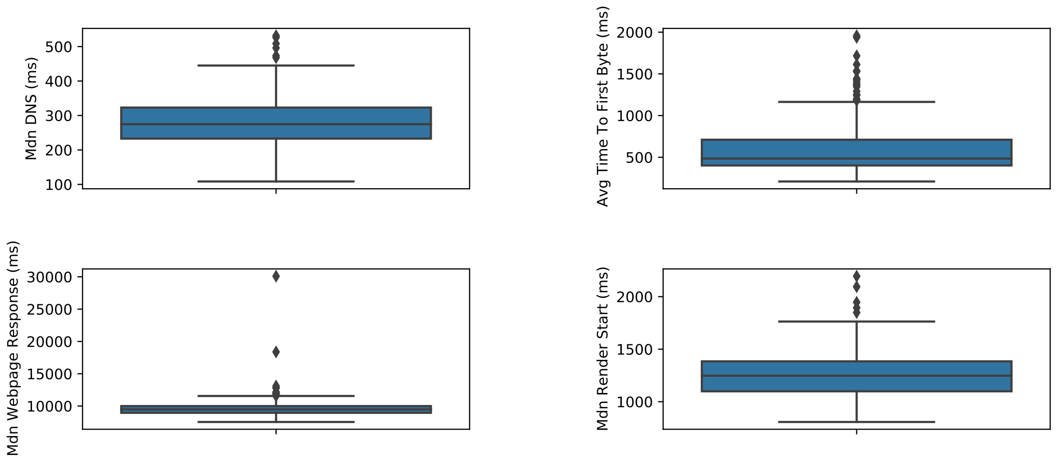
There are 4 metrics used to analyze the website speed

- Mdn DNS
- Avg Time To First Byte
- Mdn Render Start
- Mdn Webpage Response.

Firstly, let's draw the boxplot to see the distribution of these metrics

```
fig, axes = plt.subplots(nrows = 2, ncols = 2, figsize = (12,5))
plt.subplots_adjust(wspace = 0.5, hspace = 0.5)
sns.boxplot(data = tripvn, y = 'Mdn DNS (ms)', ax = axes[0,0])
sns.boxplot(data = tripvn, y = 'Avg Time To First Byte (ms)', ax = axes[0,1])
sns.boxplot(data = tripvn, y = 'Mdn Render Start (ms)', ax = axes[1,1])
sns.boxplot(data = tripvn, y = 'Mdn Webpage Response (ms)', ax = axes[1,0])

plt.show()
```



*Graph*2.1 − 2.4

There are a lot of outliers in these plots and the differences is very significant too when most of the noise outranges the 3rd quartile (maximum) in the boxplots But since we don't actually push it into any machine learning alogrithms, it's okay to just ignore these outliers.

```
tripvn['Mdn DNS (ms)'].describe()
```

```
## count    360.000000
## mean     280.076389
## std       70.513539
## min      108.500000
## 25%      232.875000
## 50%      274.750000
## 75%      323.000000
## max      531.500000
## Name: Mdn DNS (ms), dtype: float64
```

MDN DNS or DNS look up is the amount of time it takes for your DNS provider to translate a domain name into an IP address. With the Mdn DNS in the average of 280, we can say that this website's DNS is very slow when the average of DNS Look up falls around 20-120ms

When it comes to Avg Time To First Byte, we see a small upward trend in the graph 1.3, when Mdn DNS increases, the Avg Time To First Byte rise slightly from 250 (ms) to around 500 (ms).

> From the standard definition, Avg Time To First Byte is a measurement of time a browser (user) has to wait to receive the first byte of data from the server-side (breeze.io), but according to several researchs and blogs recently, the actual Avg Time To First Byte measures the time it takes for the first HTTP response to be received, but not the time it takes for the whole page to be sent. So just only by testing the TTFB of a website, we can not conclude that the rest of the page can be load quickly too. And even if we do that, we can take a look at the average number of this:

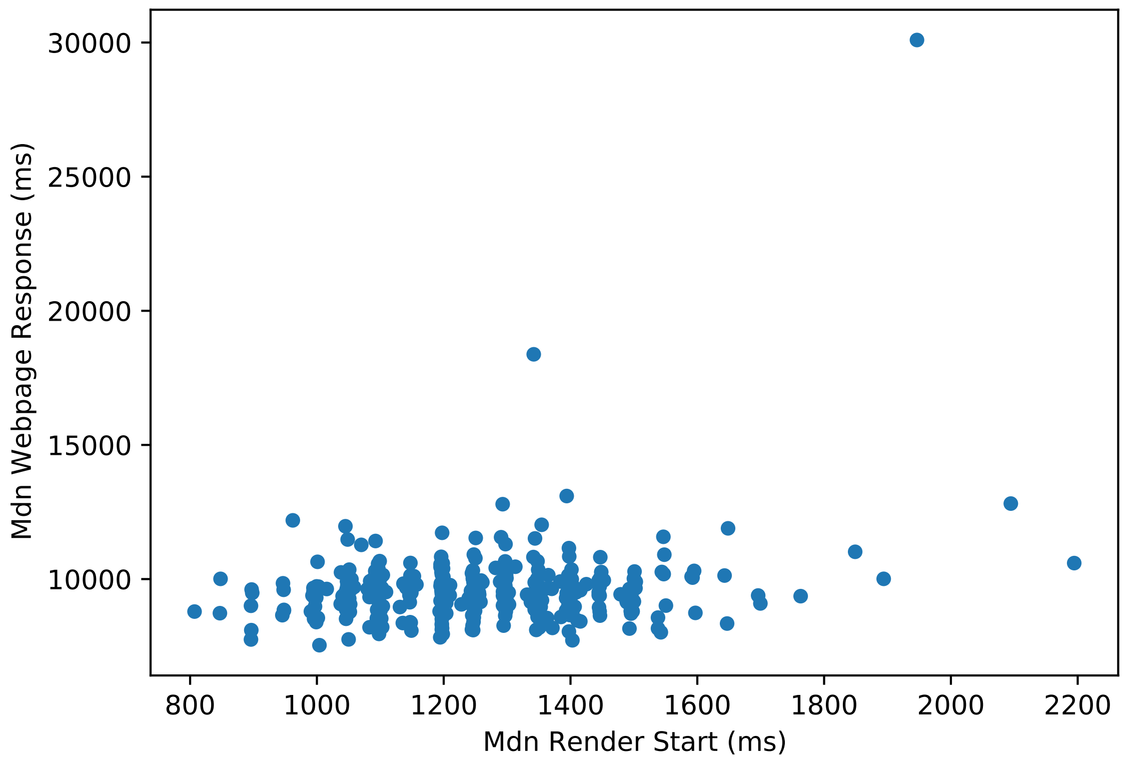
```
tripvn['Avg Time To First Byte (ms)'].describe()
```

```
## count      360.000000
## mean       593.206944
## std        286.766903
## min        209.750000
## 25%        401.747500
## 50%        484.170000
## 75%        710.352500
## max       1957.580000
## Name: Avg Time To First Byte (ms), dtype: float64
```

According to KeyCDN, the proper Avg Time To First Byte falls about less than 400 ms, with 75% of the data has TTFB bigger than 400 ms. So there is no way this is a fast page for user.

Let's quickly draw a scatter graph between Mdn Render Start and Mdn Webpage Response

```
tripvn.plot(kind = 'scatter', x = 'Mdn Render Start (ms)', y = 'Mdn Webpage Response (ms)')
```



We can see that Mdn Webpage Response always lies around 10000ms and is very constant even the Mdn Render Start changes.

```
tripvn['Mdn Webpage Response (ms)'].describe()
```

```
## count       360.000000
## mean       9593.341667
## std       1461.569220
## min       7535.500000
## 25%       8954.875000
## 50%       9486.750000
## 75%       9997.625000
## max      30094.000000
## Name: Mdn Webpage Response (ms), dtype: float64
```

In the recent year, with the development of technology, we have been able to improve the website response time to catch up with user's expectation, specifically, that is around 2-3s, and in 2018 Google had successfully upgrade their Webpage Response to around 1.3s. But that's is Google, so we cannot take that for granted for our website. But normally, a 10-second delay often can make users leave your site immediately, well unless they have a must to use your page. And a key takeaway here is that Webpage Response is not only the amount time for the website to send the response to your browser, but the time from your first byte to the last.

```
tripvn['Mdn Render Start (ms)'].describe()
```

```
## count     360.000000
## mean     1255.504167
## std       192.891979
## min       807.000000
## 25%      1099.000000
## 50%      1247.500000
## 75%      1384.250000
## max      2194.500000
## Name: Mdn Render Start (ms), dtype: float64
```

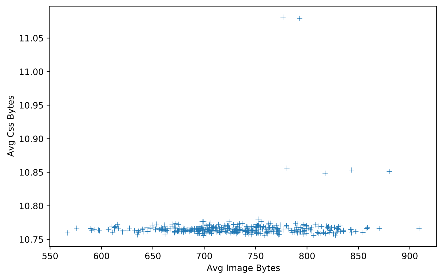
Nonetheless, suprisingly, the Render Start is way more faster than the avarage of 2 seconds, when 75% of the test are smaller than 1.3 seconds. It means like after the webpage response is received (it takes a lot of time), the page was almost instantly start render the first content for the users to interact with. Despite having this speed, the time spent for waiting others to load still makes user leave the site before recognizing this fantastic feature.

Q3. Website Content Analysis

At the given time, the size of Traveloka is 878.1 KB, and it has been 3 years from that, so let's take the real current size of this 2 pages. According to Pingdom Website Speed Test (tools.pingdom.com), the respective size of Traveloka and TripVn is 2.1 MB and 660.8 KB Take the look at the numbers of Avg Image Bytes and Avg Script Bytes, we see it takes up to minutes to load all the image and script So, it is better if we scale it down to second for easier examination.

```
fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (20,5))
plt.subplots_adjust(wspace = 0.5, hspace = 0.5)
sns.scatterplot(tripvn['Avg Image Bytes'] / 1000, tripvn['Avg Css Bytes'] / 1000,
                marker = '+', ax = axes[0])

sns.scatterplot(tripvn['Avg Script Bytes'] / 1000, tripvn['Avg Css Bytes'] / 1000,
                marker = 'x', ax = axes[1])
plt.show()
```



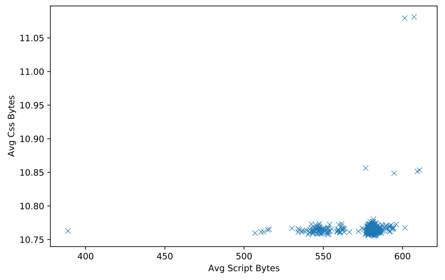


Image Solution

- Here we can see that the image and script takes far more time to load in comparison with CSS. So the problems here might lie in the number of images this site is using, or either the size of the picture. When we look at these two pages, we can easily see that Traveloka uses CSS sprites way more often than TripVN with individual pictures, because when you separate the images you make your browser constantly trying to retrieve images every time certain pages on your site load.
- And besides using CSS sprites, reducing or compressing the images' size and number of images, you can also use Lazy loading, make your website becomes a dynamic website, or in another term,if users don't scroll all the way down, images placed at the bottom of the page won't even be loaded.
- Additionally, the mobile users and website users should have different page look so that you are not forcing the browser to load a variety of scripts or images that won't be useful for certain devices or viewpoints. Selecting appropriate image format can also a great solution to optimize your images.

Script Solution

- With the script loading time, similar methods can be applied to like compressing content using HTTP Compression.
- In addition, TripVn can take advantage of reducing the length of the text, because most of users nowadays are just way too lazy to read, so try to reduce the number of words and give some short motto.
- Some other suggestion like putting script references at the bottom and/or stylesheet references at the top can also benefits the scripts loading too.

CSS Solution

- Average CSS loading of TripVN takes around 10 seconds to be fully loaded, and currently there is no standard criteria for this, but placing JavaScript and CSS in external files can also boost the loading time due to preventing it is downloaded every time an HTML document is requested.

Q4. Website Ability

The Availability normally relates to the time when user access to the page, so let's extract twp separate column: Hour and Day of Week

```
tripvn['Hour'] = tripvn['Time'].map(lambda x: x.split()[1])
tripvn.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## Int64Index: 360 entries, 0 to 359
## Data columns (total 11 columns):
##  #   Column                       Non-Null Count  Dtype
## ---  ------                       --------------  -----
##  0   Time                         360 non-null    object
##  1   Mdn DNS (ms)                 360 non-null    float64
##  2   Avg Time To First Byte (ms)  360 non-null    float64
##  3   Mdn Webpage Response (ms)    360 non-null    float64
##  4   Mdn Render Start (ms)        360 non-null    float64
##  5   Avg Image Bytes              360 non-null    float64
##  6   Avg Script Bytes             360 non-null    float64
##  7   Avg Css Bytes                360 non-null    float64
##  8   % Availability               360 non-null    float64
##  9   # Runs                       360 non-null    int64
##  10  Hour                         360 non-null    object
## dtypes: float64(8), int64(1), object(2)
## memory usage: 33.8+ KB
```

Create function to convert from date to day of week

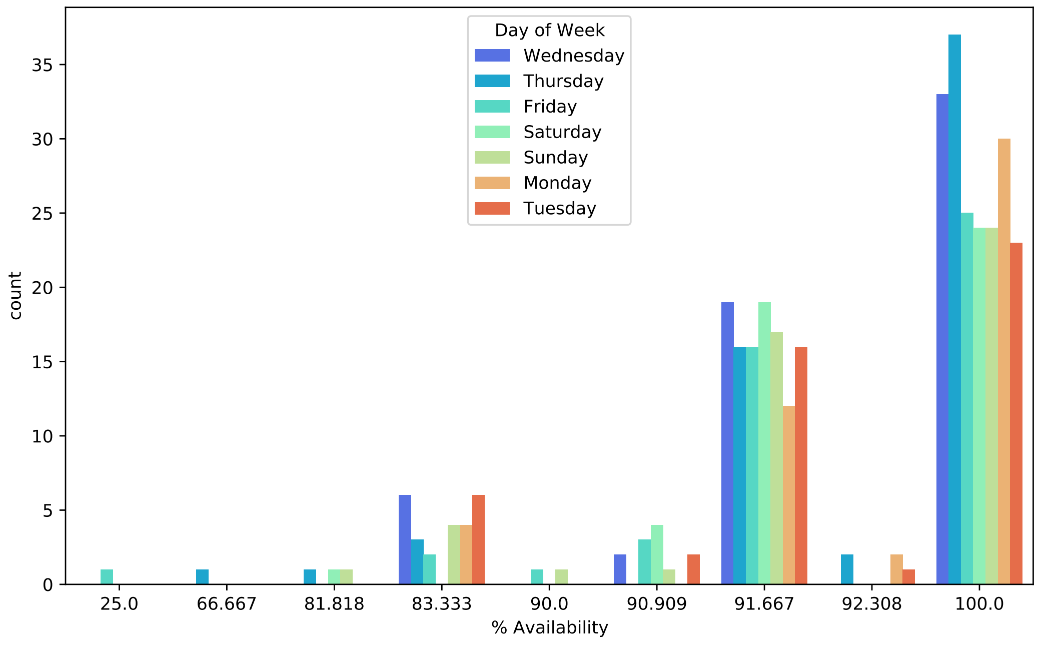
```
import datetime
import calendar

def findDay(date):
    born = datetime.datetime.strptime(date, '%m/%d/%Y').weekday()
    return (calendar.day_name[born])
```

```
tripvn['Day of Week'] = tripvn['Time'].map(lambda x: findDay(x.split()[0]))
tripvn.head()
```

```
##               Time  Mdn DNS (ms)  ...  Hour  Day of Week
## 0  11/01/2017 0:00         287.5  ...  0:00    Wednesday
## 1  11/01/2017 2:00         254.5  ...  2:00    Wednesday
## 2  11/01/2017 4:00         228.0  ...  4:00    Wednesday
## 3  11/01/2017 6:00         180.0  ...  6:00    Wednesday
## 4  11/01/2017 8:00         227.0  ...  8:00    Wednesday
##
## [5 rows x 12 columns]
```

```
plt.figure(figsize = (10,6))
sns.countplot('% Availability', hue = 'Day of Week', data = tripvn, palette = 'rainbow')
plt.show()
```



```
tripvn['% Availability'].describe()
```

```
## count    360.000000
## mean      95.262775
## std        6.689858
## min       25.000000
## 25%       91.667000
## 50%      100.000000
## 75%      100.000000
## max      100.000000
## Name: % Availability, dtype: float64
```

The availability of TripVn is relatively high, as a sign of stablity, and most of the time when the availability falls is lying around weekend, and maybe a sign for maintenance, but it also has a time when it drop to 25%, let's see what is the day it was drop to that low.

```
tripvn.loc[tripvn['% Availability'] <= 25]
```

```
##                         Time  Mdn DNS (ms)  ...      Hour  Day of Week
## 192  11/17/2017 12:00:00 AM         275.5  ...  12:00:00       Friday
##
## [1 rows x 12 columns]
```

It happened on Friday and at 12:00:00 pm, normally this is the time people take some nap and rest, and it happend only once time, so it's not really a serious problem, this might be caused by a crash from the server.