Video Games

Josh Gardner
DSC 680: Applied Data Science

Introduction

In 2018, the gaming industry generated around $135 billion, marking a 10.9 percent increase from 2017. Most of the growth in the gaming industry has been in mobile gaming, generating $63.2 billion. Console gaming generated $38.3 billion and pc gaming generated $33.4 billion. With so much money on the line, it is important to know which games will be successful.

Looking at the datasets about video games available at Kaggle (https://www.kaggle.com/datasets?search=video+games), we can start asking questions and see if we can predict what the user's rating of the game will be.

For this project, I have the following 3 research questions:

1. Do ratings influence the number of games bought?
2. Do critic ratings influence user ratings?
3. How much time is spent beforea game is rated? Is this time different between critic and user ratings?
4. Can machine learning predict what the user ratings will be?

Setting Up

The first things that we need to do is to import and clean the data.

In [122]:
```
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns
import numpy as np
import statsmodels.formula.api as sm
from scipy import stats
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn import metrics
from scipy.stats.stats import pearsonr
```

In [2]:
```
xbox_sales = pd.read_csv('C:/Users/yasam/OneDrive/Documents/Grad School/DSC680
Applied Data Science/Project 3/data/XboxOne_GameSales.csv', encoding='latin-1'
)
ps4_sales = pd.read_csv('C:/Users/yasam/OneDrive/Documents/Grad School/DSC680
 Applied Data Science/Project 3/data/PS4_GamesSales.csv', encoding='latin-1')
game_sales = pd.read_csv('C:/Users/yasam/OneDrive/Documents/Grad School/DSC680
Applied Data Science/Project 3/data/Video_Game_Sales_as_of_Jan_2017.csv', enco
ding='latin-1')
play_time = pd.read_csv('C:/Users/yasam/OneDrive/Documents/Grad School/DSC680
 Applied Data Science/Project 3/data/GamesDataset.csv', encoding='latin-1')
ratings = pd.read_csv('C:/Users/yasam/OneDrive/Documents/Grad School/DSC680 Ap
plied Data Science/Project 3/data/gamecomm.csv', encoding='latin-1')
```

In [3]:
```
set(list(play_time['Variable']))
```

Out[3]:
```
{'Critic_Count',
 'Critic_Score',
 'Developer',
 'EU_Sales',
 'Genre',
 'Global_Sales',
 'JP_Sales',
 'MeanPlaytime',
 'MedianPlaytime',
 'NA_Sales',
 'Other_Sales',
 'Publisher',
 'Rating',
 'User_Count',
 'User_Score',
 'numberOfObservances'}
```

In [4]:
```
# Ok, so I want 3 variables from the play_time set. MeanPlaytime, MedianPlayti
me, and numberOfObservances.

mean_play_time = play_time[play_time['Variable'] == 'MeanPlaytime']
median_play_time = play_time[play_time['Variable'] == 'MedianPlaytime']
observances_play_time = play_time[play_time['Variable'] == 'numberOfObservance
s']

# I need to reset the indexes
mean_play_time.reset_index(inplace = True)
median_play_time.reset_index(inplace = True)
observances_play_time.reset_index(inplace = True)

# Drop the unnecessary columns
mean_play_time = mean_play_time.drop('index', 1).drop('Index', 1).drop('Platfo
rm', 1).drop('Year_of_Release', 1).drop('Variable', 1)
median_play_time = median_play_time.drop('index', 1).drop('Index', 1).drop('Pl
atform', 1).drop('Year_of_Release', 1).drop('Variable', 1)
observances_play_time = observances_play_time.drop('index', 1).drop('Index', 1
).drop('Platform', 1).drop('Year_of_Release', 1).drop('Variable', 1)

# I need to change the Value variable into an int
l1 = [mean_play_time, median_play_time, observances_play_time]
def to_num(df):
    test = []
    for i in range(df.shape[0]):
        test.append(float(df['Value'][i]))
    df['Value'] = test

for record in l1:
    to_num(record)


# Let's clean up the column names for each.
mean_play_time.columns = ['name', 'mean_playtime']
median_play_time.columns = ['name', 'median_playtime']
observances_play_time.columns = ['name', 'observations']

# I need to group the former value variable by game
mean_play_time = mean_play_time.groupby('name').sum().reset_index()
median_play_time = median_play_time.groupby('name').sum().reset_index()
observances_play_time = observances_play_time.groupby('name').sum().reset_inde
x()

# Now I want to combine these three sets into a single dataframe again.
step1 = pd.merge(mean_play_time, median_play_time, on='name', how='outer')
cleaned_playtime = pd.merge(step1, observances_play_time, on='name', how='oute
r')
```

In [5]:
```
max(cleaned_playtime['observations'])
```

Out[5]: 3680.0

And that does it for the cleaning the `play_time` variable. Fortunately, the `game_sales` dataset includes the critic and user ratings for the different games. Let's take a look at the `game_sales` and clean it.

In [6]:
```
# cleaning Year_of_Release to be ints. Missing values are coded to 0.
year = []
for i in range(game_sales.shape[0]):
    try:
        year.append(int(game_sales['Year_of_Release'][i]))
    except:
        year.append(int(0))
game_sales['Year_of_Release'] = year

# Getting the lowercase names of the columns
conames = []
for name in game_sales.columns:
    conames.append(name.lower())

# Renaming the columns with the lowercase names.
game_sales.columns = conames
```

In [7]:
```
game_sales.sort_values(by=['name'])
```

Out[7]:

	name	platform	year_of_release	genre	publisher	na_sales	eu_sal
8394	.hack//G.U. Vol.1//Rebirth	PS2	2006	Role-Playing	Namco Bandai Games	0.00	0.00
7130	.hack//G.U. Vol.2//Reminisce	PS2	2006	Role-Playing	Namco Bandai Games	0.11	0.09
8651	.hack//G.U. Vol.2//Reminisce (jp sales)	PS2	2006	Role-Playing	Namco Bandai Games	0.00	0.00
8347	.hack//G.U. Vol.3//Redemption	PS2	2007	Role-Playing	Namco Bandai Games	0.00	0.00
1568	.hack//Infection Part 1	PS2	2002	Role-Playing	Atari	0.49	0.38
...	...	...	...	...	...	...	...
16593	thinkSMART: Chess for Kids	DS	2011	Misc	Mentor Interactive	0.01	0.00
645	uDraw Studio	Wii	2010	Misc	THQ	1.65	0.57
8285	uDraw Studio: Instant Artist	Wii	2011	Misc	THQ	0.06	0.09
15696	uDraw Studio: Instant Artist	X360	2011	Misc	THQ	0.01	0.01
480	wwe Smackdown vs. Raw 2006	PS2	2005	Fighting	THQ	1.57	1.02

17416 rows × 15 columns

Now, we need to group the `game_sales` by each game, but we need two grouping methods. For the sales records, we need to sum them together, but for the scores, we need to average them together.

In [8]:
```
# First, get the columns that can be summed together
l1 = ['name', 'genre', 'publisher', 'na_sales', 'eu_sales',
         'jp_sales', 'other_sales', 'global_sales', 'critic_count', 'user_coun
t', 'rating']
step1 = game_sales[l1].groupby('name').sum().reset_index()

# Now get the columns that need to be averaged
l2 = ['name', 'year_of_release', 'critic_score', 'user_score']
step2 = game_sales[l2].groupby('name').mean().reset_index().round(2)

# Now to join the two datasets
games_cleaned = pd.merge(step1, step2, on='name', how='outer')
```

Ok, this should be enough cleaning of `game_sales` for now. There are still a lot of missing values that we will need to consider later on, but for now, I will be leaving the missing values within the dataframe.

The next thing up is to merge the two cleaned dataframes together. I want to keep all of the data, including any missing data, so I'll need to use a full outer join again.

In [9]:
```
gamedf = pd.merge(games_cleaned, cleaned_playtime, on='name', how='outer')

# for later on, let's get a dataframe with no missing values
gamedf_no_na = gamedf.dropna().reset_index()
```

Alright, looks like we have our data combined. We're ready to start exploring the data.

One-Dimensional Exploration

First up is the one-dimensional exploration. Let's start out by looking at how many games have been published for each platform.

In [10]:
```
game_sales[['platform', 'name']].groupby('platform').count().reset_index()
```

Out[10]:

	platform	name
0	2600	133
1	3DO	3
2	3DS	553
3	DC	52
4	DS	2251
5	G	98
6	GBA	844
7	GC	563
8	GEN	27
9	GG	1
10	N64	319
11	NES	98
12	NG	12
13	PC	1128
14	PCFX	1
15	PS	1200
16	PS2	2206
17	PS3	1362
18	PS4	424
19	PSP	1304
20	PSV	503
21	SAT	173
22	SCD	6
23	SNES	239
24	TG16	2
25	WS	7
26	Wii	1359
27	WiiU	153
28	X	833
29	X360	1298
30	XOne	264

The Nintendo DS has the most games published within this dataset. That's pretty impressive. I know this data isn't completely up-to-date, but it's still impressive that a handheld console has the highest number of games.

Let's move on to looking at the actual data and plotting the histograms for each of the numeric variables.

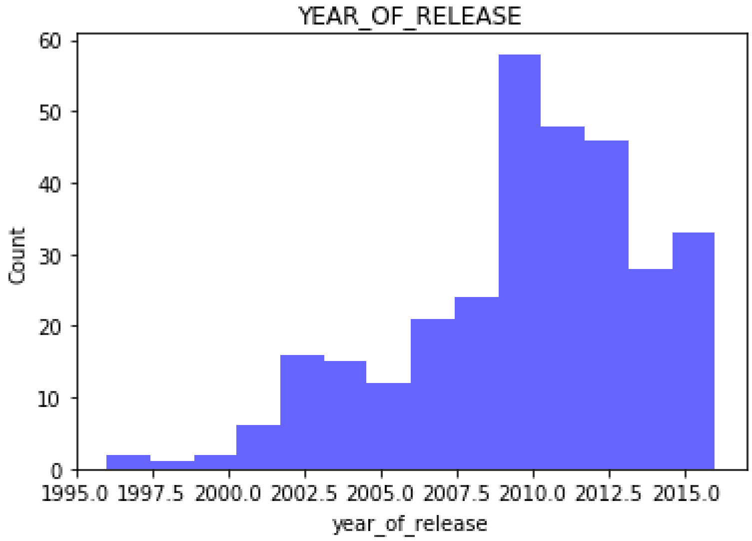
In [11]:
```
def make_hist(var, bins='auto'):
    n, bins, patches = plt.hist(x=gamedf_no_na[var], bins=bins, color='blue',
                                alpha=.6)
    plt.xlabel(var)
    plt.ylabel('Count')
    plt.title(var.upper())
    plt.show()

def make_vio(vari):
    tips = px.data.tips()
    fig = px.violin(tips, y=gamedf_no_na[vari], box=True, points='all')
    fig.update_layout(title_text=vari.upper())
    fig.show()

'''
Source for violin plots:
plotly. "plotly Graphing Libraries." Retrieved 26 Oct. 2019 from https://plot.
ly/python/violin/#targetText=Violin%20Plots%20in%20Python&targetText=A%20violi
n%20plot%20is%20a,list%20of%20other%20statistical%20charts.
'''
```

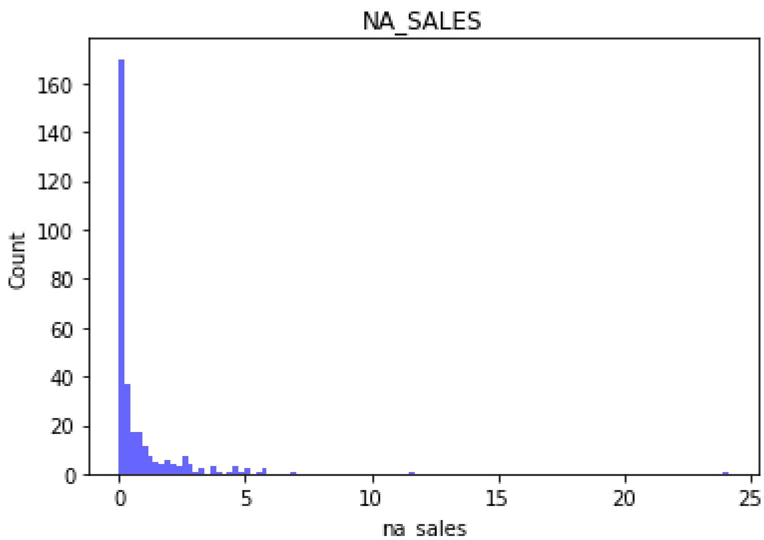
Out[11]:
```
'\nSource for violin plots:\nplotly. "plotly Graphing Libraries." Retrieved 2
6 Oct. 2019 from https://plot.ly/python/violin/#targetText=Violin%20Plots%20i
n%20Python&targetText=A%20violin%20plot%20is%20a,list%20of%20other%20statisti
cal%20charts.\n'
```

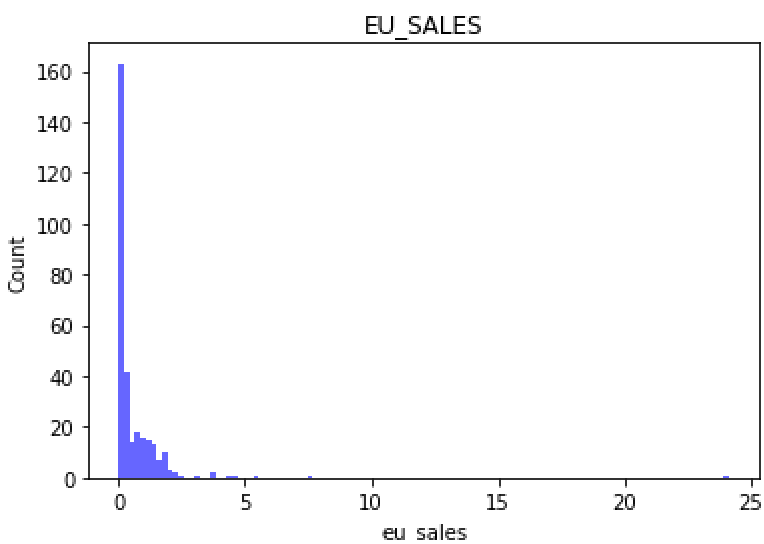
Source for Violin Plots

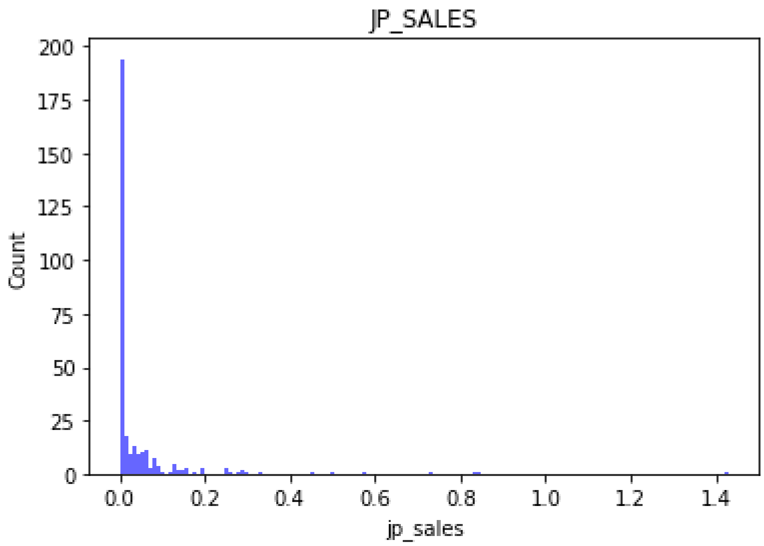
1. *plotly.* "plotly Graphing Libraries." Retrieved 26 Oct. 2019 from plotly Graphing Libraries (https://plot.ly/python/violin/#targetText=Violin%20Plots%20in%20Python&targetText=A%20violin%20plot%20i

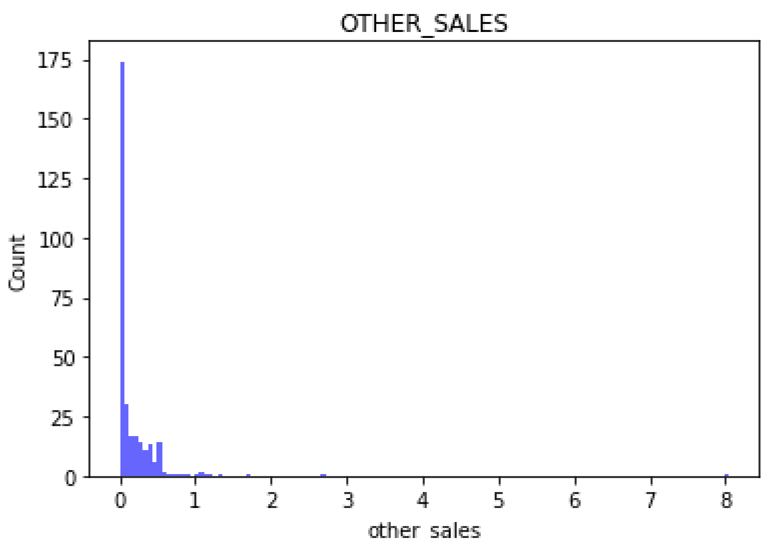
In [12]:
```
vars = ['year_of_release', 'na_sales', 'eu_sales', 'jp_sales', 'other_sales',
'global_sales',
        'critic_score', 'critic_count', 'user_score', 'user_count', 'mean_play
time',
        'median_playtime', 'observations']

for name in vars:
    make_hist(name)
```

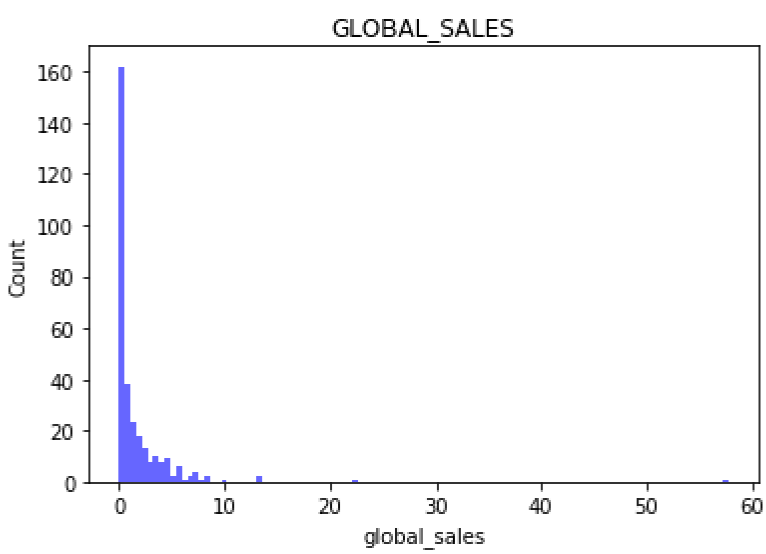

YEAR_OF_RELEASE
Count
year_of_release
1995.0 1997.5 2000.0 2002.5 2005.0 2007.5 2010.0 2012.5 2015.0
0 10 20 30 40 50 60

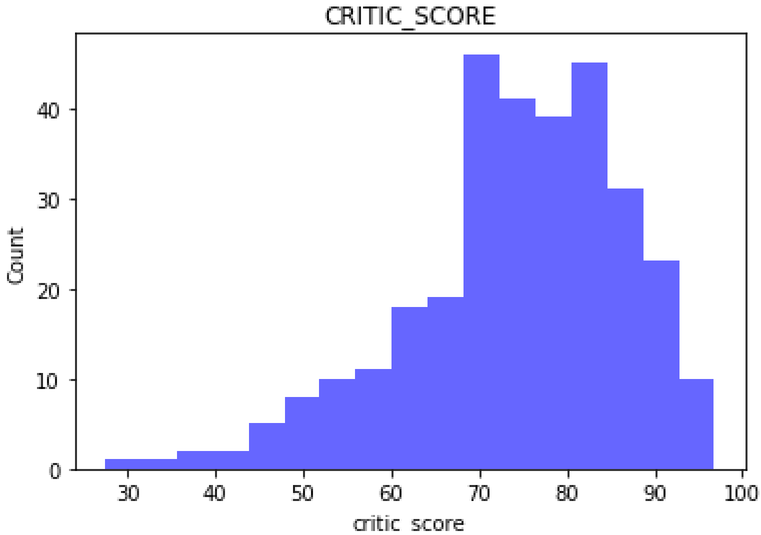

NA_SALES
Count
na_sales
0 5 10 15 20 25
0 20 40 60 80 100 120 140 160

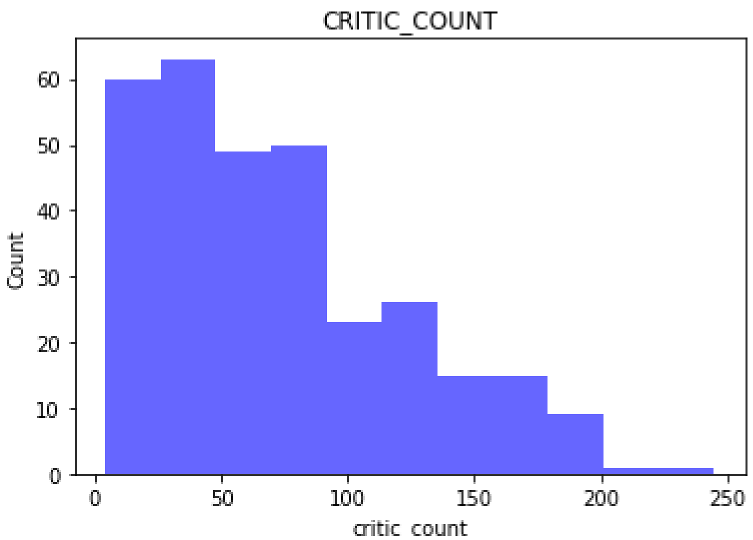

EU_SALES
Count
eu_sales
0 5 10 15 20 25
0 20 40 60 80 100 120 140 160

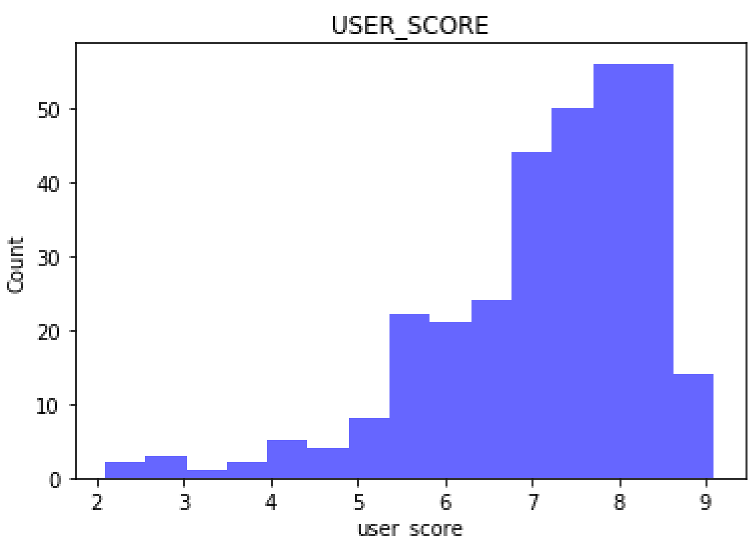

JP_SALES
Count
jp_sales
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
25
50
75
100
125
150
175
200

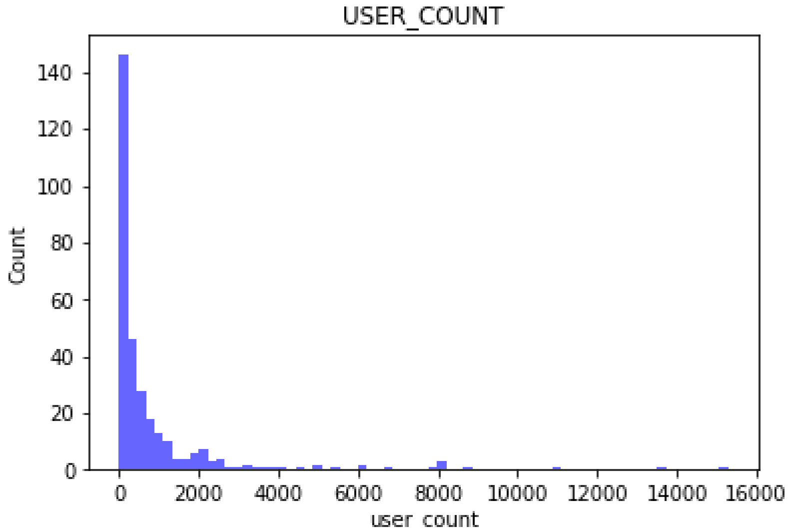

OTHER_SALES
Count
other_sales
0
1
2
3
4
5
6
7
8
0
25
50
75
100
125
150
175

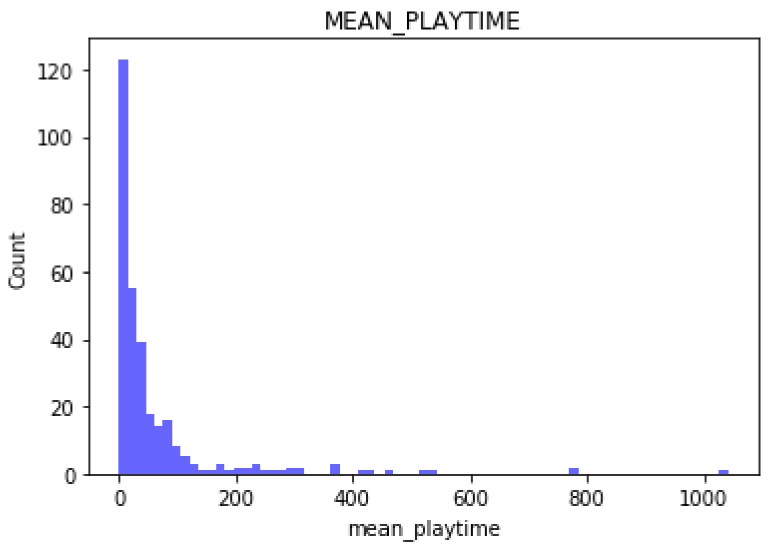

GLOBAL_SALES
Count
global_sales
0
10
20
30
40
50
60
0
20
40
60
80
100
120
140
160

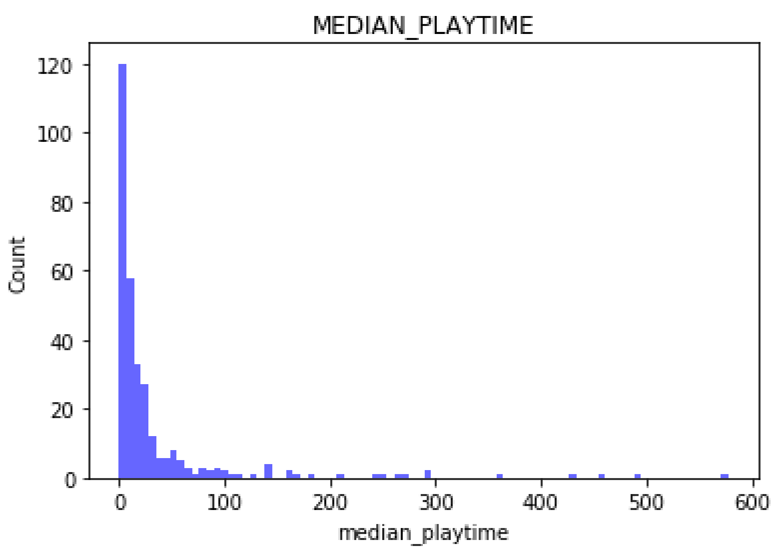

CRITIC_SCORE
Count
0
10
20
30
40
30
40
50
60
70
80
90
100
critic_score

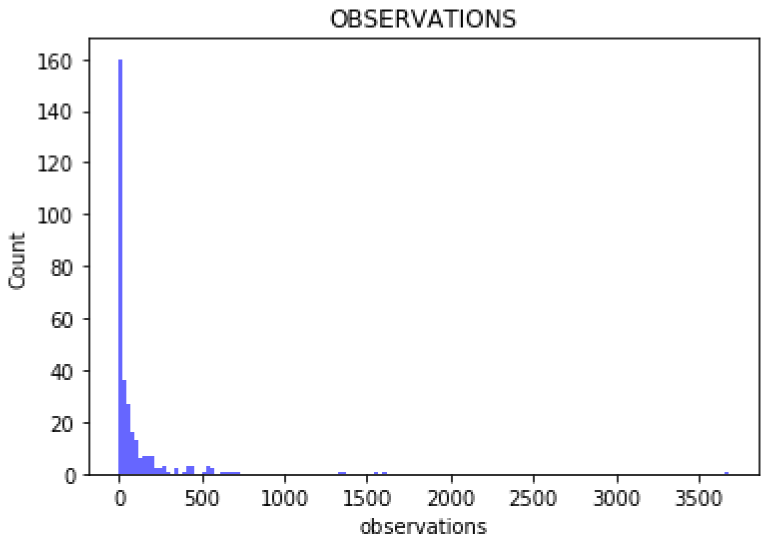

CRITIC_COUNT
Count
0
10
20
30
40
50
60
0
50
100
150
200
250
critic_count


USER_SCORE
Count
0
10
20
30
40
50
2
3
4
5
6
7
8
9
user_score

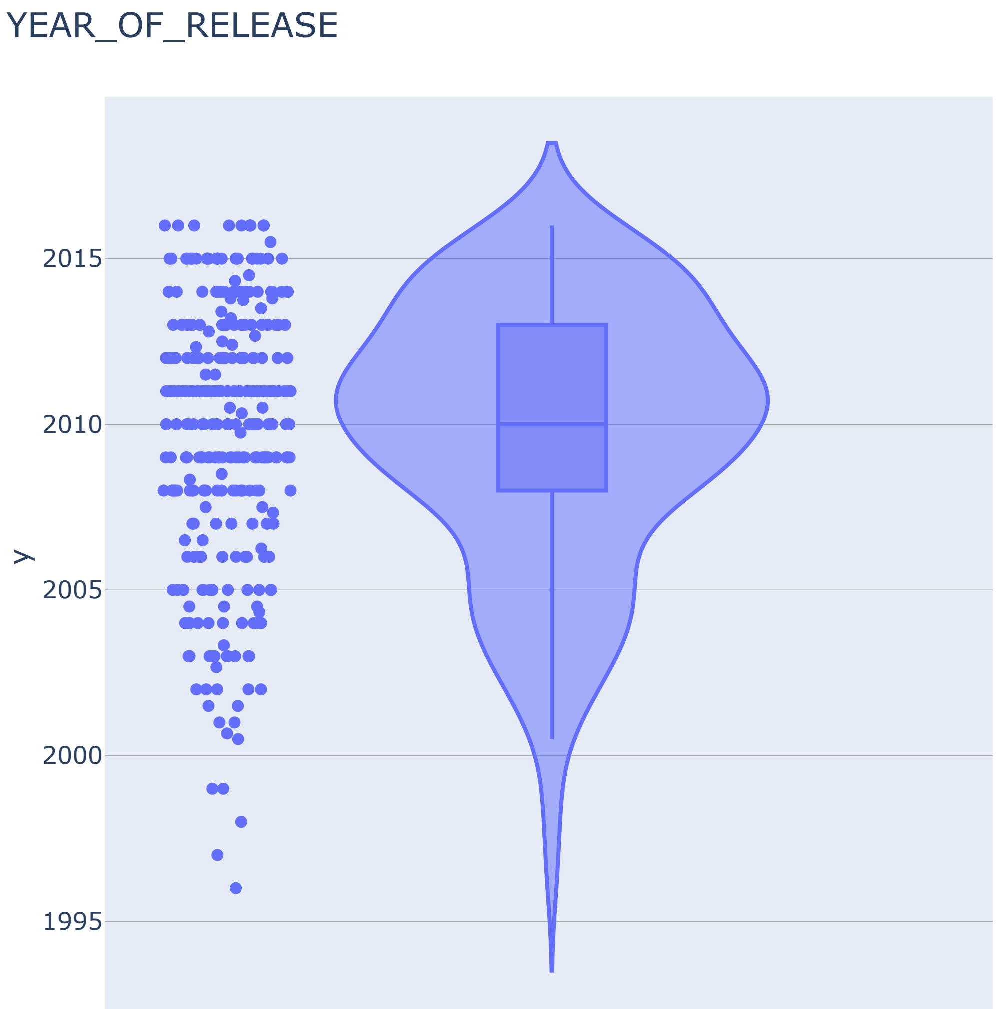

USER_COUNT
Count
user_count
0
20
40
60
80
100
120
140
0
2000
4000
6000
8000
10000
12000
14000
16000

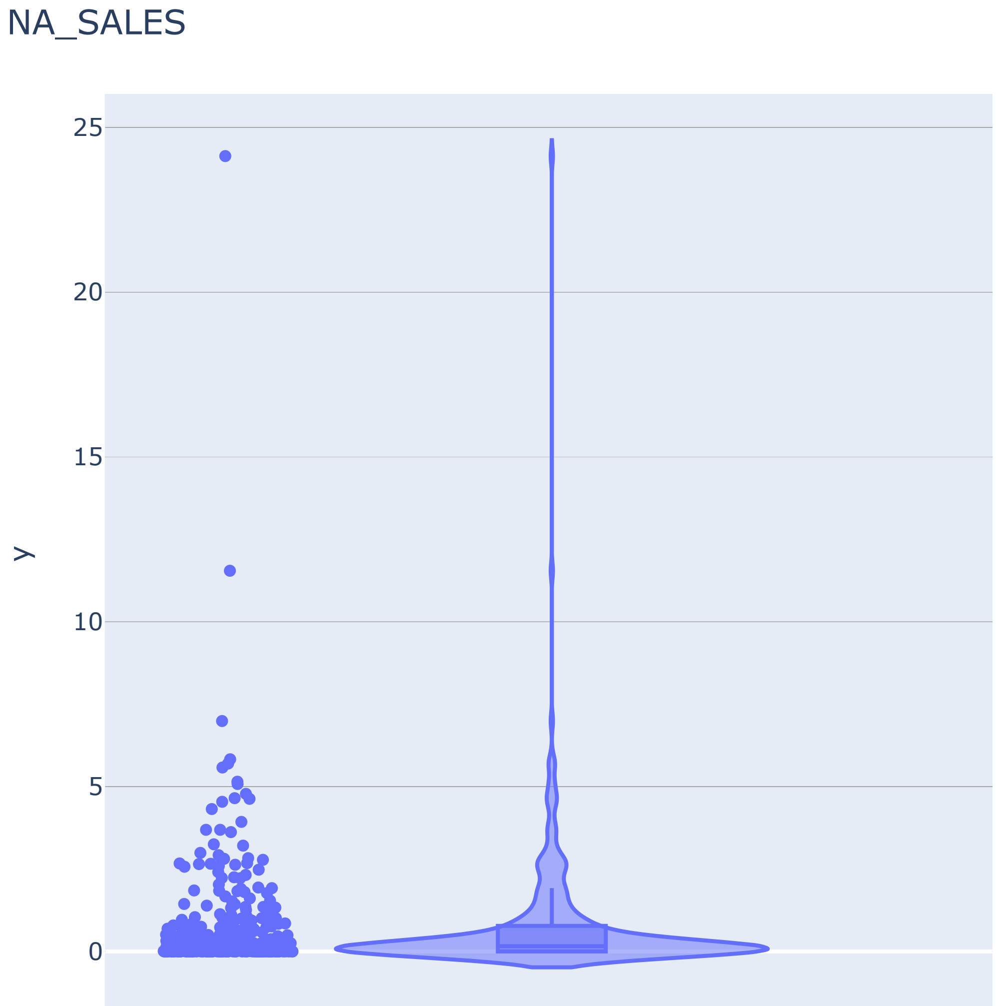

MEAN_PLAYTIME
Count
mean_playtime
0
20
40
60
80
100
120
0
200
400
600
800
1000

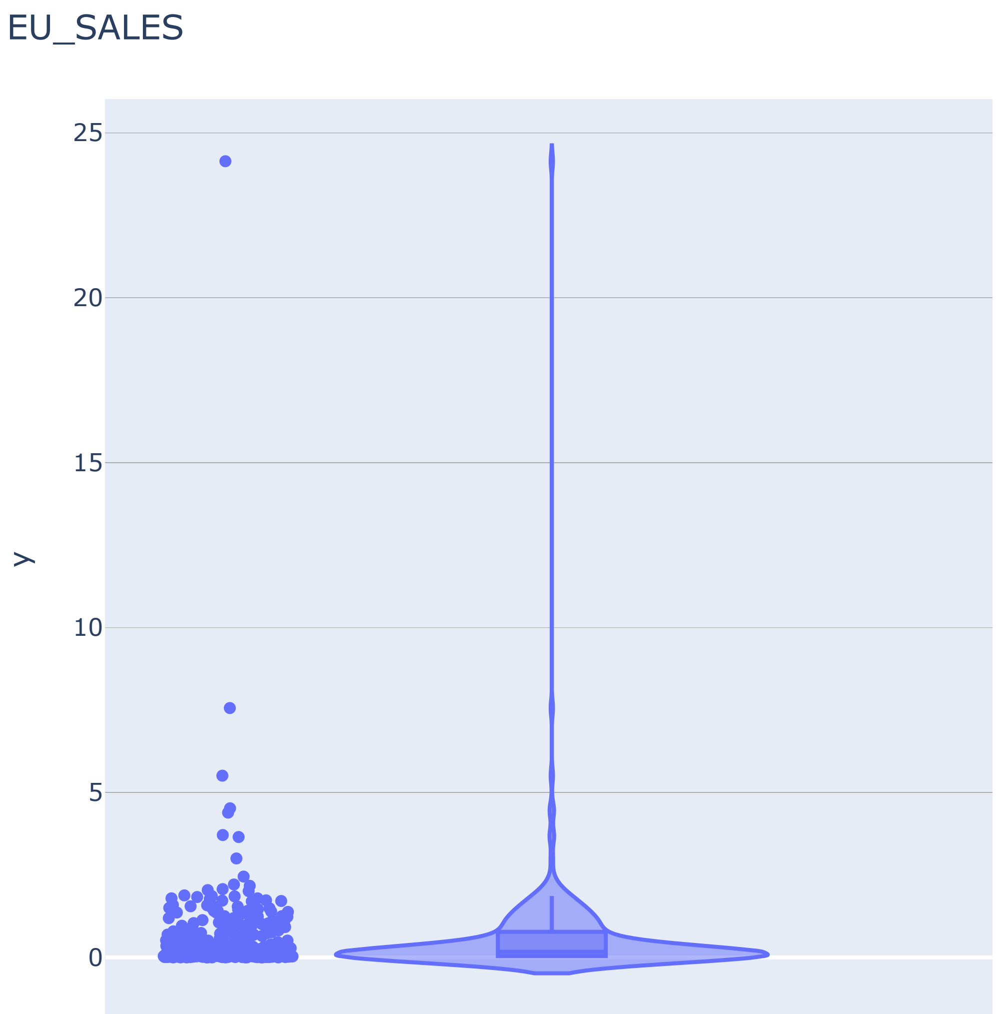

MEDIAN_PLAYTIME
Count
median_playtime
0
20
40
60
80
100
120
0
100
200
300
400
500
600


OBSERVATIONS
160
140
120
100
80
60
40
20
0
Count
0
500
1000
1500
2000
2500
3000
3500
observations

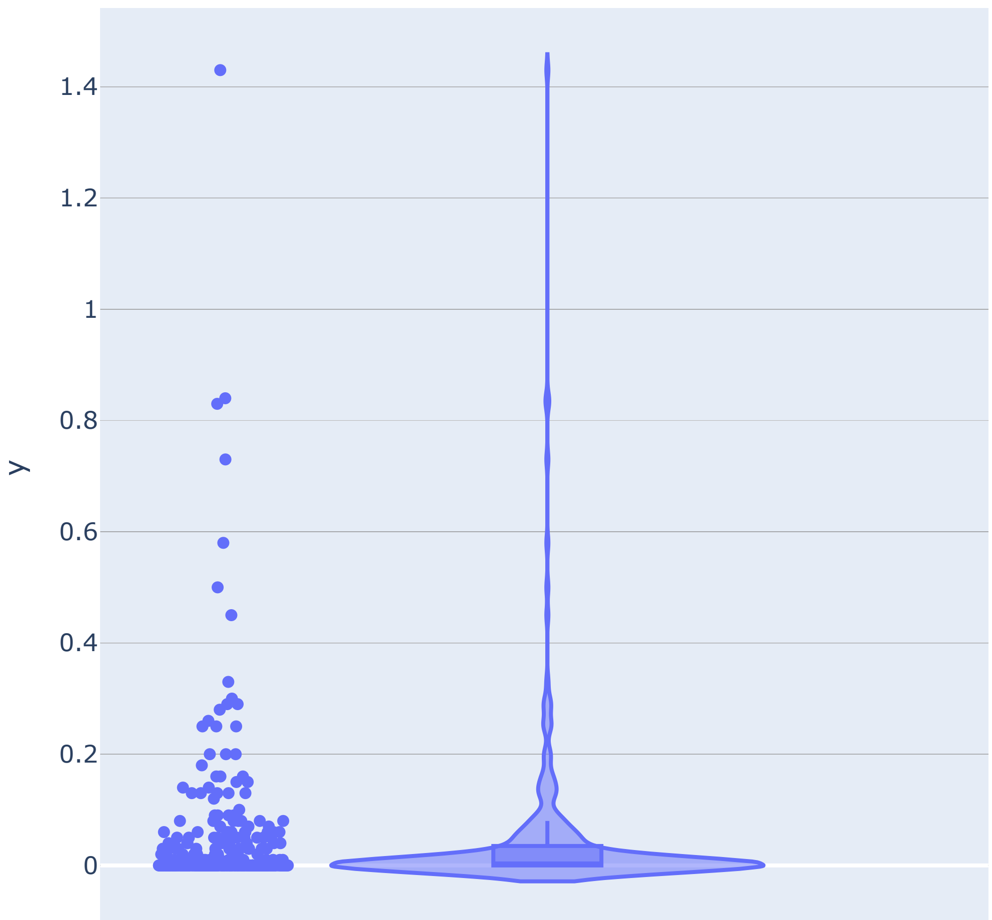
In [13]:
```
for name in vars:
    make_vio(name)
```

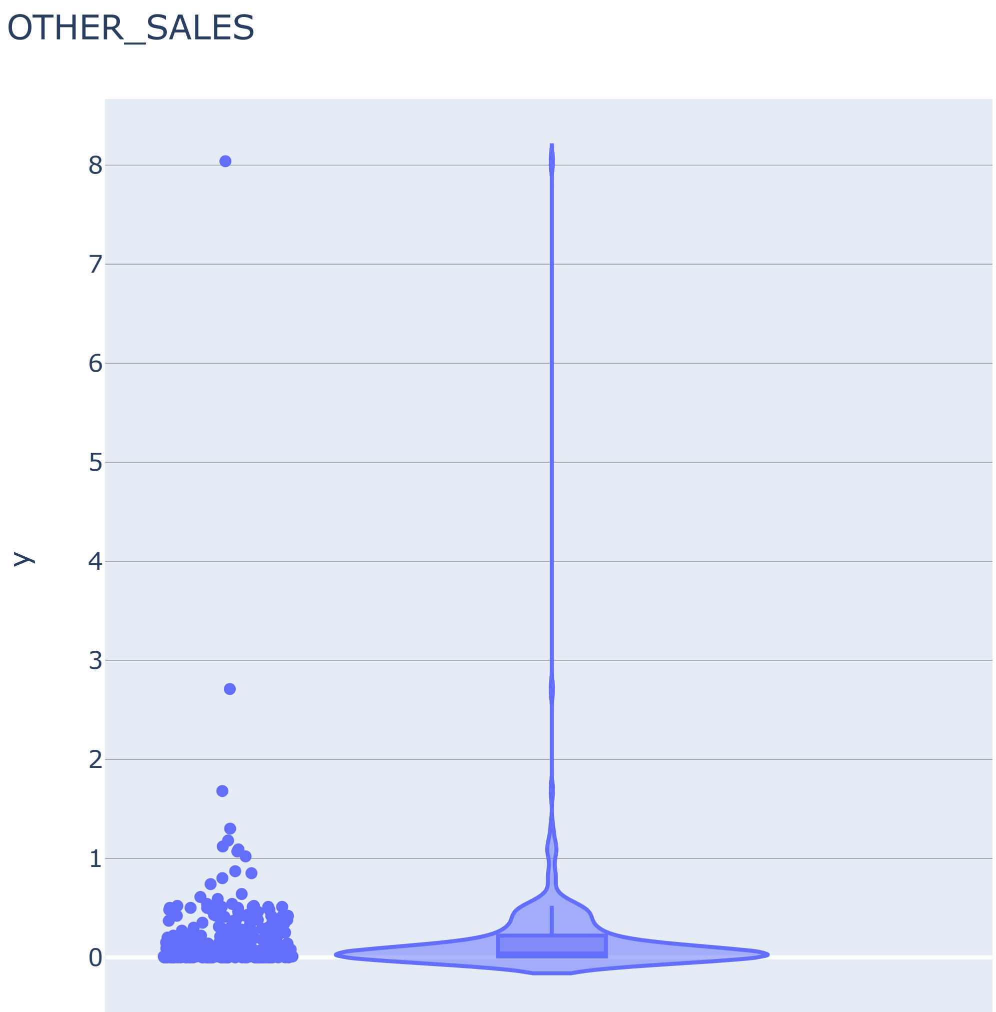

YEAR_OF_RELEASE
y
2015
2010
2005
2000
1995

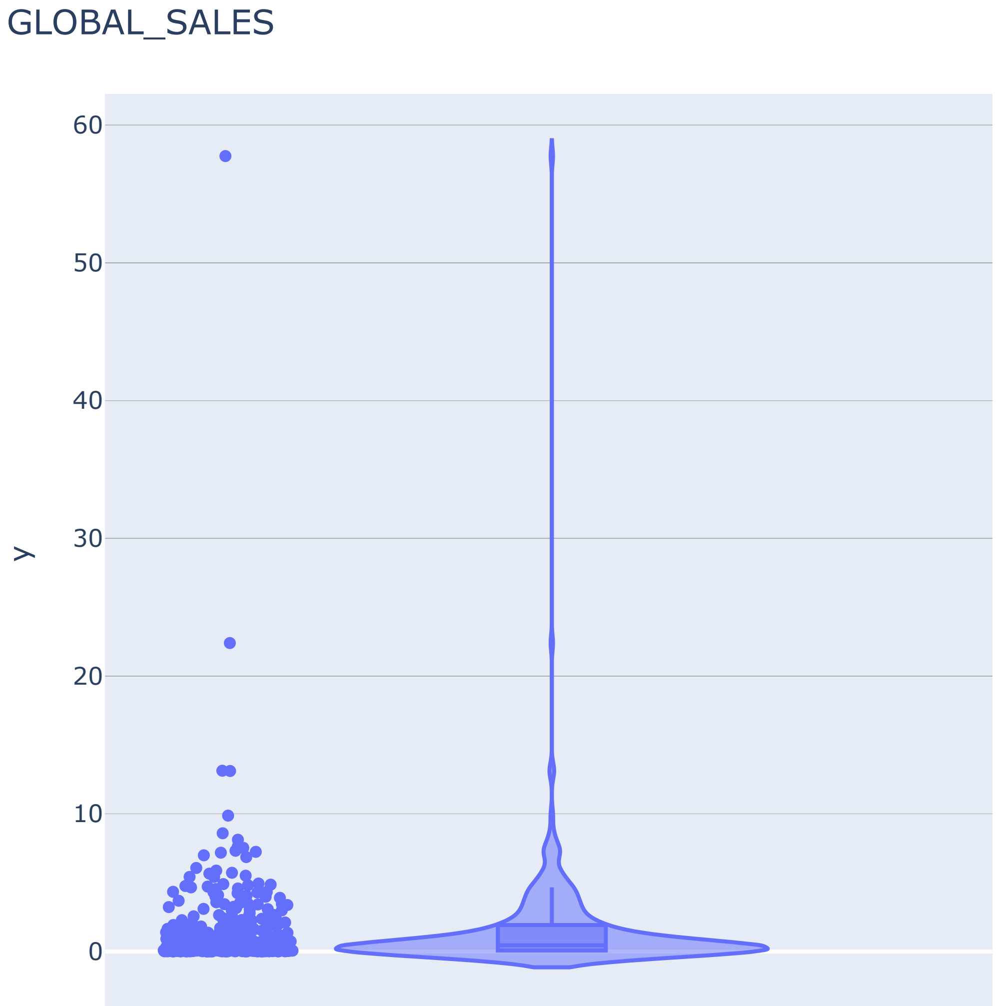

NA_SALES
25
20
15
y
10
5
0

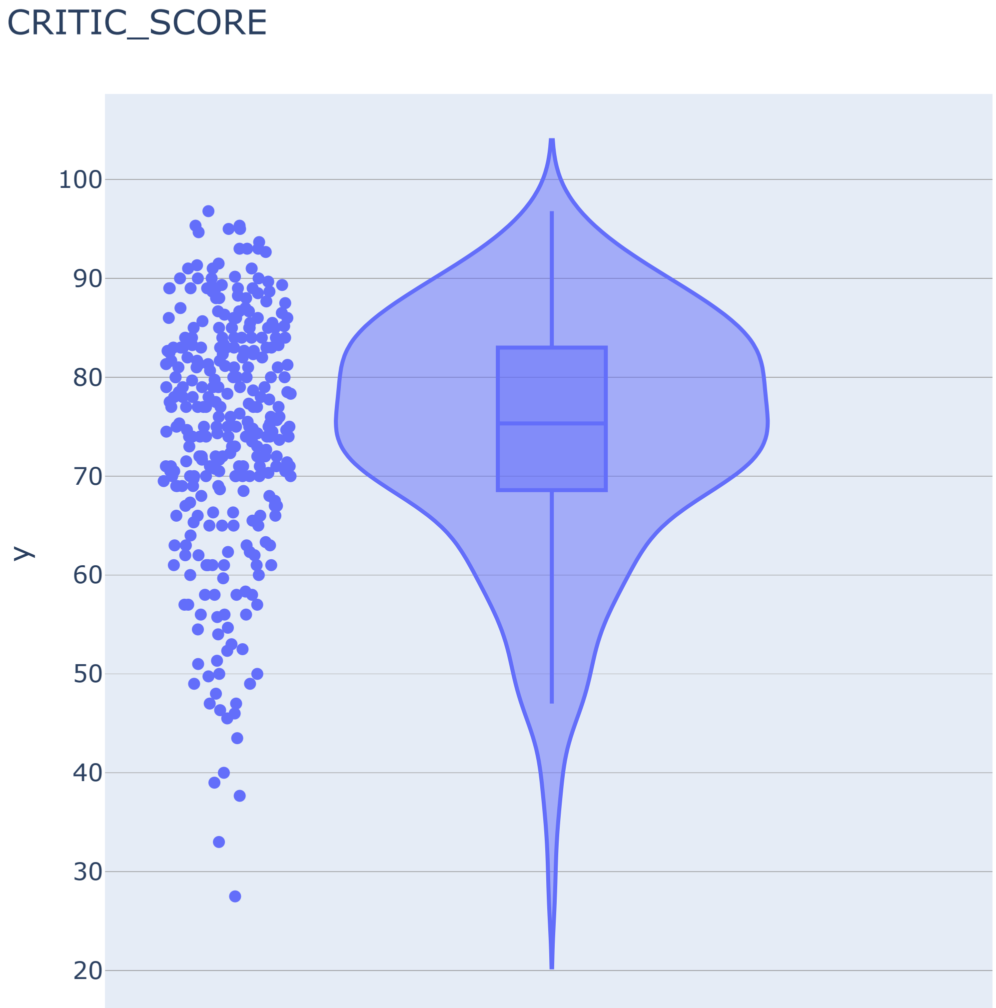

EU_SALES
y
25
20
15
10
5
0

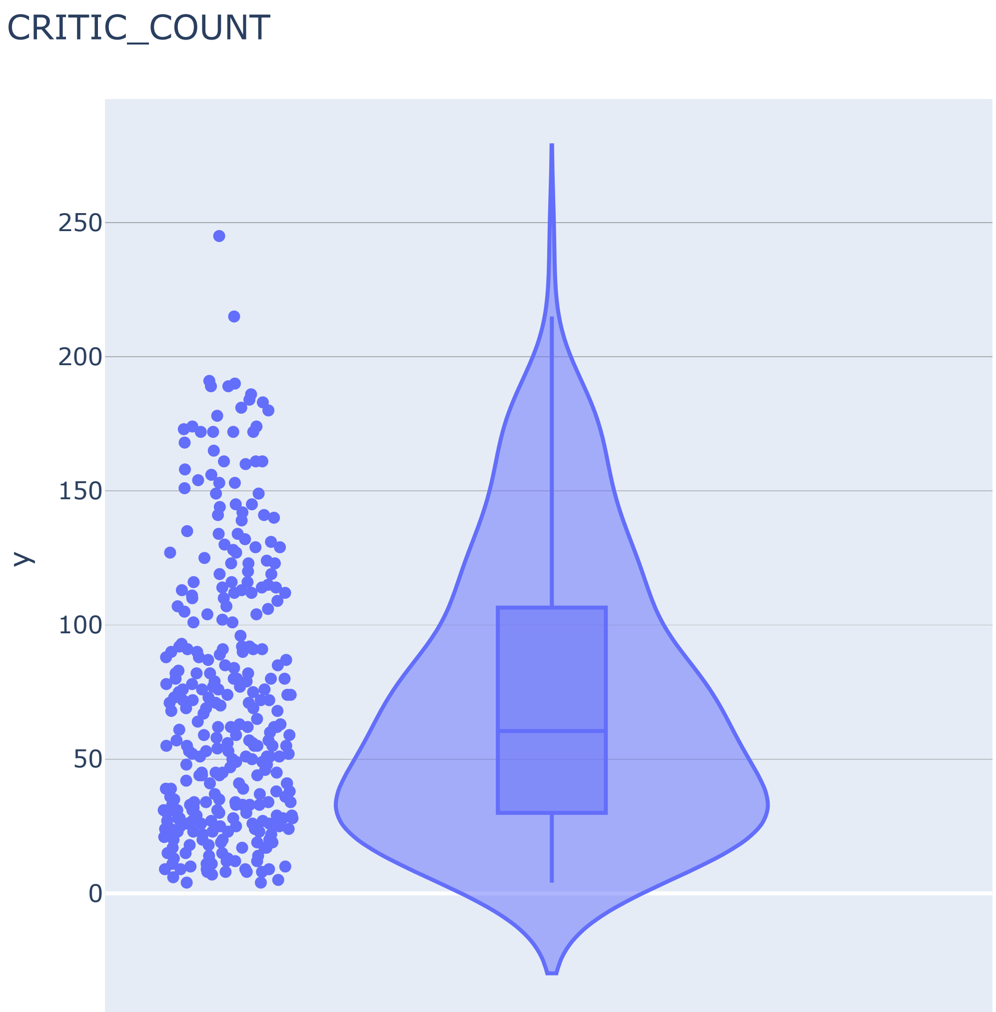
JP_SALES

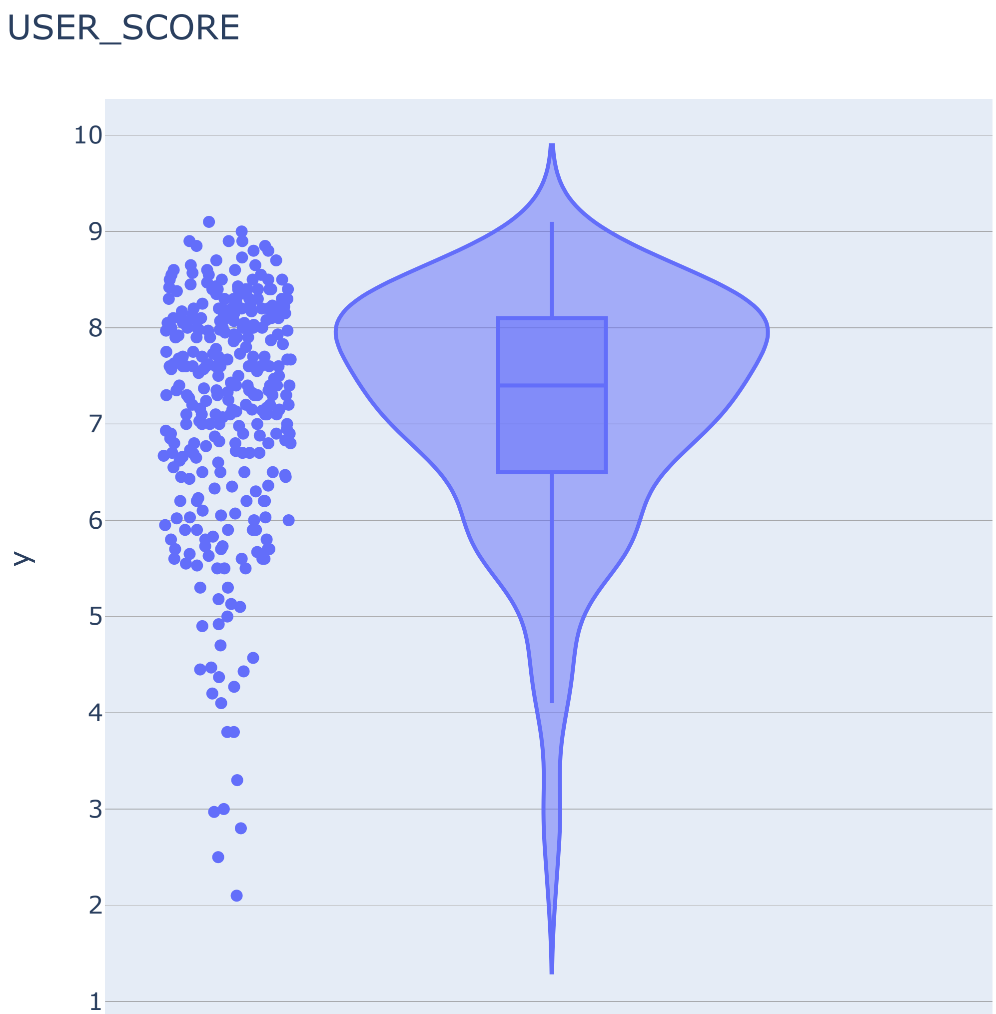



OTHER_SALES
y
8
7
6
5
4
3
2
1
0

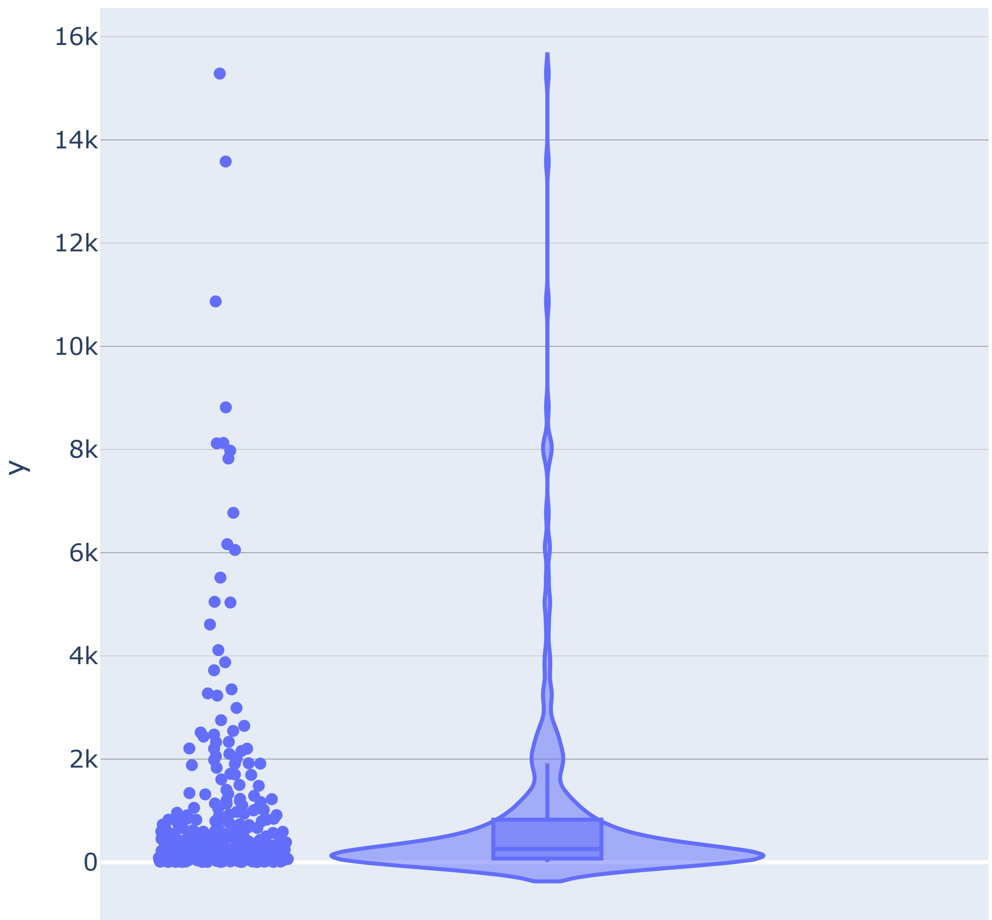

GLOBAL_SALES
60
50
40
30
20
10
0
y

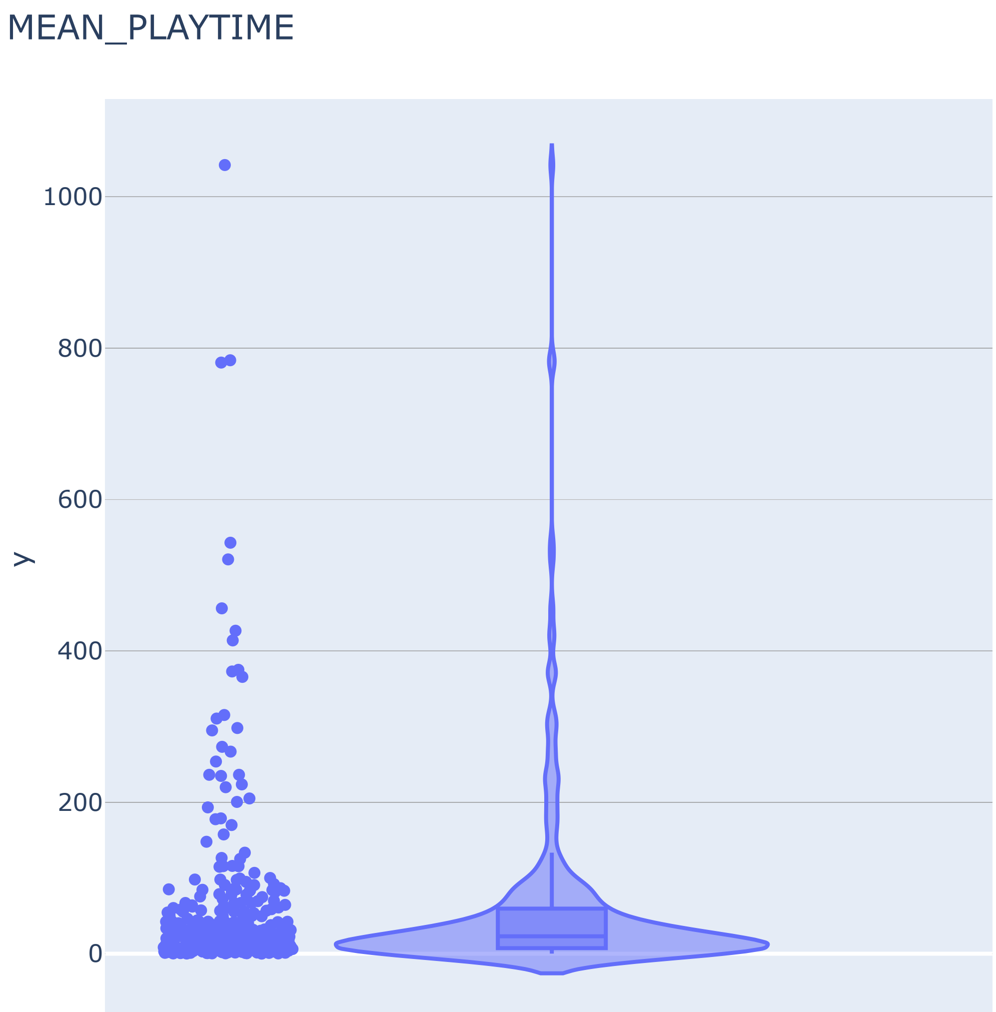

CRITIC_SCORE
y
100
90
80
70
60
50
40
30
20

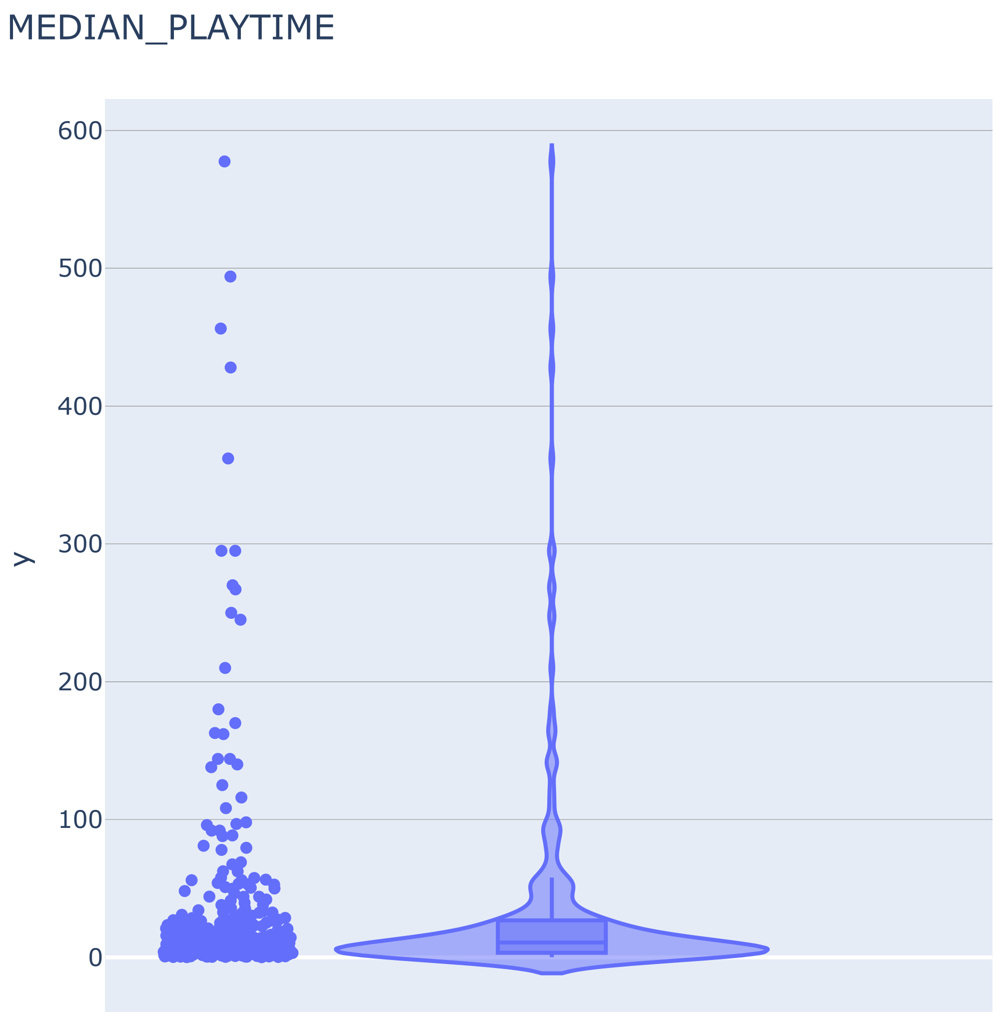

CRITIC_COUNT
250
200
150
100
50
0
y

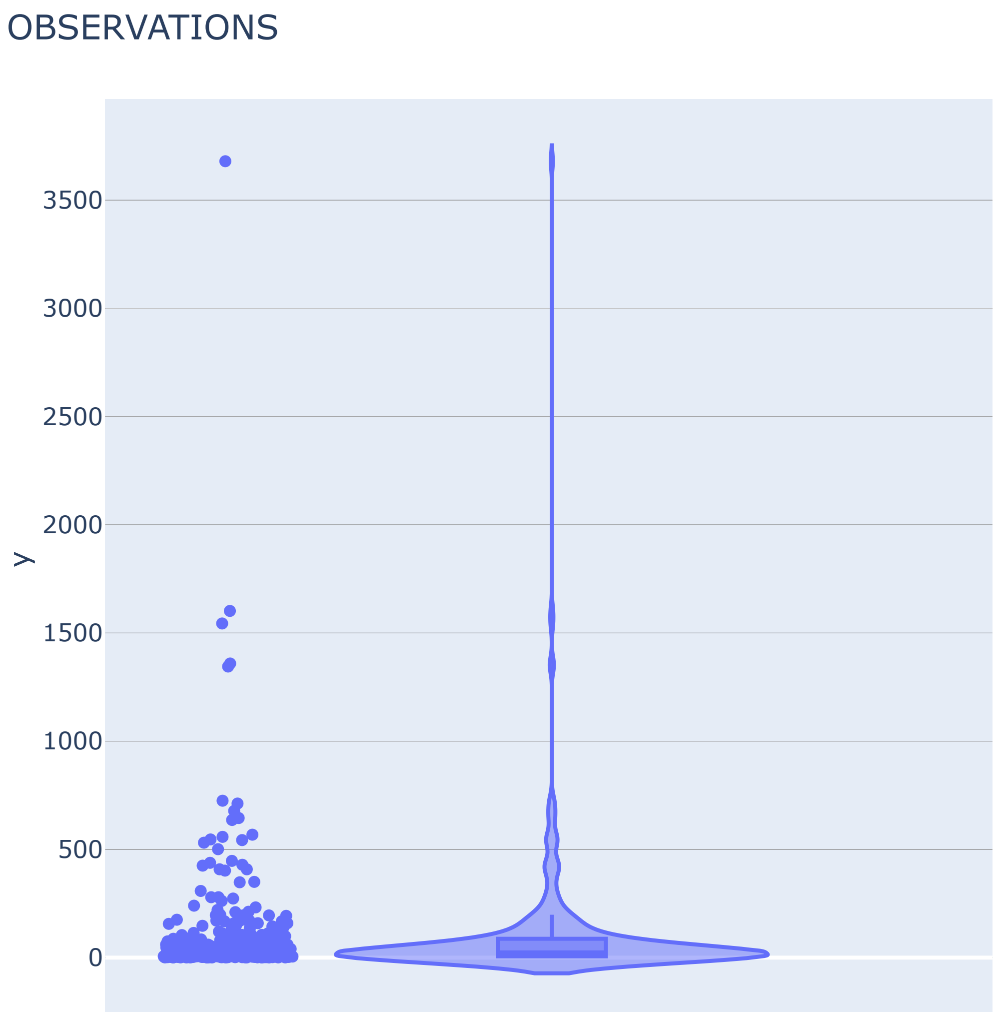

USER_SCORE
y
10
9
8
7
6
5
4
3
2
1

USER_COUNT




MEAN_PLAYTIME
y
1000
800
600
400
200
0


MEDIAN_PLAYTIME
600
500
400
300
200
100
0
y

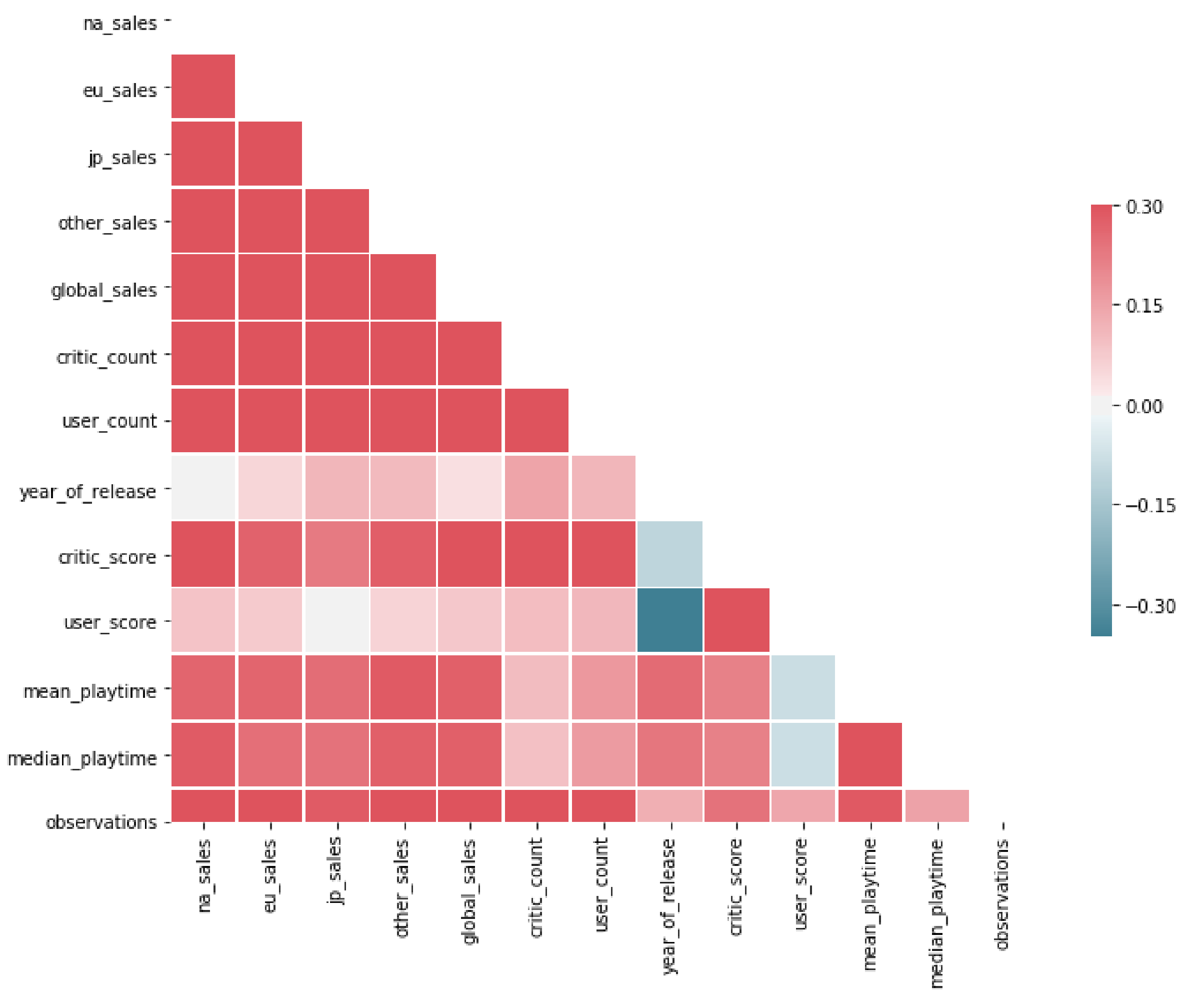


Multi-Dimensional Exploration

Now it's time to look at the multi-dimensional exploration, specifically the correlations between the different variables.

In [14]:
```
# Removing the index from the correlations
gamedf_no_na = gamedf_no_na.drop('index', 1)

corr = gamedf_no_na.corr()

mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

f, ax = plt.subplots(figsize=(11,9))
cmap = sns.diverging_palette(220,10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidth=.5, cbar_kws={'shrink':.5})
```

Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x1f505a82898>



Well, now, this is unexpected. Most of the variables are moderately positively correlated, with the exception of the `user_score`. This makes sense as 5 of the variables are sales counts. What I find interesting is that the `user_score` is so weakly correlated with most of the other variables, especially the `mean_playtime` and `median_playtime`. I thought that the higher the `user_score`, the more time would be spent playing the game. But what about the p-value? Are these correlations statistically significant?

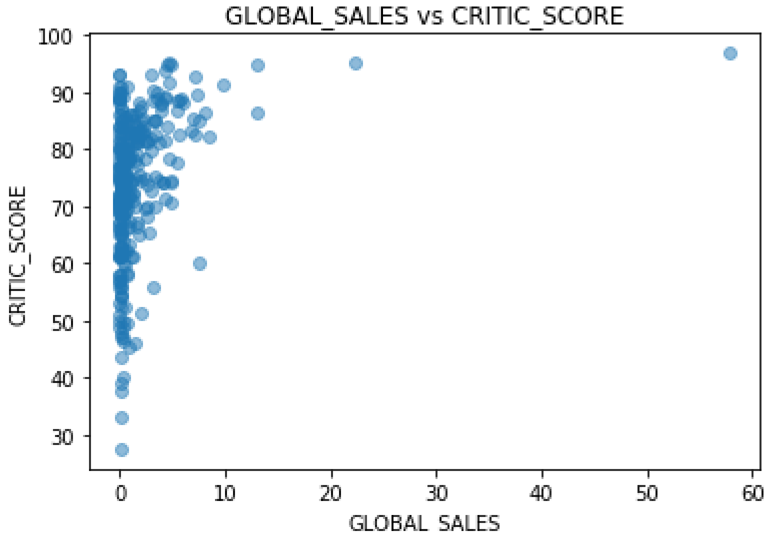
In [15]:
```
print('user_score vs mean_playtime, (correlation, p-value)\n',
      stats.pearsonr(gamedf_no_na['user_score'], gamedf_no_na['mean_playtime'
]),
     '\n\n')
print('user_score vs median_playtime, (correlation, p-value)\n',
     stats.pearsonr(gamedf_no_na['user_score'], gamedf_no_na['median_playtime'
]),
     '\n\n')
print('user_score vs year_of_release, (correlation, p-value)\n',
     stats.pearsonr(gamedf_no_na['user_score'], gamedf_no_na['year_of_release'
]),
     '\n\n')
print('user_score vs global_sales, (correlation, p-value)\n',
     stats.pearsonr(gamedf_no_na['user_score'], gamedf_no_na['global_sales']))
```

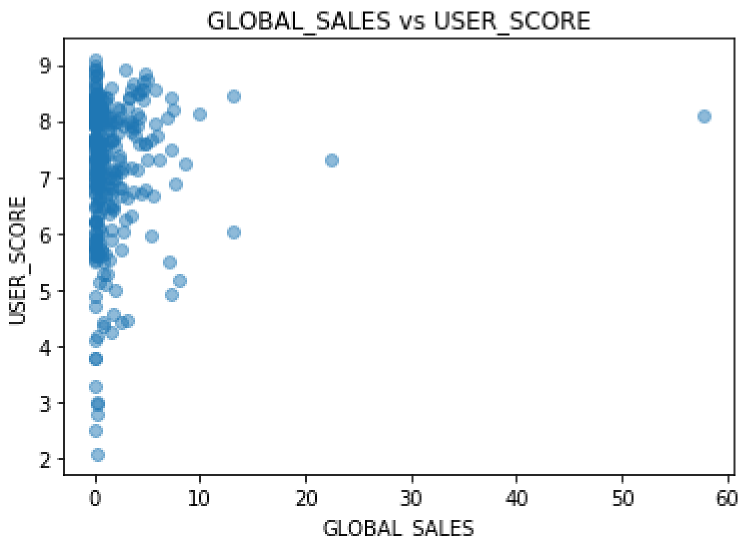
```
user_score vs mean_playtime, (correlation, p-value)
 (-0.08408617488446471, 0.13836104317449885)


user_score vs median_playtime, (correlation, p-value)
 (-0.08118995945990891, 0.1525141154601749)


user_score vs year_of_release, (correlation, p-value)
 (-0.3486499756859278, 2.396639057220405e-10)


user_score vs global_sales, (correlation, p-value)
 (0.07990376437372666, 0.15914025342052227)
```

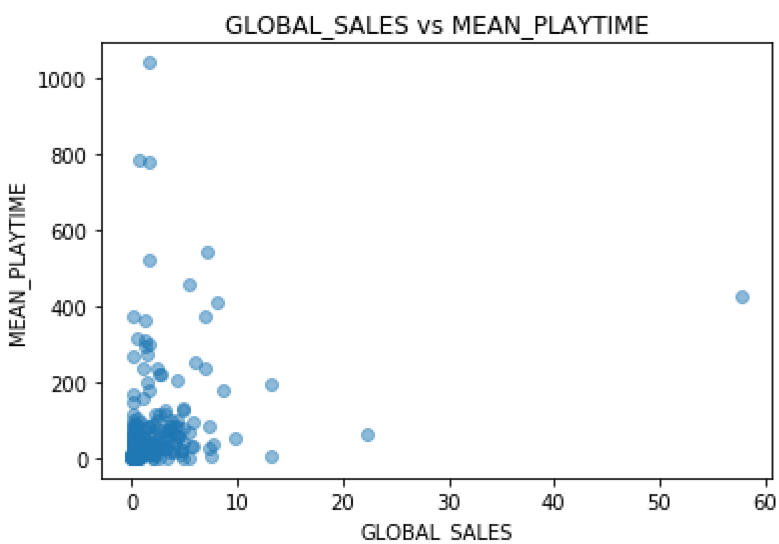
Interesting. None of these correlations are statistically significant. Let's move on to looking at some of the scatterplots.

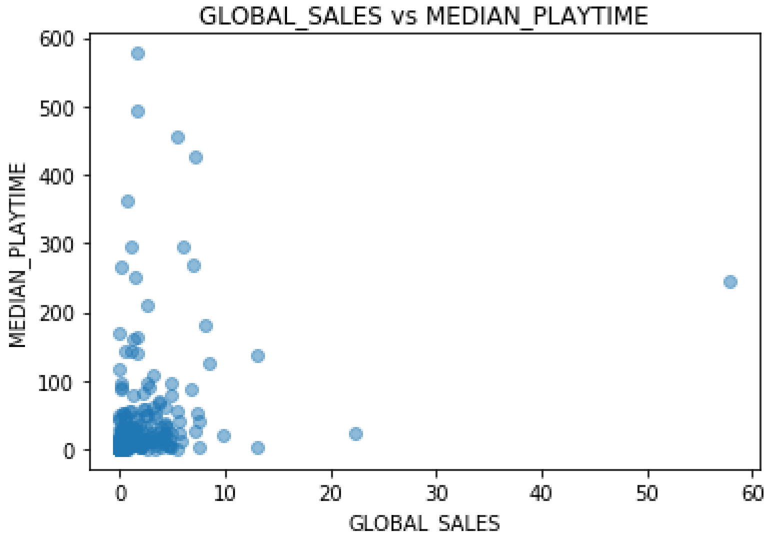
In [16]:
```
def make_scatter(var1, var2):
    plt.scatter(x=gamedf_no_na[var1], y=gamedf_no_na[var2], alpha=.5)
    plt.xlabel(var1.upper())
    plt.ylabel(var2.upper())
    plt.title('{} vs {}'.format(var1.upper(), var2.upper()))
    plt.show()
```

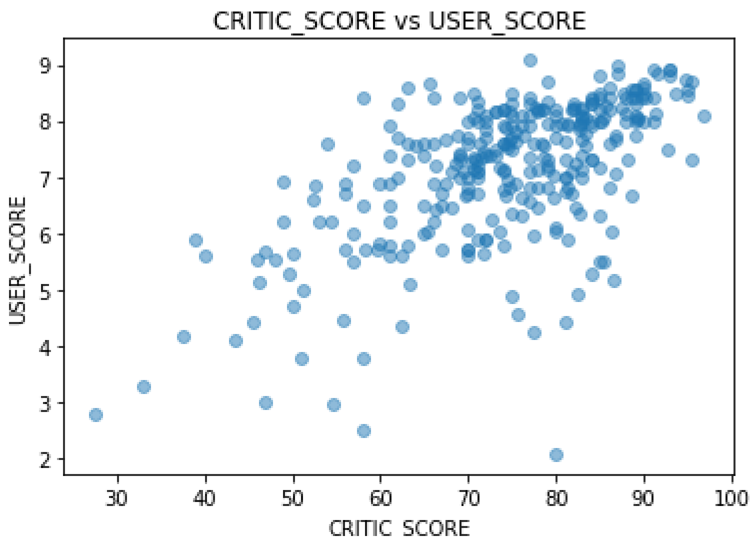
In [17]:
```
make_scatter('global_sales','critic_score')
make_scatter('global_sales', 'user_score')
make_scatter('global_sales', 'mean_playtime')
make_scatter('global_sales', 'median_playtime')
make_scatter('critic_score', 'user_score')
make_scatter('critic_score', 'mean_playtime')
make_scatter('critic_score', 'critic_count')
make_scatter('user_score', 'user_count')
make_scatter('user_score', 'mean_playtime')
make_scatter('user_score', 'median_playtime')
```

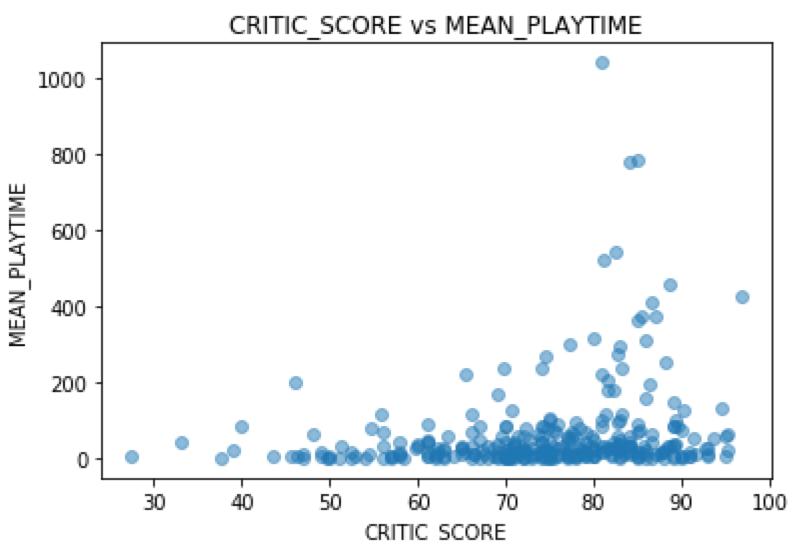

GLOBAL_SALES vs CRITIC_SCORE
CRITIC_SCORE
100
90
80
70
60
50
40
30
0
10
20
30
40
50
60
GLOBAL_SALES

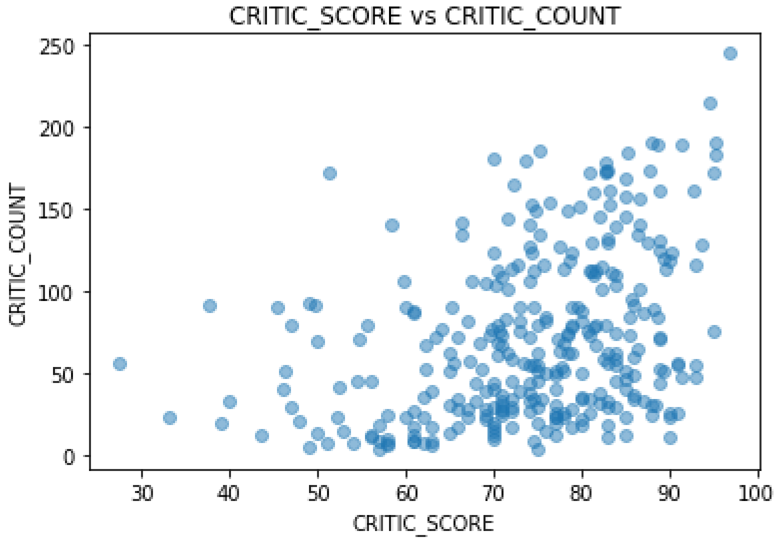

GLOBAL_SALES vs USER_SCORE
USER_SCORE
9
8
7
6
5
4
3
2
0
10
20
30
40
50
60
GLOBAL_SALES

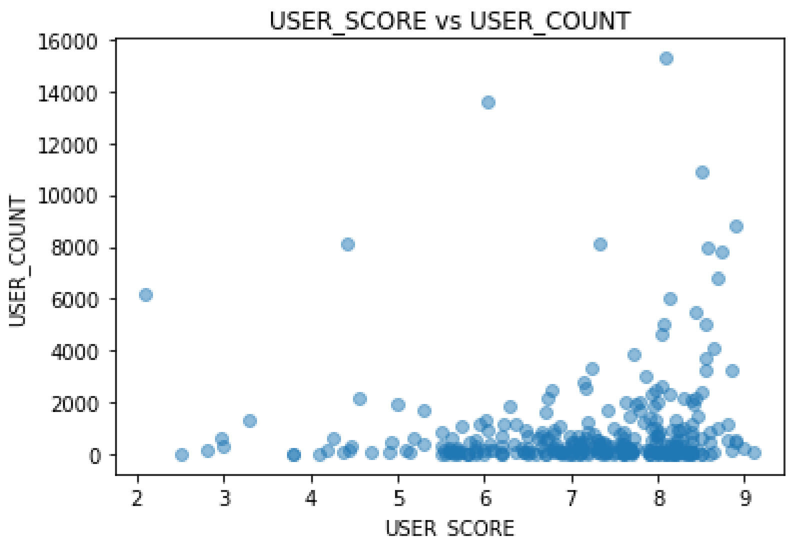

GLOBAL_SALES vs MEAN_PLAYTIME
MEAN_PLAYTIME
1000
800
600
400
200
0
0
10
20
30
40
50
60
GLOBAL_SALES

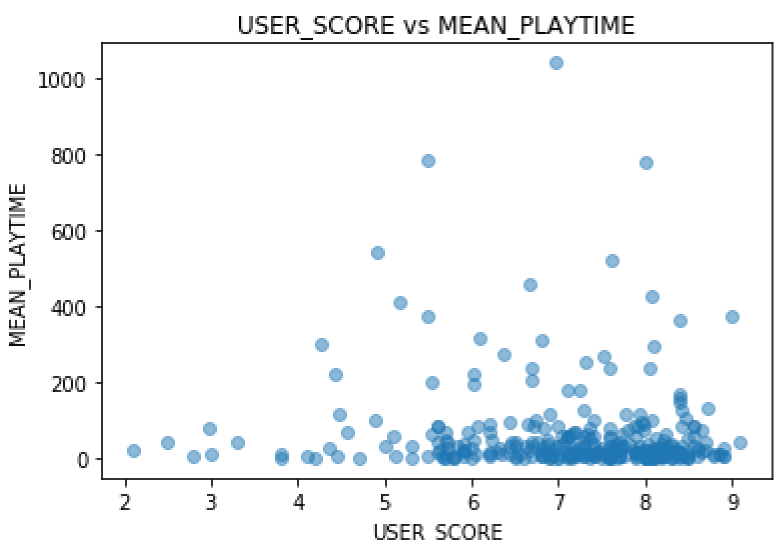

GLOBAL_SALES vs MEDIAN_PLAYTIME
MEDIAN_PLAYTIME
GLOBAL_SALES
0
100
200
300
400
500
600
0
10
20
30
40
50
60

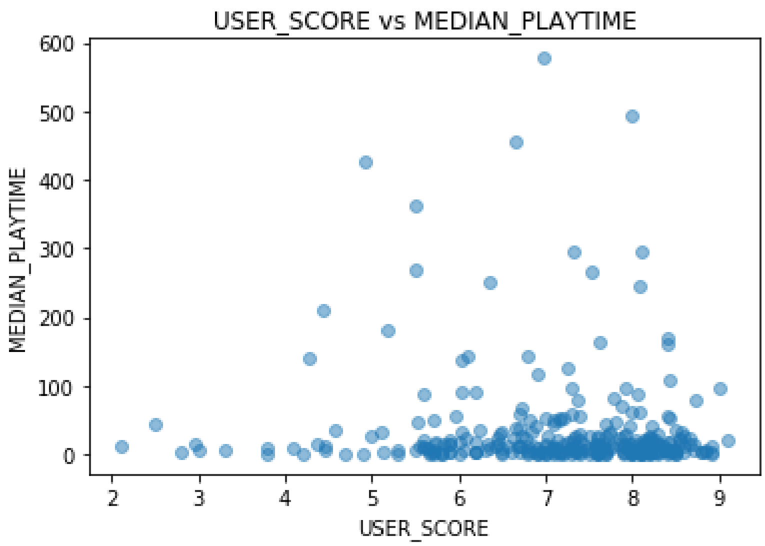

CRITIC_SCORE vs USER_SCORE
USER_SCORE
CRITIC_SCORE
2
3
4
5
6
7
8
9
30
40
50
60
70
80
90
100


CRITIC_SCORE vs MEAN_PLAYTIME
MEAN_PLAYTIME
CRITIC_SCORE
0
200
400
600
800
1000
30
40
50
60
70
80
90
100


CRITIC_SCORE vs CRITIC_COUNT
CRITIC_COUNT
CRITIC_SCORE
0
50
100
150
200
250
30
40
50
60
70
80
90
100


USER_SCORE vs USER_COUNT
USER_COUNT
USER_SCORE
0
2000
4000
6000
8000
10000
12000
14000
16000
2
3
4
5
6
7
8
9


USER_SCORE vs MEAN_PLAYTIME
MEAN_PLAYTIME
USER_SCORE
0
200
400
600
800
1000
2
3
4
5
6
7
8
9



In [18]:
```
print(gamedf_no_na.shape[0], '\n', gamedf.shape[0])
```

```
312
 12101
```

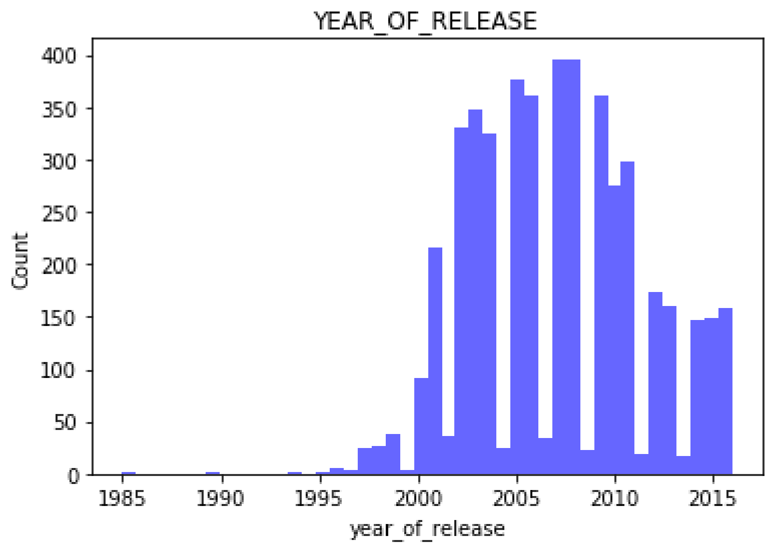
In [19]:
```
gamedf['user_score'].dropna()
```

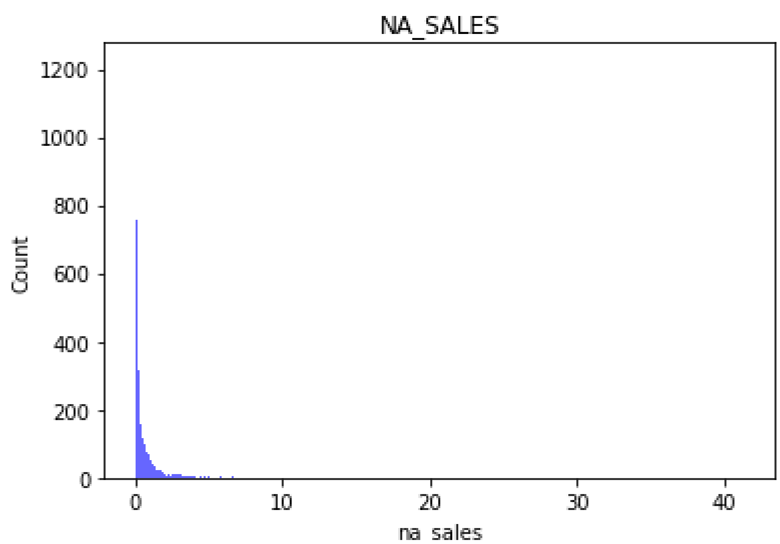
Out[19]:
```
4        8.50
6        8.90
7        8.70
11       4.60
12       6.88
         ...
12065    8.40
12066    7.90
12067    7.00
12068    6.12
12078    5.70
Name: user_score, Length: 4825, dtype: float64
```

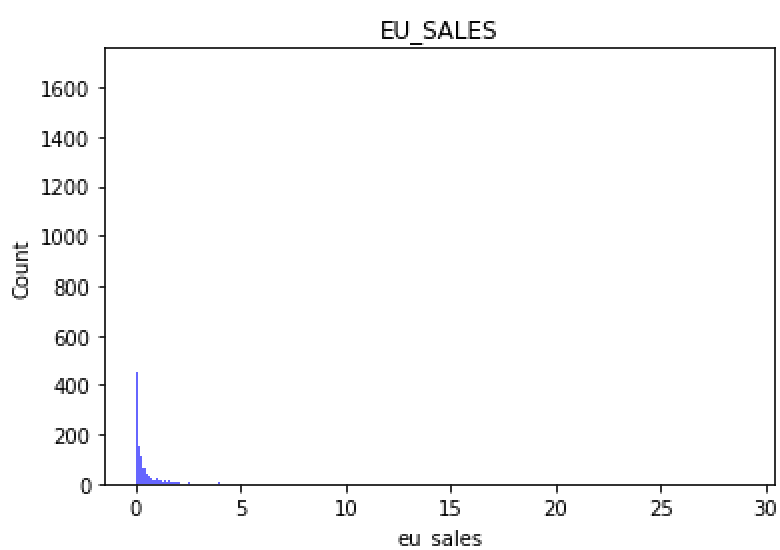
Hmmmm, so this is good, but it only looks at the data that has all data, which is only 312 records. What about the reemaining games that have scores, but are missing data?

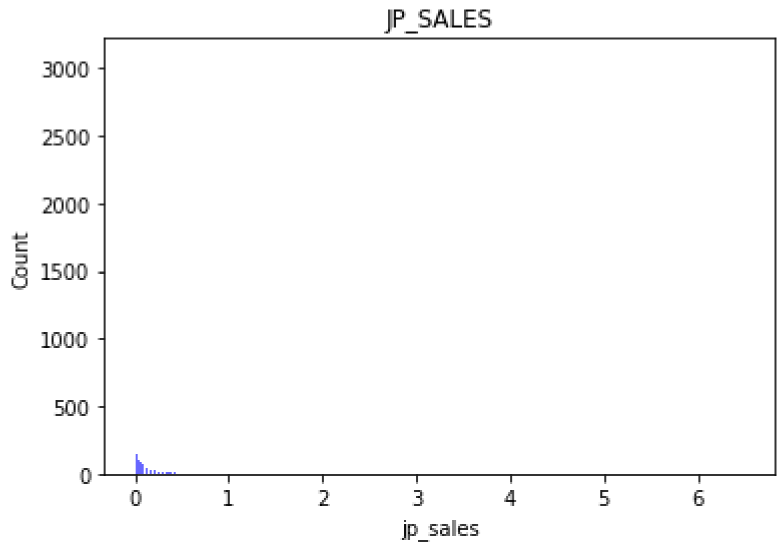
There are a total of 4,825 games that have a `user_score`. What if we took this subset and replaced the `nan` values with that column's mean score? Then used these values to build our predictions of the `user_score`?

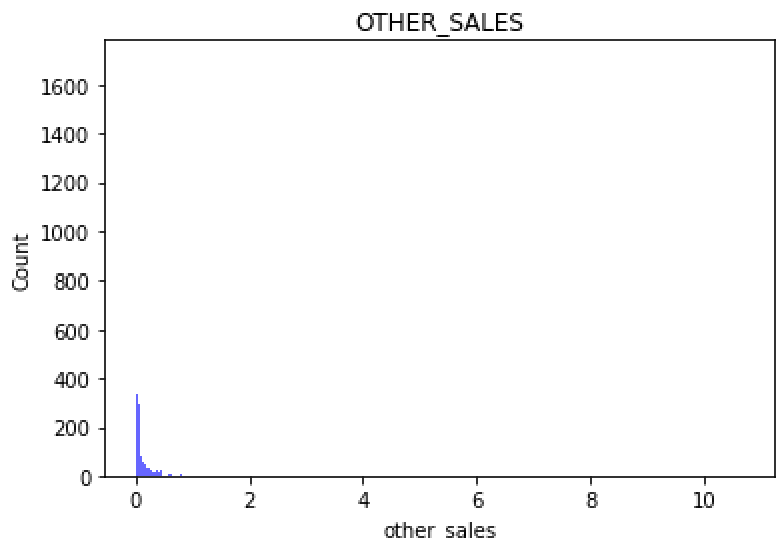
In [20]:
```
gamescores = gamedf.dropna(subset=['user_score'])
gamescores = gamescores.fillna(round(gamescores.mean(),2))

'''
This changes how our graphs will look. Let's take another quick look at the
histograms and the violin plots to see how they're different.
'''

def make_histogram(var, bins='auto'):
    n, bins, patches = plt.hist(x=gamescores[var], bins=bins, color='blue',
                                alpha=.6)
    plt.xlabel(var)
    plt.ylabel('Count')
    plt.title(var.upper())
    plt.show()

def make_violin(vari):
    tips = px.data.tips()
    fig = px.violin(tips, y=gamescores[vari], box=True, points='all')
    fig.update_layout(title_text=vari.upper())
    fig.show()


for name in vars:
    make_histogram(name)
for name in vars:
    make_violin(name)
```

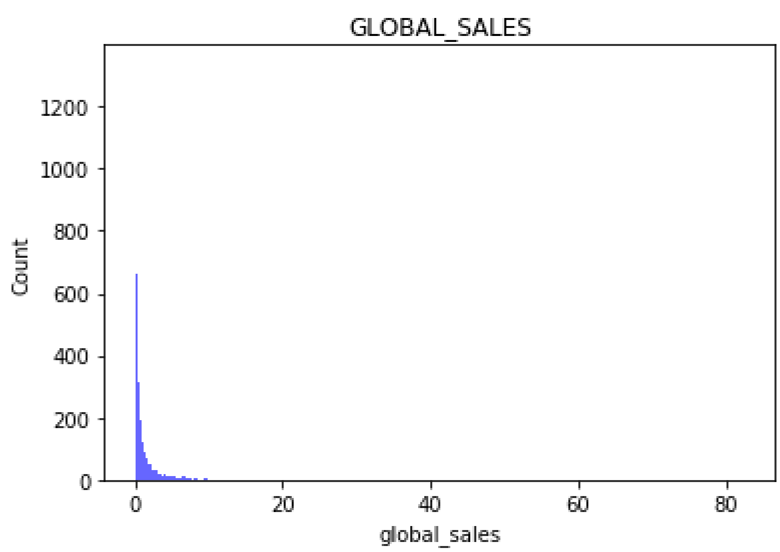

YEAR_OF_RELEASE
Count
year_of_release
0
50
100
150
200
250
300
350
400
1985
1990
1995
2000
2005
2010
2015

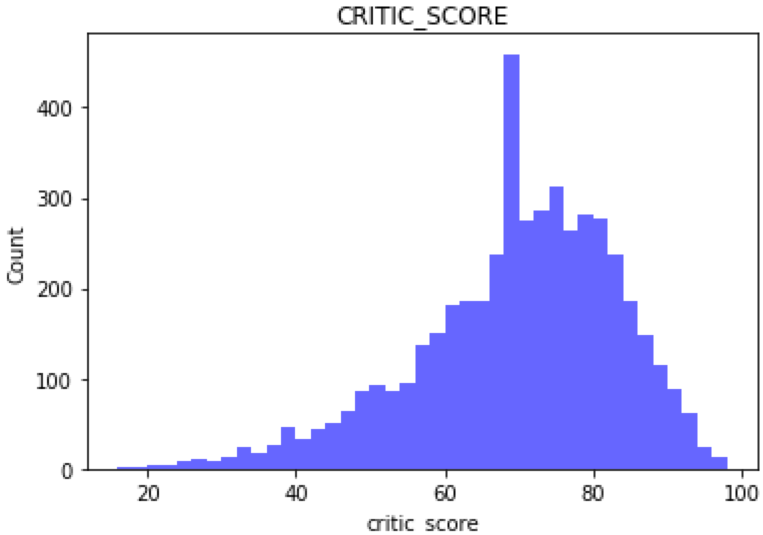

NA_SALES
Count
na_sales
0
200
400
600
800
1000
1200
0
10
20
30
40

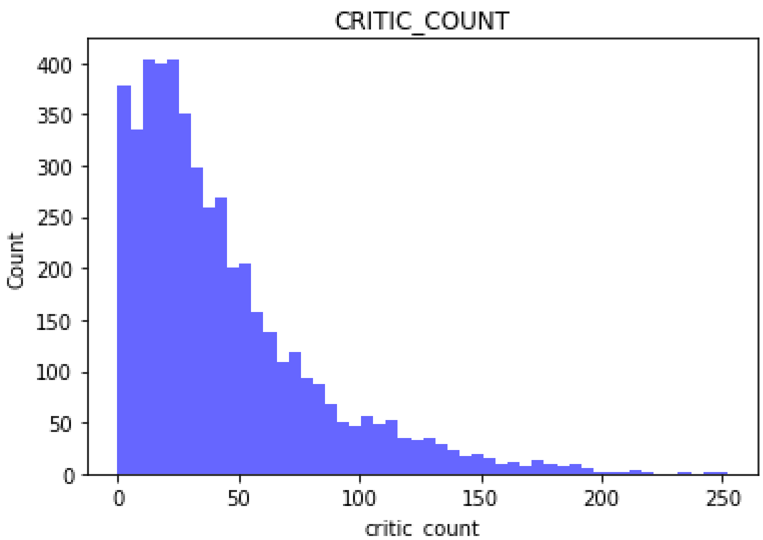

EU_SALES
Count
eu_sales
0
200
400
600
800
1000
1200
1400
1600
0
5
10
15
20
25
30

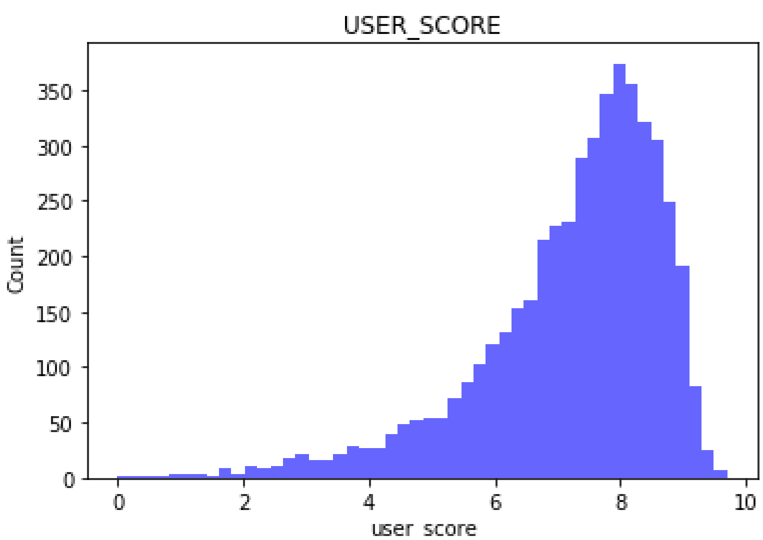

JP_SALES
Count
3000
2500
2000
1500
1000
500
0
0
1
2
3
4
5
6
jp_sales

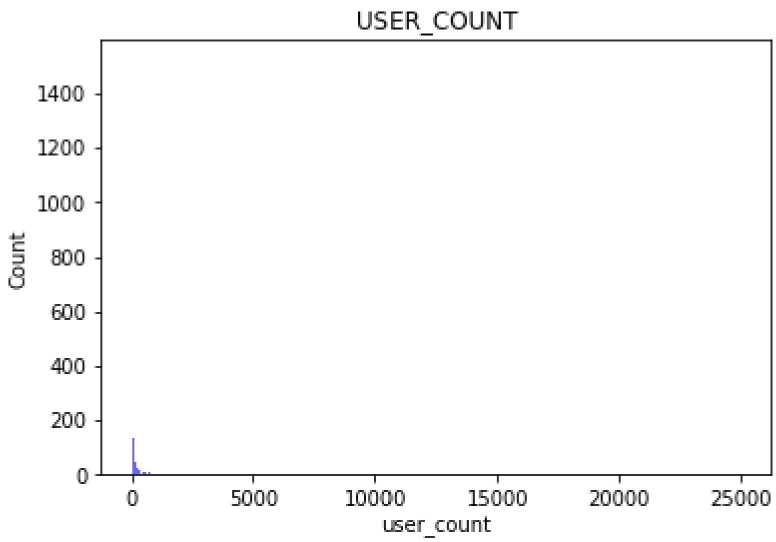

OTHER_SALES
Count
1600
1400
1200
1000
800
600
400
200
0
0
2
4
6
8
10
other_sales

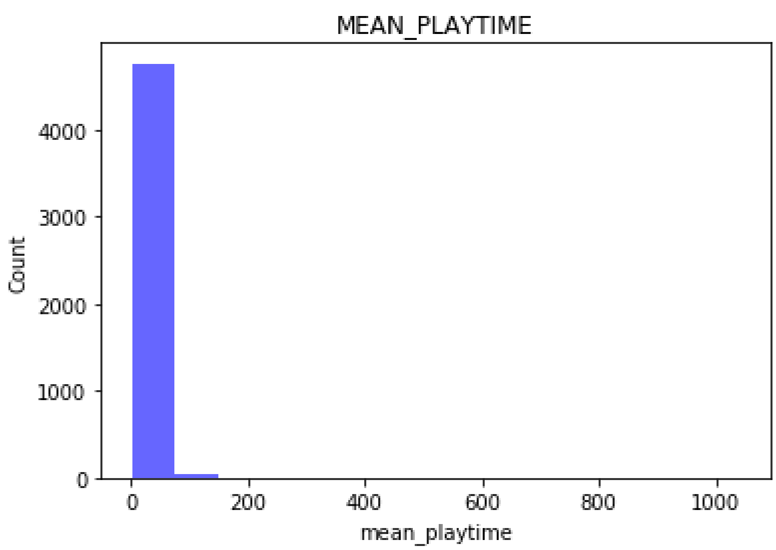

GLOBAL_SALES
Count
1200
1000
800
600
400
200
0
0
20
40
60
80
global_sales

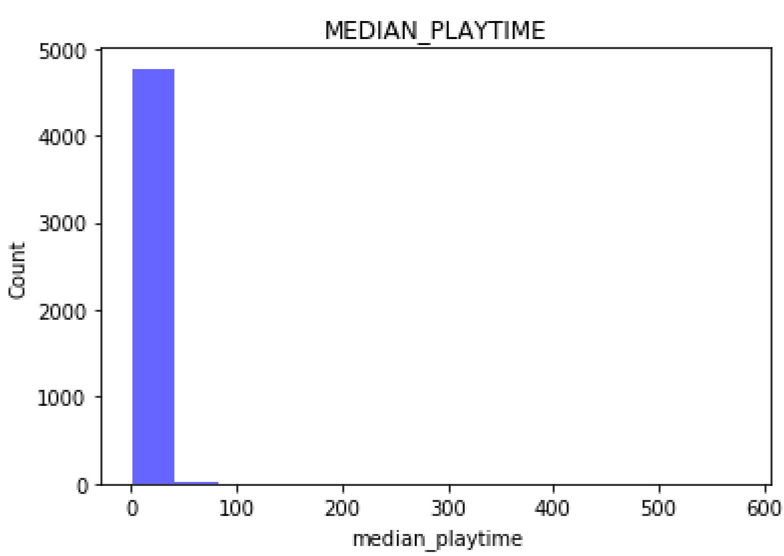

CRITIC_SCORE
Count
critic_score
0
100
200
300
400
20
40
60
80
100

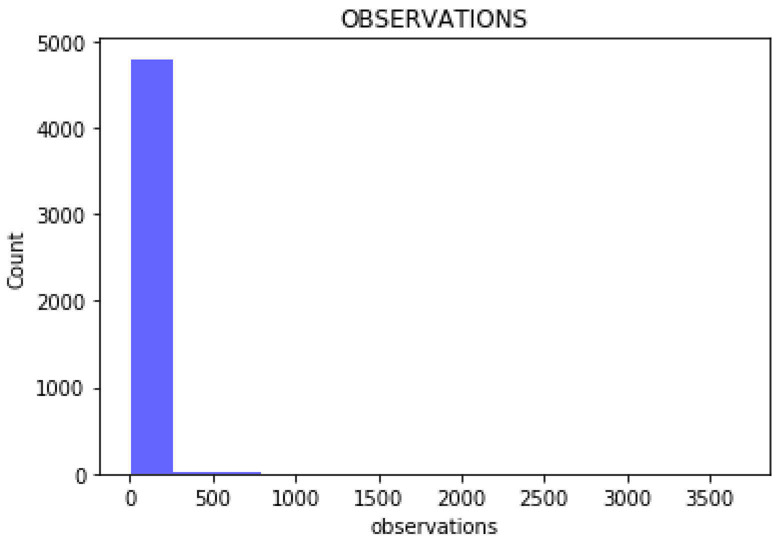

CRITIC_COUNT
Count
critic_count
0
50
100
150
200
250
300
350
400
0
50
100
150
200
250


USER_SCORE
Count
user_score
0
50
100
150
200
250
300
350
0
2
4
6
8
10

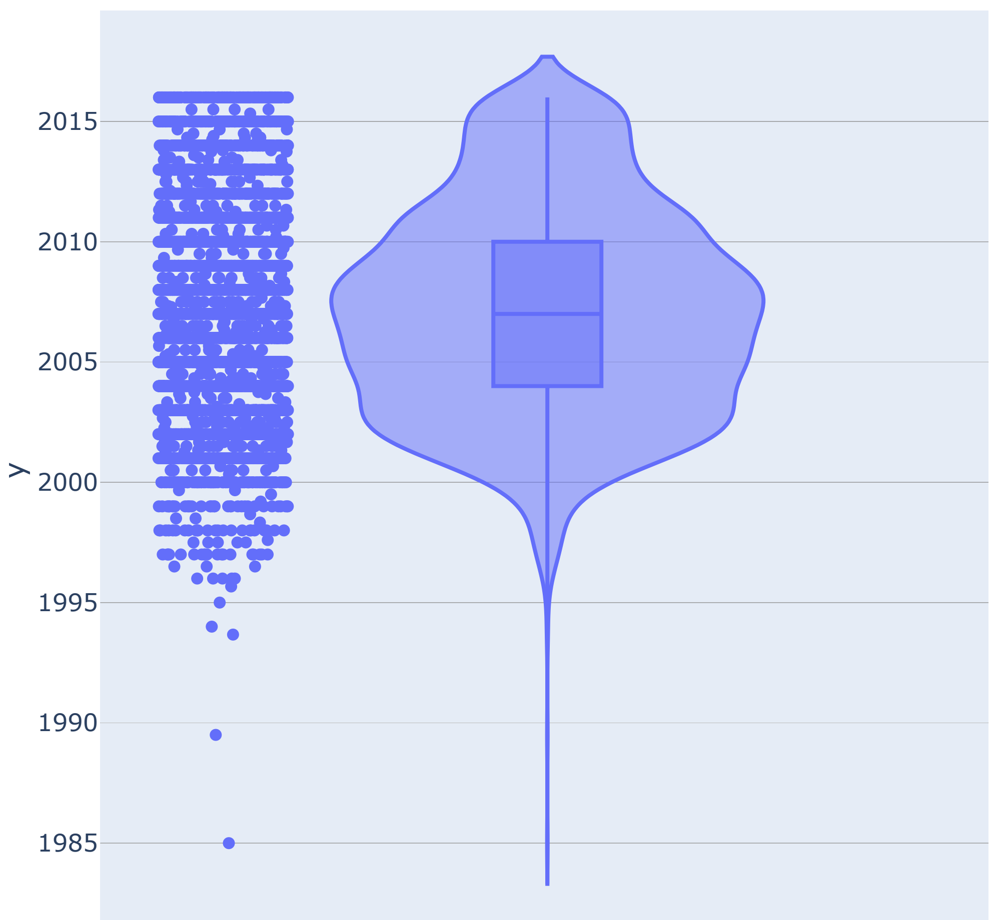

USER_COUNT
Count
user_count
0
200
400
600
800
1000
1200
1400
5000
10000
15000
20000
25000


MEAN_PLAYTIME
Count
mean_playtime
0
1000
2000
3000
4000
200
400
600
800
1000

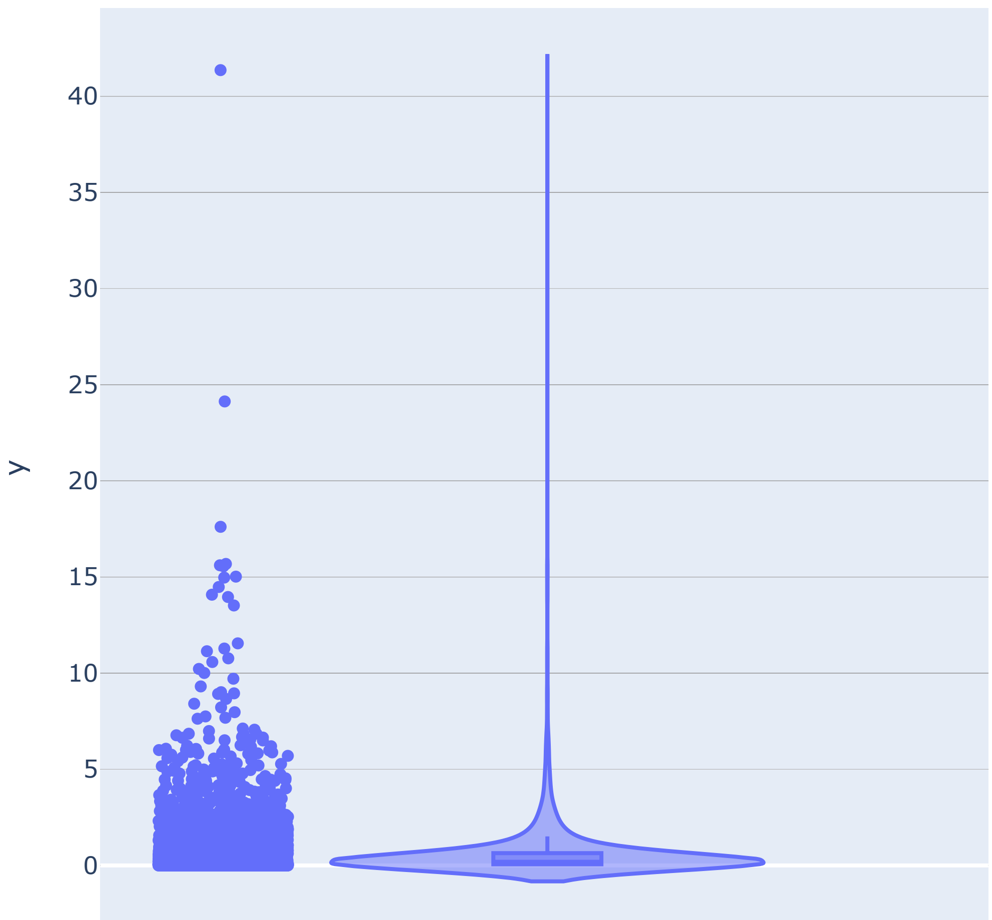

MEDIAN_PLAYTIME
Count
median_playtime
0
1000
2000
3000
4000
5000
100
200
300
400
500
600

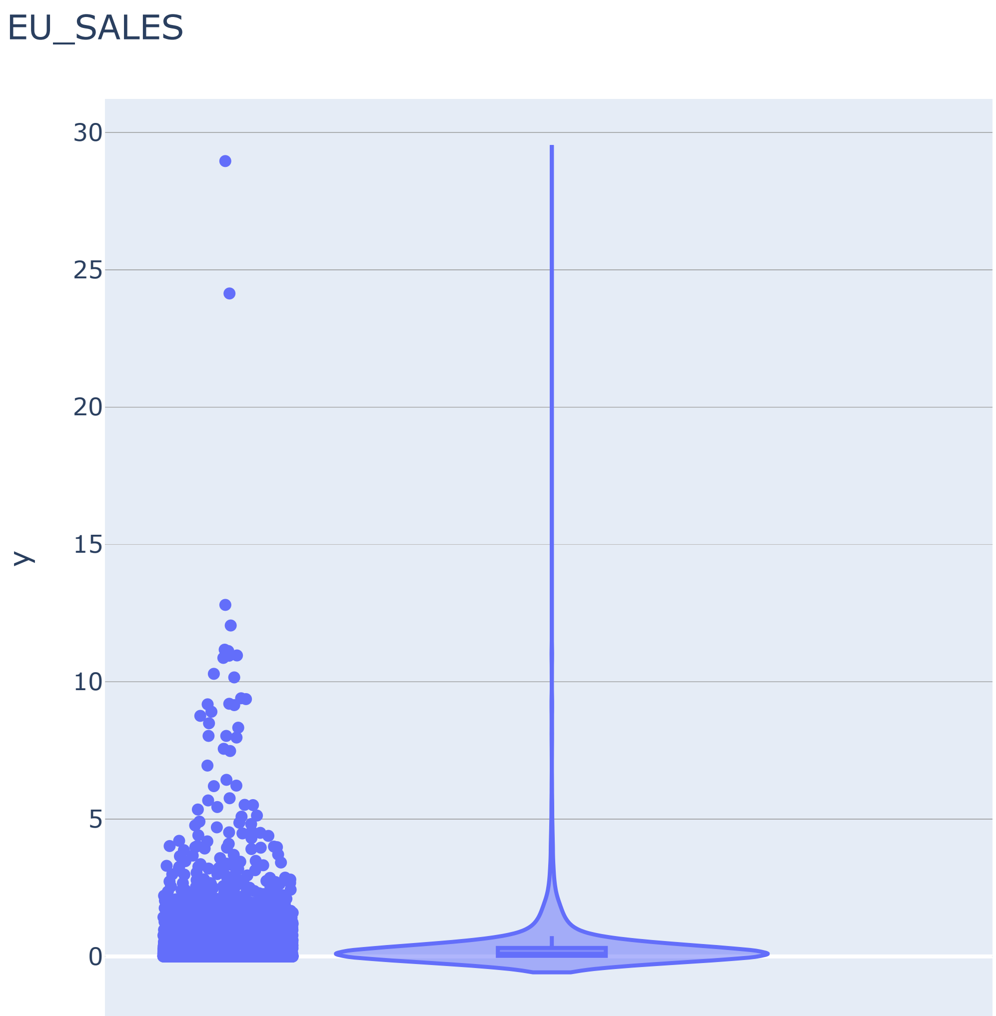


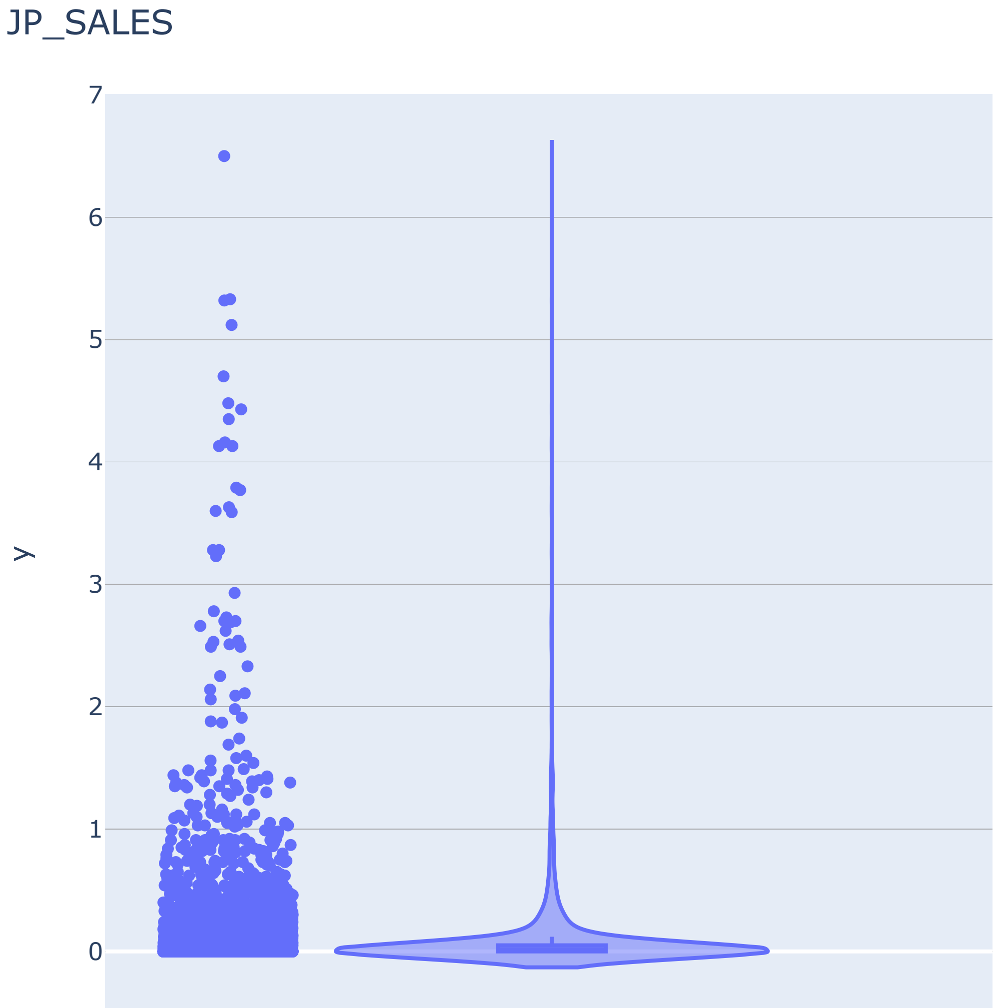
YEAR_OF_RELEASE

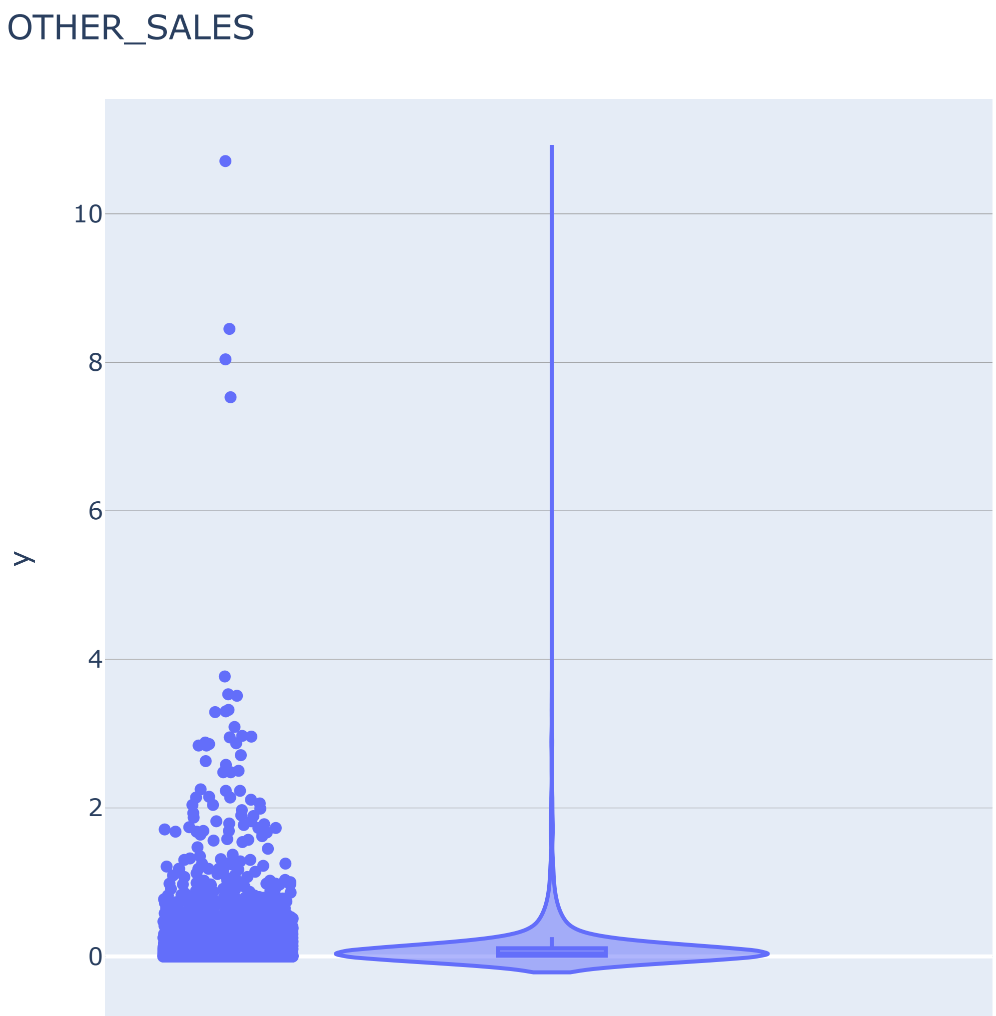


NA_SALES



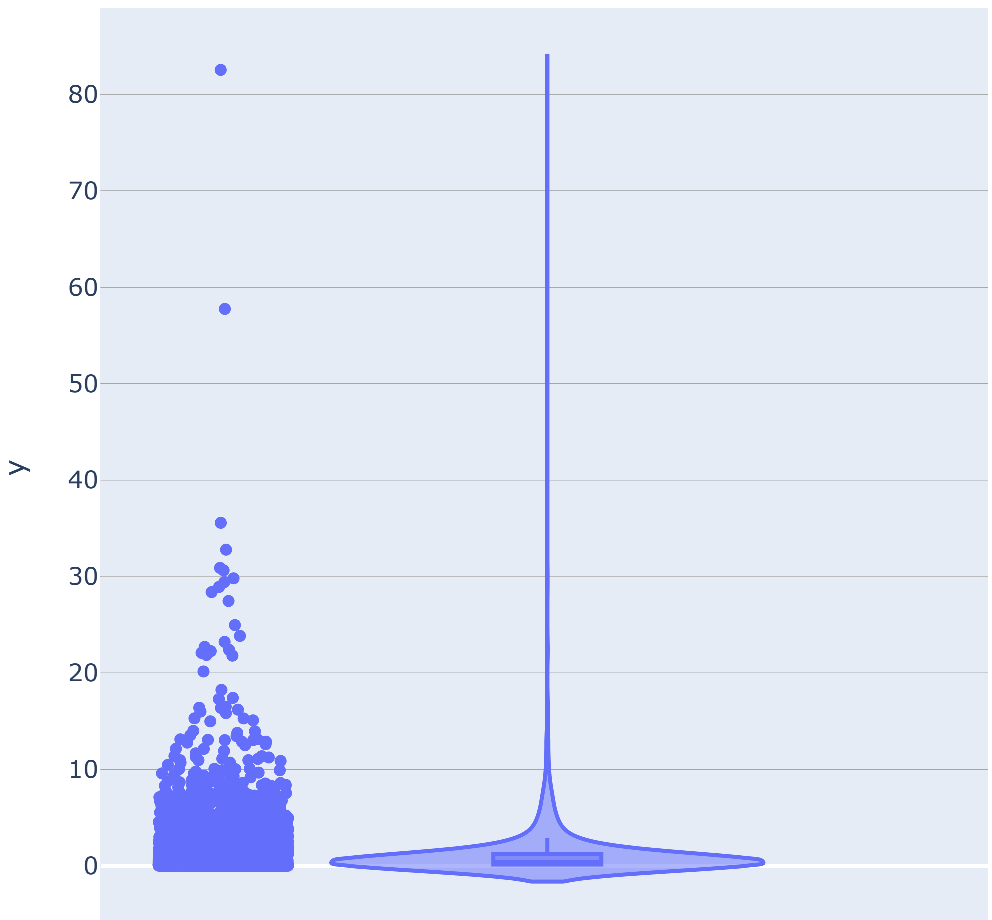

EU_SALES
y
30
25
20
15
10
5
0


JP_SALES
y
7
6
5
4
3
2
1
0

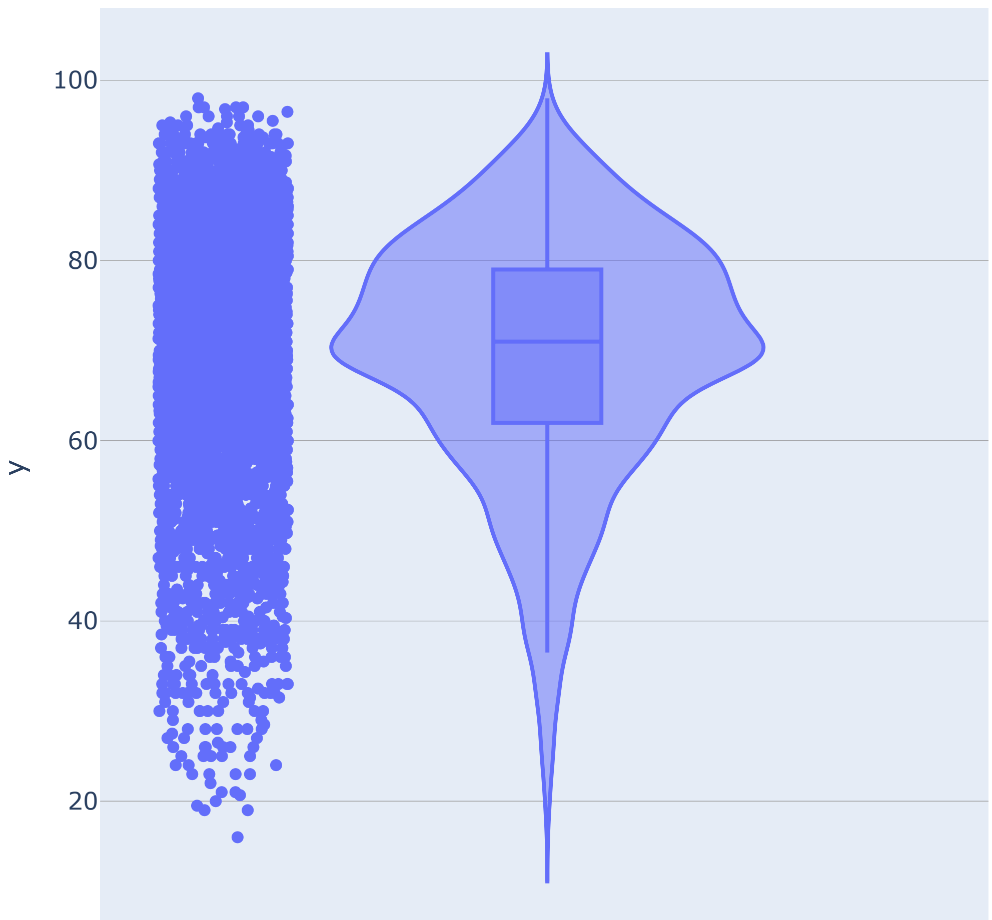


OTHER_SALES
y
10
8
6
4
2
0

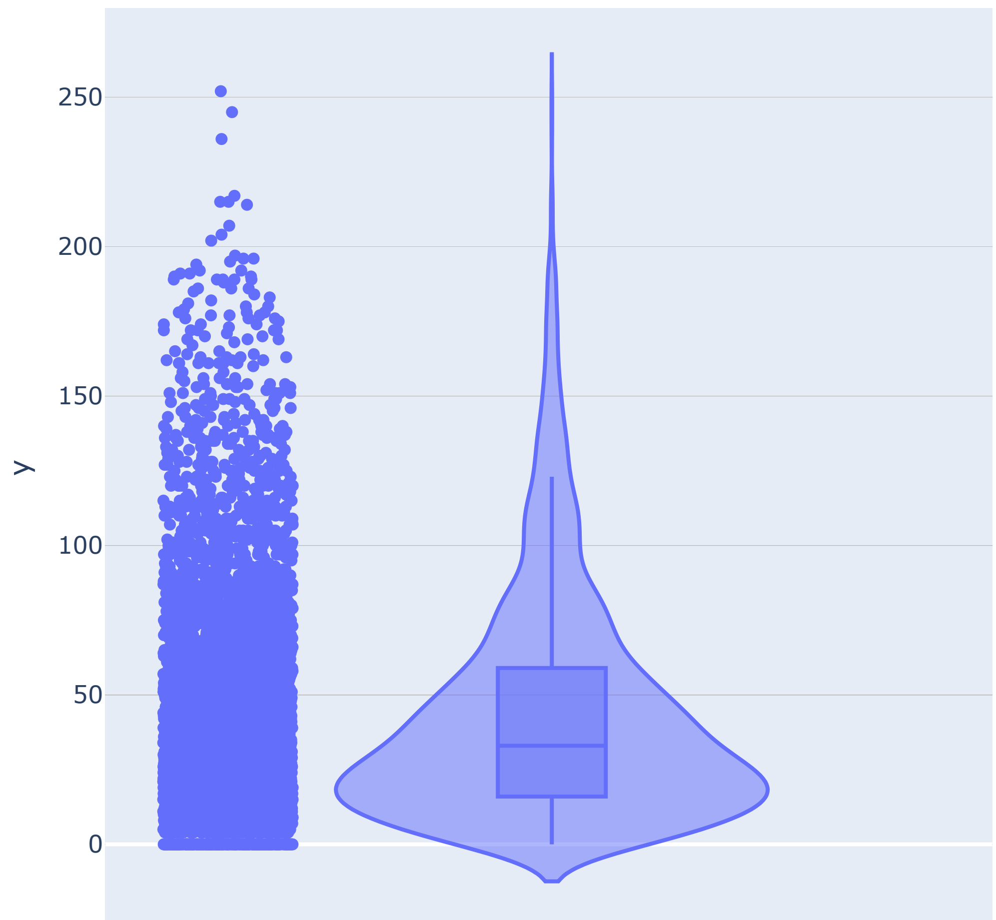
GLOBAL_SALES



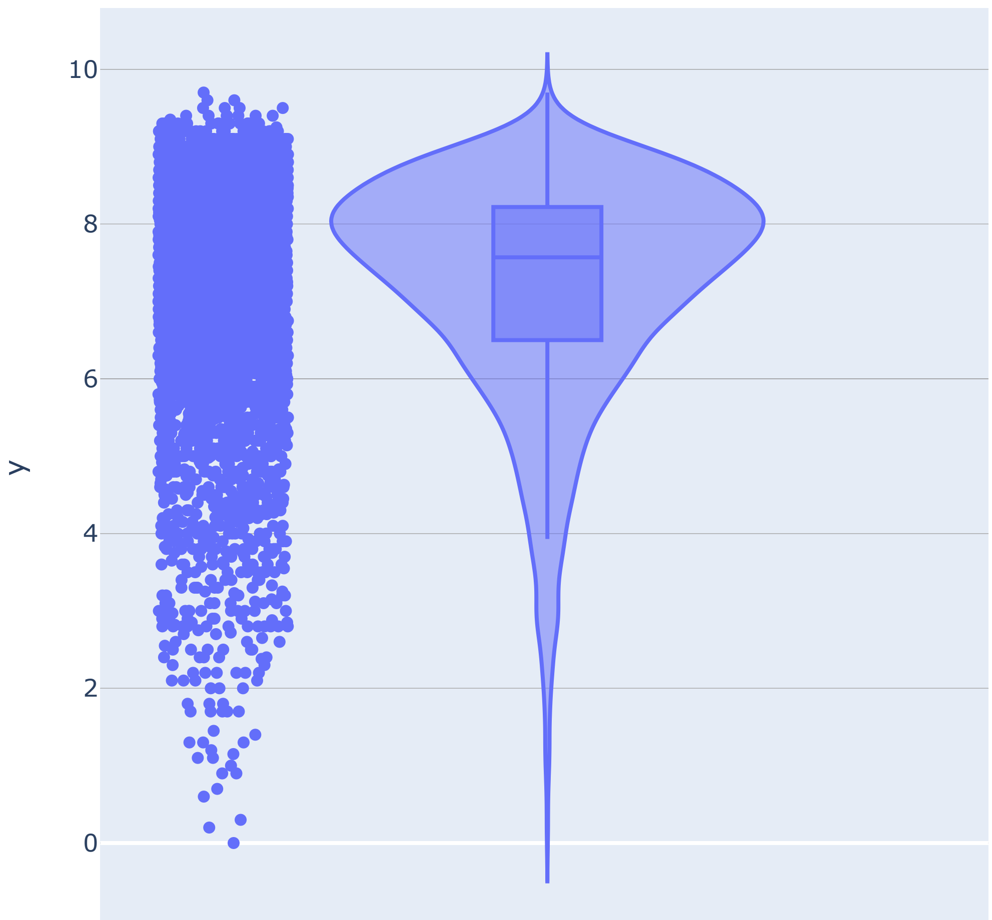
CRITIC_SCORE

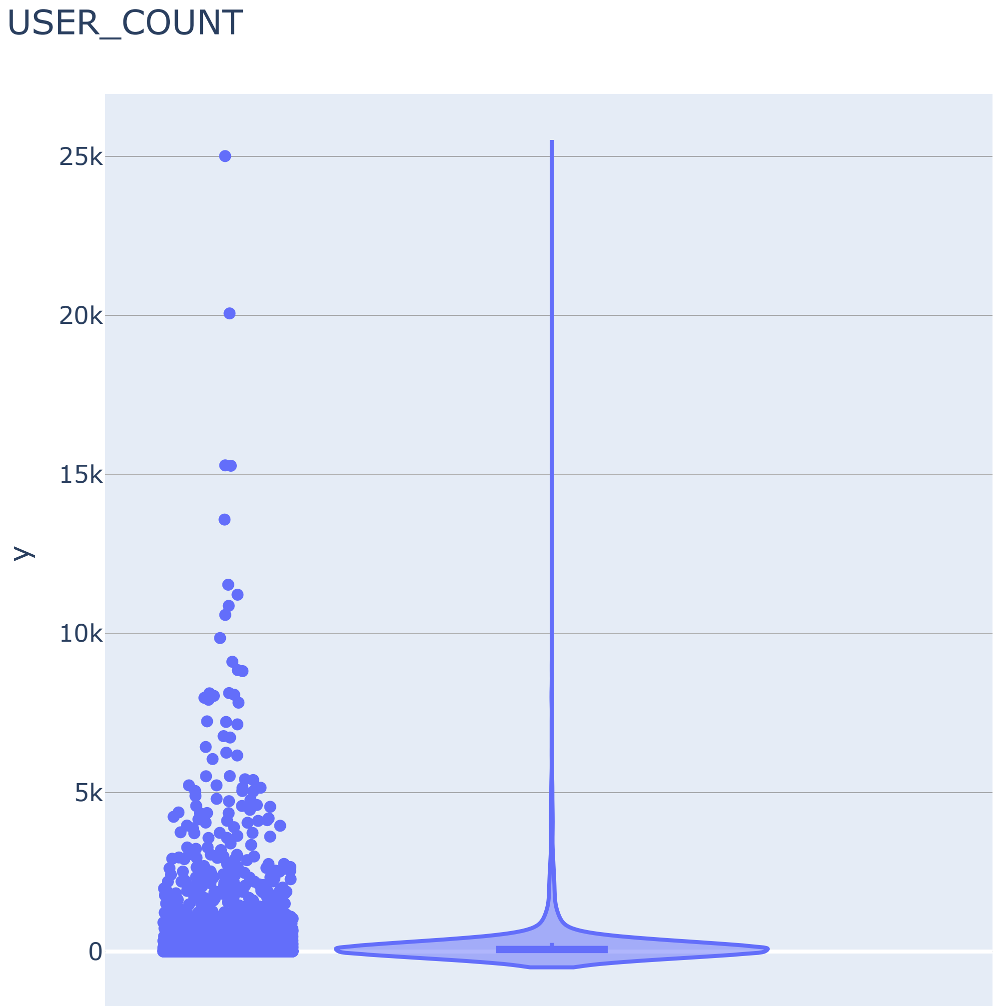


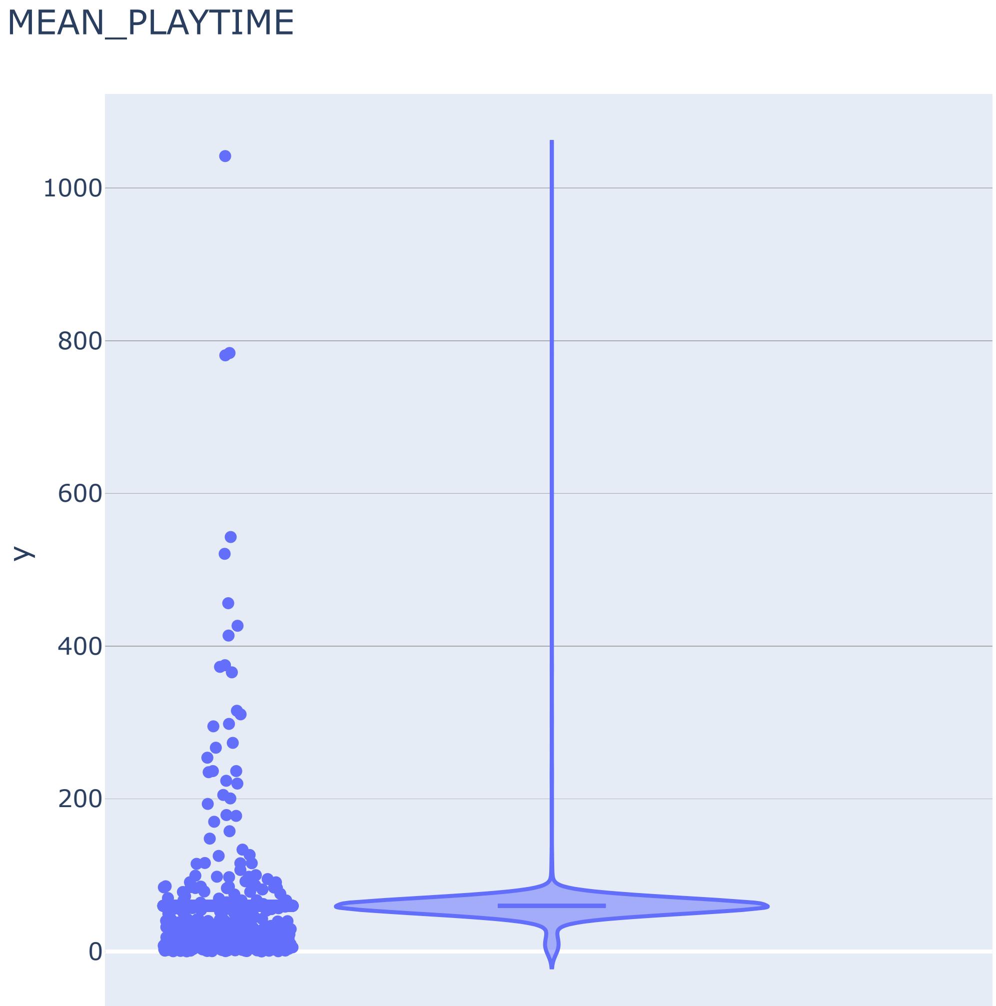
CRITIC_COUNT

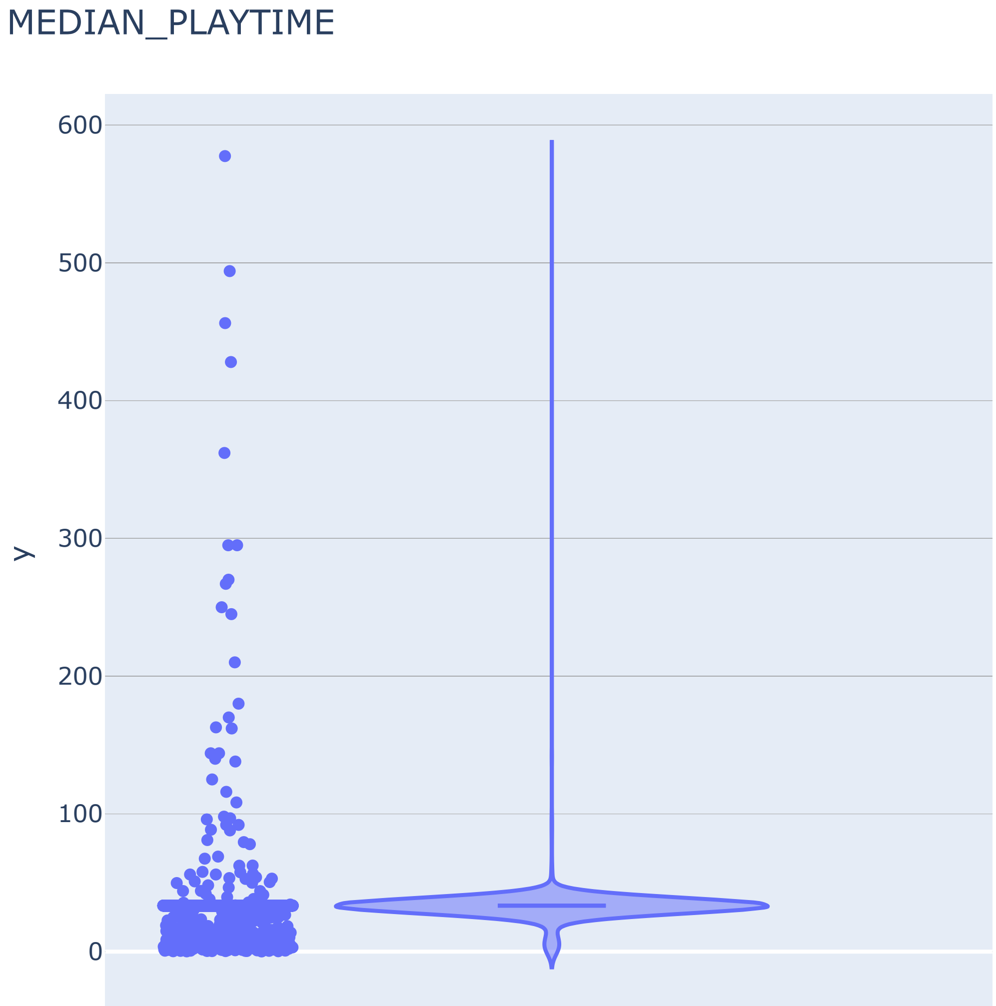


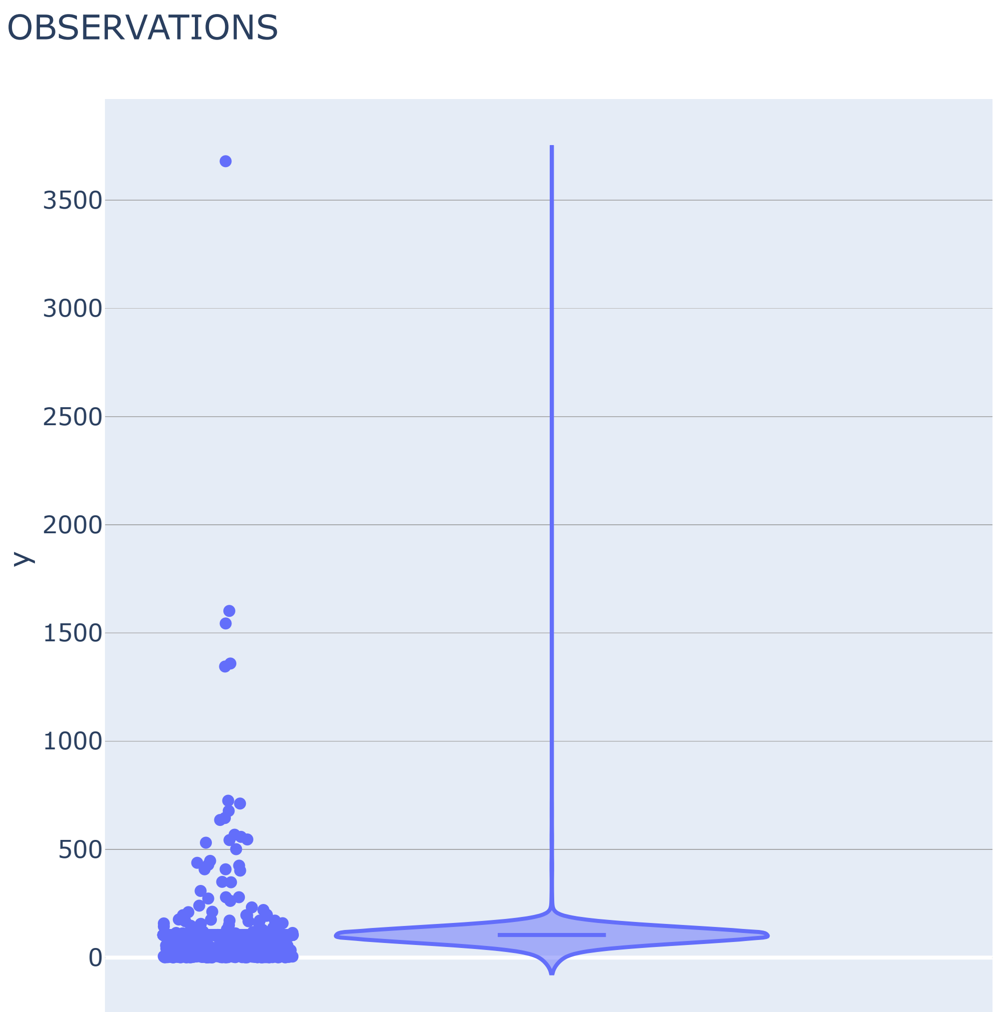
USER_SCORE




USER_COUNT
y
25k
20k
15k
10k
5k
0


MEAN_PLAYTIME
y
1000
800
600
400
200
0


MEDIAN_PLAYTIME
y
600
500
400
300
200
100
0



As expected, there are differences in the generated graphs. One notable difference is that the histograms are much more normal, which makes sense. We just added a ton of new data at the mean. But more than that, there are many more observations added to each graph. This smooths out the violin graphs as well. Now that we have a couple thousand data points, we can split the data into training and testing sets to evaluate how good of a model we have.

Before we jump right into that, though, let's look at the correlations when we use all of the data.

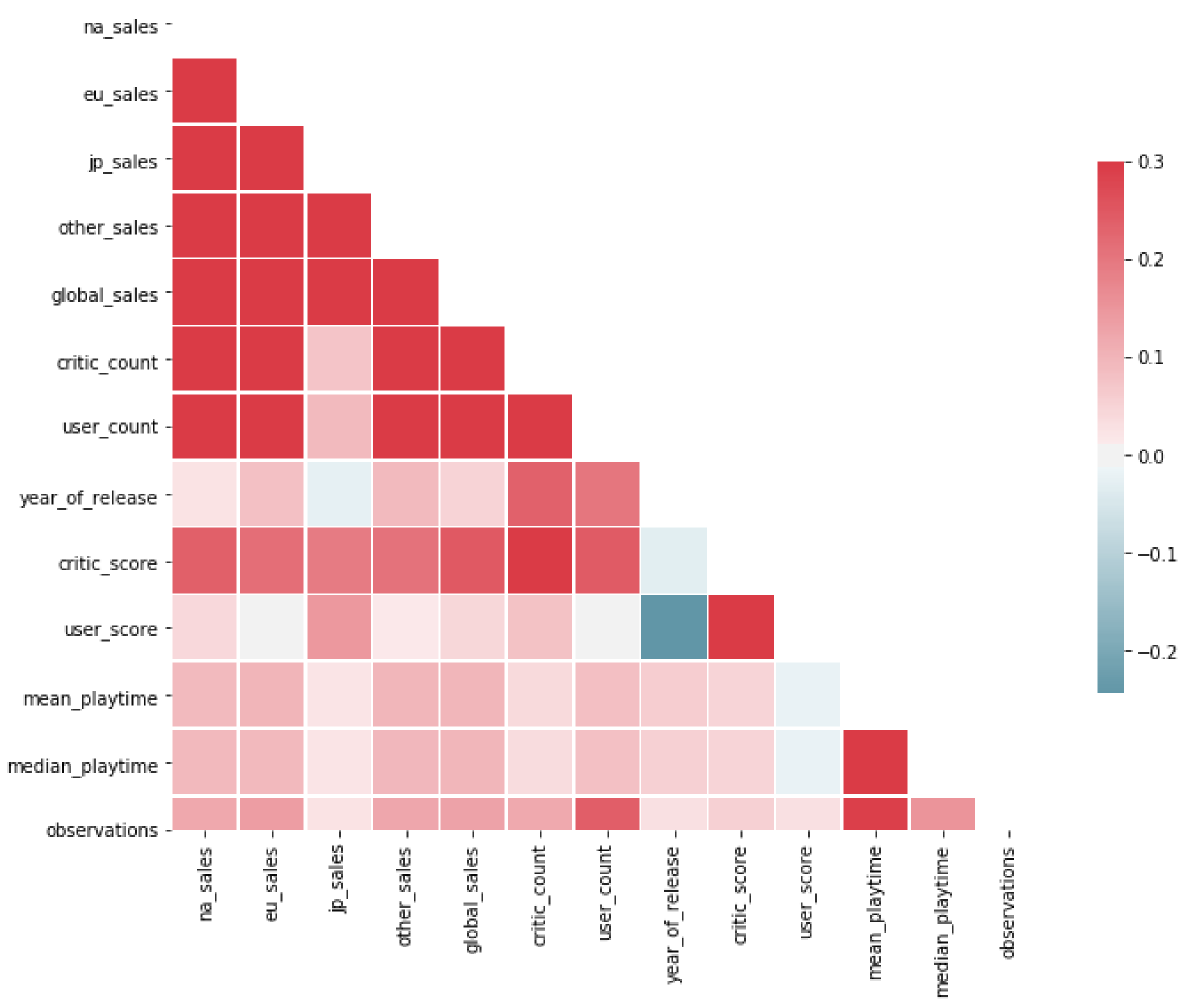
In [21]:
```
gamescores.corr()
```

Out[21]:

	na_sales	eu_sales	jp_sales	other_sales	global_sales	critic_count
na_sales	1.000000	0.826613	0.369817	0.786899	0.954571	0.427128
eu_sales	0.826613	1.000000	0.405210	0.807938	0.938608	0.391956
jp_sales	0.369817	0.405210	1.000000	0.322287	0.502809	0.074812
other_sales	0.786899	0.807938	0.322287	1.000000	0.863909	0.418510
global_sales	0.954571	0.938608	0.502809	0.863909	1.000000	0.422525
critic_count	0.427128	0.391956	0.074812	0.418510	0.422525	1.000000
user_count	0.427402	0.471672	0.090957	0.433184	0.455779	0.426706
year_of_release	0.024689	0.079801	-0.024060	0.088876	0.050385	0.236245
critic_score	0.238095	0.213765	0.192122	0.204782	0.248969	0.325263
user_score	0.040243	0.011021	0.145819	0.014696	0.044571	0.078442
mean_playtime	0.088208	0.100410	0.022993	0.098342	0.096936	0.037894
median_playtime	0.093872	0.093861	0.022050	0.095252	0.096825	0.035234
observations	0.120398	0.137288	0.026140	0.123101	0.130247	0.118413

In [22]:
```
corr = gamescores.corr()

mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

f, ax = plt.subplots(figsize=(11,11))
cmap = sns.diverging_palette(220,10, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidth=.5, cbar_kws={'shrink':.5})
```

Out[22]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f505b61ef0>
```



Significance of Correlations

Let's take a look at some of the correlations to see if they are significant or not.

In [126]:
```
print('Critic Rating vs Global Sales:\n',
      pearsonr(gamescores['critic_score'], gamescores['global_sales']),
      '\n\nUser Ratings vs Global Sales:\n',
      pearsonr(gamescores['user_score'], gamescores['global_sales']),
     '\n\nCritic Ratings vs User Ratings:\n',
     pearsonr(gamescores['critic_score'], gamescores['user_score']),
     '\n\nYear of Release vs User Score:\n',
     pearsonr(gamescores['year_of_release'], gamescores['user_score']))
```

```
Critic Rating vs Global Sales:
 (0.24896889685687515, 4.42673516640509e-69)

User Ratings vs Global Sales:
 (0.04457121235044956, 0.001956534190080404)

Critic Ratings vs User Ratings:
 (0.590012157065894, 0.0)

Year of Release vs User Score:
 (-0.24287858701999748, 9.9070211884822e-66)
```

Building Our Models

First thing up is to split our data into training and testing sets. We will need to split our predicted variable out from the rest of the data.

In [23]:
```
Y = gamescores['user_score']
X = gamescores[['year_of_release', 'na_sales', 'eu_sales', 'jp_sales',
               'other_sales', 'global_sales', 'critic_count', 'user_count',
              'critic_score', 'mean_playtime', 'median_playtime', 'observation
s']]
```

In [24]:
```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,
                                                    random_state=100)
```

Now that we have our training and testing sets, we can move forward with building our models. We will be building 3 different models to predict what the `user_score` of a game is. We will be building a Linear Regression model, a Decision Tree model, and a Random Forest model.

Linear Regression

First up is the Linear Regression model.

In [25]:
```
lr = LinearRegression()
lr.fit(x_train, y_train)
print('Intercept: ', lr.intercept_, '\nSlope: ', lr.coef_)
```

```
Intercept:  132.69451073929432
Slope:  [-6.47338374e-02 -2.06763809e+00 -2.07394534e+00 -1.74810252e+00
 -2.00584037e+00  2.01446284e+00 -3.45092165e-04 -1.06286932e-04
  6.54766468e-02 -3.33756933e-03  2.68781299e-03  8.45899876e-04]
```

In [26]:
```
coeff_df = pd.DataFrame(lr.coef_, X.columns, columns=['Coefficient'])
coeff_df
```

Out[26]:

	Coefficient
year_of_release	-0.064734
na_sales	-2.067638
eu_sales	-2.073945
jp_sales	-1.748103
other_sales	-2.005840
global_sales	2.014463
critic_count	-0.000345
user_count	-0.000106
critic_score	0.065477
mean_playtime	-0.003338
median_playtime	0.002688
observations	0.000846

Now that we have an initial model built, let's make the predictions using our testing set and compare them to the actual scores:

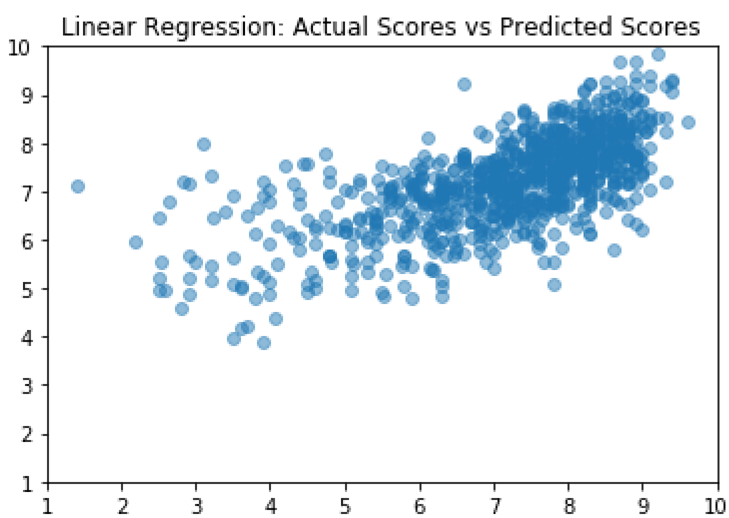
In [27]:
```
y_pred = lr.predict(x_test)
preds = pd.DataFrame({'Actual': y_test, 'Predicted':y_pred})
preds
```

Out[27]:

	Actual	Predicted
10790	7.30	7.581270
5908	6.94	6.107450
6981	7.90	7.688088
9217	4.80	7.405356
11008	8.80	7.912065
...	...	...
10607	7.80	8.616280
518	6.90	5.556195
5479	7.80	6.847751
2328	6.05	6.875831
476	8.70	7.729727

965 rows × 2 columns

In [118]:
```
plt.scatter(preds['Actual'], preds['Predicted'], alpha=.5)
plt.xlim([1,10])
plt.ylim([1,10])
plt.title('Linear Regression: Actual Scores vs Predicted Scores')
plt.show()
```



It looks like some of the predicted values are close, but others are completely off. Let's calculate the Root Mean Squared Error for the Linear Regression Model.

In [29]:
```
print('Root Mean Squared Error: ',
     np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
```

```
Root Mean Squared Error:  1.0519943487856975
```

In [30]:
```
print('Percent of Mean Error: ', np.sqrt(metrics.mean_squared_error(y_test, y_
pred)) / preds['Actual'].mean())
```

```
Percent of Mean Error:  0.14561592786985417
```

So our model has a Root Mean Squared Error of 1.05, which is roughly 14.6% of the mean value of our actual `user_score`. This means this model, as it stands now, is not all that accurate, but can make some reasonably good predictions. These good predictions can be seen as the points that lie along the y=x line in the above scatterplot.

Decision Tree Model

The next model to try is the Decision Tree Model.

In [31]:
```
dt = DecisionTreeRegressor(criterion='mse', max_depth=3)
dt.fit(x_train, y_train)
```

Out[31]:
```
DecisionTreeRegressor(criterion='mse', max_depth=3, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=None, splitter='best')
```

In [32]:
```
tree_preds = dt.predict(x_test)
tree_df = pd.DataFrame({'actual':y_test, 'preds':tree_preds})
tree_df
```

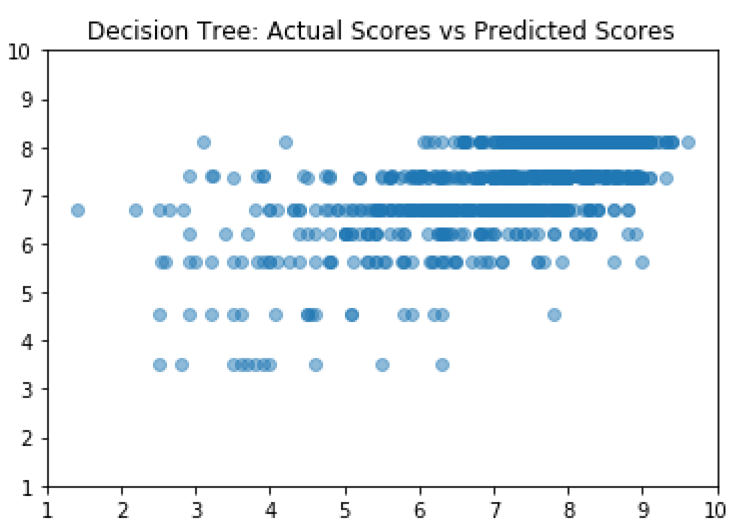
Out[32]:

	actual	preds
10790	7.30	8.109186
5908	6.94	6.219476
6981	7.90	8.109186
9217	4.80	7.395753
11008	8.80	7.395753
...	...	...
10607	7.80	8.109186
518	6.90	6.219476
5479	7.80	7.365689
2328	6.05	7.395753
476	8.70	8.109186

965 rows × 2 columns

Let's look at the scatterplot of these two values to see how it compares to the Linear Regression Model.

In [119]:
```
plt.scatter(tree_df['actual'], tree_df['preds'], alpha=.5)
plt.xlim([1,10])
plt.ylim([1,10])
plt.title('Decision Tree: Actual Scores vs Predicted Scores')
plt.show()
```



Oh, now that's interesting. But let's now calculate the Root Mean Squared Error.

In [34]:
```
print('Root Mean Squared Error: ', np.sqrt(metrics.mean_squared_error(y_test, tree_preds)),
      '\nPercent of Mean Error: ', np.sqrt(metrics.mean_squared_error(y_test, tree_preds)) / tree_df['actual'].mean())
```

```
Root Mean Squared Error:  1.0919695006621901
Percent of Mean Error:  0.1511492454574942
```

Looks like our Decision Tree Model is slightly worse than our Linear Regression model, but not by much. It looks like the Decision Tree Model overfit by using the training set.

Random Forest Model

Now let's try to improve on the Decision Tree Model by using a Random Forest Model.

In [35]:
```
rf = RandomForestRegressor(n_estimators=1000, random_state=100)
rf.fit(x_train, y_train)
```

Out[35]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=1000, n_jobs=1,
           oob_score=False, random_state=100, verbose=0, warm_start=False)
```

In [36]:
```
rf_preds = rf.predict(x_test)
rf_df = pd.DataFrame({'actual':y_test, 'preds':rf_preds})
rf_df
```

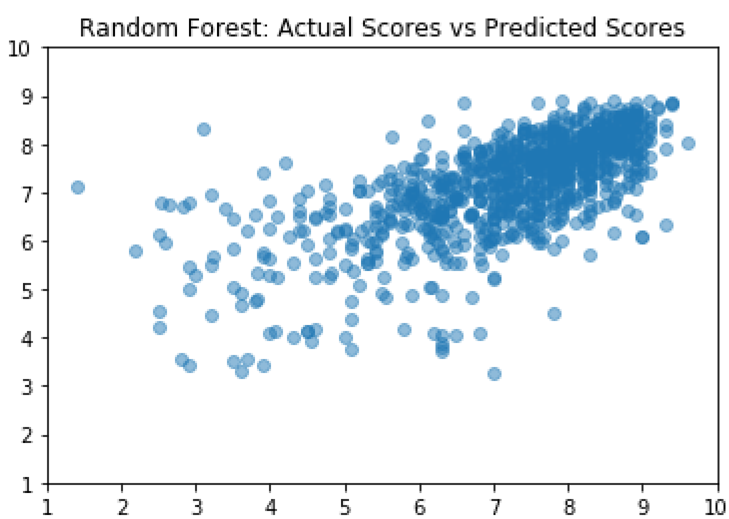
Out[36]:

	actual	preds
10790	7.30	7.87910
5908	6.94	6.00869
6981	7.90	7.99322
9217	4.80	6.87768
11008	8.80	8.02432
...	...	...
10607	7.80	8.56942
518	6.90	6.45469
5479	7.80	6.79910
2328	6.05	6.79629
476	8.70	8.24110

965 rows × 2 columns

Let's look at the scatterplot of the Random Forest Model now.

In [120]:
```
plt.scatter(rf_df['actual'], rf_df['preds'], alpha=.5)
plt.xlim([1,10])
plt.ylim([1,10])
plt.title('Random Forest: Actual Scores vs Predicted Scores')
plt.show()
```



Wow, this is much better than the single Dicision Tree Model, and looks better than the Linear Regression Model. Let's calculate the Root Mean Squared Error to compare the three models.

In [38]:
```
print('Root Mean Squared Error: ',
      np.sqrt(metrics.mean_squared_error(y_test, rf_preds)),
      '\nPercent of Mean Error: ',
      np.sqrt(metrics.mean_squared_error(y_test, rf_preds)) / rf_df['actual'].
mean())
```

```
Root Mean Squared Error:  1.0340216971353822
Percent of Mean Error:  0.1431281727318506
```

Impressive, this does have a smaller Root Mean Squared Error than the previous two models.

Another thing to note is that none of these models have been adjusted. With each model being with a single percent of the Mean Error, any of these models could be further refined and used to produce a better model.

Having said that, it is possible to get a decent estimate of what a `user_score` is going to be for a game given the sales, playtime, critic scores, and the count of user reviews, if not what the review scores are. If this were a model to predict what the `user_score` of an upcoming game will be, this model would be useless. However, these models could be used to rate previously unrated games, such as Atari games. With each game being rated, all games could then be compared to each other since video games were first developed. This could be useful to find potential patterns in what gamers do and do not like in their games.

Summary

It is possible to predict what the `user_score` of a video game is using multiple methods. Without further training and modifying of the models, the models aren't too accurate but produce decent predictions of the `user_score`.

Further Research

Further research can be done to further refine these models. Additionally, these models are reactive, not proactive. Further research can be done into building a model that will predict what the `user_score` will be for a newly developed game, one that doesn't have any playtime or sales numbers.

Sources

1. Dobrilova, T. (4 Apr. 2019) *Techjury.* "How much is the gaming industry worth?" Retrieved 26 Oct. 2019 from Techjury (https://techjury.net/stats-about/gaming-industry-worth/).
2. Tamber. Kaggle. “Steam Video Games.” Retrieved 22 Oct 2019 from Kaggle Steam Video Games (https://www.kaggle.com/tamber/steam-video-games).
3. Rakesh, J. Kaggle. “Video Game DATA.” Retrieved 22 Oct 2019 form Kaggle Video Game DATA (https://www.kaggle.com/juttugarakesh/video-game-data).
4. Tristan. Kaggle. “17k Mobile Strategy Games.” Retrieved 22 Oct 2019 from Kaggle 17k Mobile Strategy Games (https://www.kaggle.com/tristan581/17k-apple-app-store-strategy-games)
5. Smith, G. Kaggle. “Video Game Sales.” Retrieved 22 Oct 2019 from Kaggle Video Game Sales (https://www.kaggle.com/gregorut/videogamesales)
6. Kirubi, R. Kaggle. “Video Game Sales with Ratings.” Retrieved 22 Oct 2019 from Kaggle Video Game Sales with Ratings (https://www.kaggle.com/rush4ratio/video-game-sales-with-ratings)
7. SIC_TWR. Kaggle. “Video Games Sales Dataset.” Retrieved 22 Oct 2019 from Kaggle Video Games Sales Dataset (https://www.kaggle.com/sidtwr/videogames-sales-dataset)
8. Alqunber, A. Kaggle. “Video Games Sales 2019.” Retrieved 22 Oct 2019 from Kaggle Video Games Sales 2019 (https://www.kaggle.com/ashaheedq/video-games-sales-2019)
9. Gillies, K. Kaggle. “Video Game Sales and Ratings.” Retrieved 22 Oct 2019 from Kaggle Video Game Sales and Ratings (https://www.kaggle.com/kendallgillies/video-game-sales-and-ratings)
10. Christian, L. Kaggle. “Video Games Review.” Retrieved 22 Oct 2019 from Kaggle Video Games Review (https://www.kaggle.com/launay10christian/video-games-review)
11. Alex. Kaggle. “Video games with reviews and playtime statistics.” Retrieved 25 Oct 2019 from Kaggle Video Games With Reviews and Playtime Statistics (https://www.kaggle.com/alex333/video-games-with-reviews-and-playtime-statistics)