Reviewer's comment 3

All right, now everything is fine! Good luck in the next project :)

Reviewer's comment 2

Hi again, Ameer! Thank you for the fixes, but there is an error in your code :(Please fix it and I will accept your project as soon as I can😉

Reviewer's comment

Hello, Ameer! My name is Andrey Isupov. I checked you project. You did a very good job on the project. But there are few things that need fix. I think it is not so difficult for you and you will quickly cope with it. Good luck :)

Below you will find my comments - **please do not move, modify or delete them**.

You can find my comments in green, yellow or red boxes like this:

Reviewer's comment

Success. Everything is done succesfully.

Reviewer's comment

Remarks. Some recommendations.

Reviewer's comment

Needs fixing. The block requires some corrections. Work can't be accepted with the red comments.

You can answer me by using this:

Student answer.

Text here.

Optimizing The Expenses

We were offered an internship in analytical department at Yandex.Afisha. We will be studying informtaion about visitors, orders, purchases made by users and the marketing expenses/costs. Our goal is to help optimize the marketing expenses at Yandex.Afisha, to end up with a more successful, healthier business and to get rid of extra unnecessary expenses.

The purpose of the project is to determine the factors that may contribute to optimizing the expenses of a certian business. By doing that, we will end up with a healthier and more successful business that has the optimal amount of expenses spent every month/year, which in return will only make our gross revenue greater!

Plan of Work:

- We start by optimizing our data, to scale down the sizes of our datasets. Thus, running codes on them will take less time.
- Then we are going to continue by preprocessing the data, looking for missing values, duplicates and any problematic issues that we may find.
- After the initial inspection, we are going to treat the problematic data, whether it is missing data or it is duplicates.
- Once we are done with problematic data, we continue forward and start sorting the data, keeping the data that we need and getting rid of the useless data, if there's any.
- Once we're done with data optimizing and preprocessing, and we have a clean dataset to work with, we can then start analyzing the data using the methods that pandas lib. offers and display the results using graphs and histograms.
- Finally, draw general conclusions and sum up.

1 Initialization

First of all we load the libraries that we are going to work with. Then we load our datasets, take a look at them and see what we can notice by the first look.

1.1 Loading and optimizing data

Reviewer's comment

Thank you for the description

In [1]:

```
import pandas as pd
import numpy as np
from IPython.display import display
from functools import reduce
import plotly.express as px
import plotly.graph_objects as go
from datetime import datetime
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
```

In [2]:

```
# loading the dataframes
visits = pd.read_csv('/datasets/visits_log_us.csv')
orders = pd.read_csv('/datasets/orders_log_us.csv')
costs = pd.read_csv('/datasets/costs_us.csv')
```

In [3]:

```
# taking an initial look at the dataframes
display(visits.head(10))
display(orders.head(10))
display(costs.head(10))
```

	Device	End Ts	Source Id	Start Ts	Uid
0	touch	2017-12-20 17:38:00	4	2017-12-20 17:20:00	16879256277535980062
1	desktop	2018-02-19 17:21:00	2	2018-02-19 16:53:00	104060357244891740
2	touch	2017-07-01 01:54:00	5	2017-07-01 01:54:00	7459035603376831527
3	desktop	2018-05-20 11:23:00	9	2018-05-20 10:59:00	16174680259334210214
4	desktop	2017-12-27 14:06:00	3	2017-12-27 14:06:00	9969694820036681168
5	desktop	2017-09-03 21:36:00	5	2017-09-03 21:35:00	16007536194108375387
6	desktop	2018-01-30 12:09:00	1	2018-01-30 11:13:00	6661610529277171451
7	touch	2017-11-05 15:15:00	3	2017-11-05 15:14:00	11423865690854540312
8	desktop	2017-07-19 10:44:00	3	2017-07-19 10:41:00	2987360259350925644
9	desktop	2017-11-08 13:43:00	5	2017-11-08 13:42:00	1289240080042562063

	Buy Ts	Revenue	Uid
0	2017-06-01 00:10:00	17.00	10329302124590727494
1	2017-06-01 00:25:00	0.55	11627257723692907447
2	2017-06-01 00:27:00	0.37	17903680561304213844
3	2017-06-01 00:29:00	0.55	16109239769442553005
4	2017-06-01 07:58:00	0.37	14200605875248379450
5	2017-06-01 08:43:00	0.18	10402394430196413321
6	2017-06-01 08:54:00	1.83	12464626743129688638
7	2017-06-01 09:22:00	1.22	3644482766749211722
8	2017-06-01 09:22:00	3.30	17542070709969841479
9	2017-06-01 09:23:00	0.37	1074355127080856382

	source_id	dt	costs
0	1	2017-06-01	75.20
1	1	2017-06-02	62.25
2	1	2017-06-03	36.53
3	1	2017-06-04	55.00
4	1	2017-06-05	57.08
5	1	2017-06-06	40.39
6	1	2017-06-07	40.59

	source_id	dt	costs
7	1	2017-06-08	56.63
8	1	2017-06-09	40.16
9	1	2017-06-10	43.24

By the first look, we can notice that column names are in upper case sometimes, lower case in the other times. Therefore, lets start with the first step, changing column names:

In [4]:

```
# converting all column names to lower case
visits.columns = ['device', 'end_ts', 'source_id', 'start_ts', 'uid']
orders.columns = ['buy_ts', 'revenue', 'uid']

display(visits.head())
display(orders.head())
```

	device	end_ts	source_id	start_ts	uid
0	touch	2017-12-20 17:38:00	4	2017-12-20 17:20:00	16879256277535980062
1	desktop	2018-02-19 17:21:00	2	2018-02-19 16:53:00	104060357244891740
2	touch	2017-07-01 01:54:00	5	2017-07-01 01:54:00	7459035603376831527
3	desktop	2018-05-20 11:23:00	9	2018-05-20 10:59:00	16174680259334210214
4	desktop	2017-12-27 14:06:00	3	2017-12-27 14:06:00	9969694820036681168

	buy_ts	revenue	uid
0	2017-06-01 00:10:00	17.00	10329302124590727494
1	2017-06-01 00:25:00	0.55	11627257723692907447
2	2017-06-01 00:27:00	0.37	17903680561304213844
3	2017-06-01 00:29:00	0.55	16109239769442553005

Now that we got that behind our backs, we can start optimizing our datasets:

In [5]:

```
display(visits.info(memory_usage='deep'))
display(orders.info(memory_usage='deep'))
display(costs.info(memory_usage='deep'))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 359400 entries, 0 to 359399
Data columns (total 5 columns):
 #   Column     Non-Null Count   Dtype 
---  ------     --------------   ----- 
 0   device     359400 non-null  object
 1   end_ts     359400 non-null  object
 2   source_id  359400 non-null  int64 
 3   start_ts   359400 non-null  object
 4   uid        359400 non-null  uint64
dtypes: int64(1), object(3), uint64(1)
memory usage: 79.3 MB

None

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50415 entries, 0 to 50414
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   buy_ts   50415 non-null  object 
 1   revenue  50415 non-null  float64
 2   uid      50415 non-null  uint64 
dtypes: float64(1), object(1), uint64(1)
memory usage: 4.4 MB

None

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2542 entries, 0 to 2541
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   source_id  2542 non-null   int64  
 1   dt         2542 non-null   object 
 2   costs      2542 non-null   float64
dtypes: float64(1), int64(1), object(1)
memory usage: 206.2 KB

None
```

As we can see above, dates are in form of strings, and a couple other columns can be converted into a *category* type:

In [6]:

```
# converting dates into datetime and device into category
visits['end_ts'] = pd.to_datetime(visits['end_ts'])
visits['start_ts'] = pd.to_datetime(visits['start_ts'])
orders['buy_ts'] = pd.to_datetime(orders['buy_ts'])
costs['dt'] = pd.to_datetime(costs['dt'])
```

In [7]:

```
# converting device into category type
visits['device'] = visits['device']
```

In [8]:

```
# checking if these columns are compatible to converted into category type
display(visits['device'].value_counts())
display(visits['source_id'].value_counts())
display(costs['source_id'].value_counts())
```

```
desktop    262567
touch       96833
Name: device, dtype: int64
```

```
4     101794
3      85610
5      66905
2      47626
1      34121
9      13277
10     10025
7         36
6          6
Name: source_id, dtype: int64
```

```
5     364
2     363
4     363
10    363
1     363
3     363
9     363
Name: source_id, dtype: int64
```

In [9]:

```
# converting to category type
visits['device'] = visits['device'].astype('category')
visits['source_id'] = visits['source_id'].astype('category')
costs['source_id'] = costs['source_id'].astype('category')
```

Now we are done optimizing our data, let's seem by how much did we scale down the size of our dataframes:

In [10]:

```
display(visits.info(memory_usage='deep'))
display(orders.info(memory_usage='deep'))
display(costs.info(memory_usage='deep'))
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 359400 entries, 0 to 359399
Data columns (total 5 columns):
 #   Column     Non-Null Count   Dtype         
---  ------     --------------   -----         
 0   device     359400 non-null  category      
 1   end_ts     359400 non-null  datetime64[ns]
 2   source_id  359400 non-null  category      
 3   start_ts   359400 non-null  datetime64[ns]
 4   uid        359400 non-null  uint64        
dtypes: category(2), datetime64[ns](2), uint64(1)
memory usage: 8.9 MB

None

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50415 entries, 0 to 50414
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype         
---  ------   --------------  -----         
 0   buy_ts   50415 non-null  datetime64[ns]
 1   revenue  50415 non-null  float64       
 2   uid      50415 non-null  uint64        
dtypes: datetime64[ns](1), float64(1), uint64(1)
memory usage: 1.2 MB

None

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2542 entries, 0 to 2541
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype         
---  ------     --------------  -----         
 0   source_id  2542 non-null   category      
 1   dt         2542 non-null   datetime64[ns]
 2   costs      2542 non-null   float64       
dtypes: category(1), datetime64[ns](1), float64(1)
memory usage: 42.7 KB

None
```

Reviewer's comment

Very good data review!

The dataframes were scaled down in size massively! This is going to help us save alot of time analyzing these dataframes.

Now that we are done with optimizing the data, let's check if the dataframes are clean and ready to go.

1.2 Preprocessing the data

First, we have no missing values obviously, looking at the results above. Second, we already changed the column names to lower-case. We also inspected the values of some of the columns we have, nothing suspecious there. We're left with checking for duplicates and then we'll be set to go:

In [11]:

```
# Checking for dups in the dataframes
display(visits.duplicated().sum())
display(orders.duplicated().sum())
display(costs.duplicated().sum())
```

```
0
```

```
0
```

```
0
```

A duplicate in these dataframes is a row that is repeated more than one time (same uid and date). We have 0 of these rows in our dataframes, we are all set!

Reviewer's comment

Good

2 Making reports and calculating metrics

2.1 Product

First we are asked to calculate how much people use our service per day, week and month. To do so, we add two extra columns in the **visits** dataframe, 'week' and 'month', in order to be able to group by those columns later on.

In [12]:

```
# creating the week and month columns
visits['week']  = visits['start_ts'].astype('datetime64[W]')
visits['month'] = visits['start_ts'].astype('datetime64[M]')
visits
```

Out[12]:

	device	end_ts	source_id	start_ts	uid	week	month
0	touch	2017-12-20 17:38:00	4	2017-12-20 17:20:00	16879256277535980062	2017-12-14	2017-12-01
1	desktop	2018-02-19 17:21:00	2	2018-02-19 16:53:00	104060357244891740	2018-02-15	2018-02-01
2	touch	2017-07-01 01:54:00	5	2017-07-01 01:54:00	7459035603376831527	2017-06-29	2017-07-01
3	desktop	2018-05-20 11:23:00	9	2018-05-20 10:59:00	16174680259334210214	2018-05-17	2018-05-01
4	desktop	2017-12-27 14:06:00	3	2017-12-27 14:06:00	9969694820036681168	2017-12-21	2017-12-01
...	...	...	...	...	...	...	...
359395	desktop	2017-07-29 19:07:19	2	2017-07-29 19:07:00	18363291481961487539	2017-07-27	2017-07-01
359396	touch	2018-01-25 17:38:19	1	2018-01-25 17:38:00	18370831553019119586	2018-01-25	2018-01-01
359397	desktop	2018-03-03 10:12:19	4	2018-03-03 10:12:00	18387297585500748294	2018-03-01	2018-03-01
359398	desktop	2017-11-02 10:12:19	5	2017-11-02 10:12:00	18388616944624776485	2017-11-02	2017-11-01
359399	touch	2017-09-10 13:13:19	2	2017-09-10 13:13:00	18396128934054549559	2017-09-07	2017-09-01

359400 rows × 7 columns

In [13]:

```
# get the YYYY-MM-DD date of every session
visits['session_date'] = visits['start_ts'].dt.date
visits
```

Out[13]:

	device	end_ts	source_id	start_ts	uid	week	month	session_d
0	touch	2017-12-20 17:38:00	4	2017-12-20 17:20:00	16879256277535980062	2017-12-14	2017-12-01	2017-12
1	desktop	2018-02-19 17:21:00	2	2018-02-19 16:53:00	104060357244891740	2018-02-15	2018-02-01	2018-02
2	touch	2017-07-01 01:54:00	5	2017-07-01 01:54:00	7459035603376831527	2017-06-29	2017-07-01	2017-07
3	desktop	2018-05-20 11:23:00	9	2018-05-20 10:59:00	16174680259334210214	2018-05-17	2018-05-01	2018-05
4	desktop	2017-12-27 14:06:00	3	2017-12-27 14:06:00	9969694820036681168	2017-12-21	2017-12-01	2017-12
...	...	...	...	...	...	...	...	
359395	desktop	2017-07-29 19:07:19	2	2017-07-29 19:07:00	18363291481961487539	2017-07-27	2017-07-01	2017-07
359396	touch	2018-01-25 17:38:19	1	2018-01-25 17:38:00	18370831553019119586	2018-01-25	2018-01-01	2018-01
359397	desktop	2018-03-03 10:12:19	4	2018-03-03 10:12:00	18387297585500748294	2018-03-01	2018-03-01	2018-03
359398	desktop	2017-11-02 10:12:19	5	2017-11-02 10:12:00	18388616944624776485	2017-11-02	2017-11-01	2017-11
359399	touch	2017-09-10 13:13:19	2	2017-09-10 13:13:00	18396128934054549559	2017-09-07	2017-09-01	2017-09

359400 rows × 8 columns

2.1.1 How many people use it every day, week, and month?

To find out, we simply group by the week and month columns we added in the last block and count the unique user ids:

In [14]:

```
# calculating how much unique user ids there is every day
dau = visits.groupby(visits['session_date'])['uid'].nunique().reset_index()
# calculating how much unique user ids there is every week
wau=visits.groupby(visits['week'])['uid'].nunique().reset_index()
# calculating how much unique user ids there is every month
mau=visits.groupby(visits['month'])['uid'].nunique().reset_index()
# displaying results
display(dau.head(7))
display(wau.head(4))
display(mau.head(12))
```

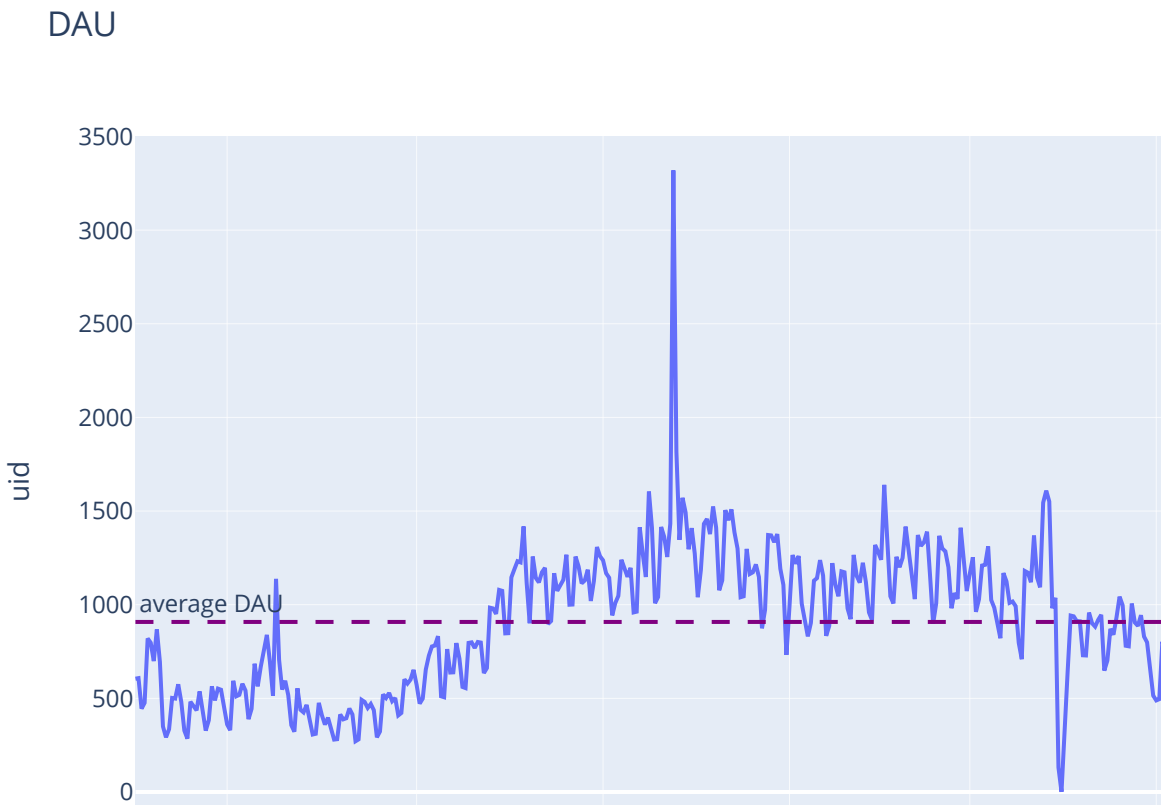
	session_date	uid
0	2017-06-01	605
1	2017-06-02	608
2	2017-06-03	445
3	2017-06-04	476
4	2017-06-05	820
5	2017-06-06	797
6	2017-06-07	699

	week	uid
0	2017-06-01	4082
1	2017-06-08	3311
2	2017-06-15	2844
3	2017-06-22	3079

	month	uid
0	2017-06-01	13259
1	2017-07-01	14183
2	2017-08-01	11631
3	2017-09-01	18975
4	2017-10-01	29692
5	2017-11-01	32797
6	2017-12-01	31557
7	2018-01-01	28716
8	2018-02-01	28749
9	2018-03-01	27473
10	2018-04-01	21008
11	2018-05-01	20701

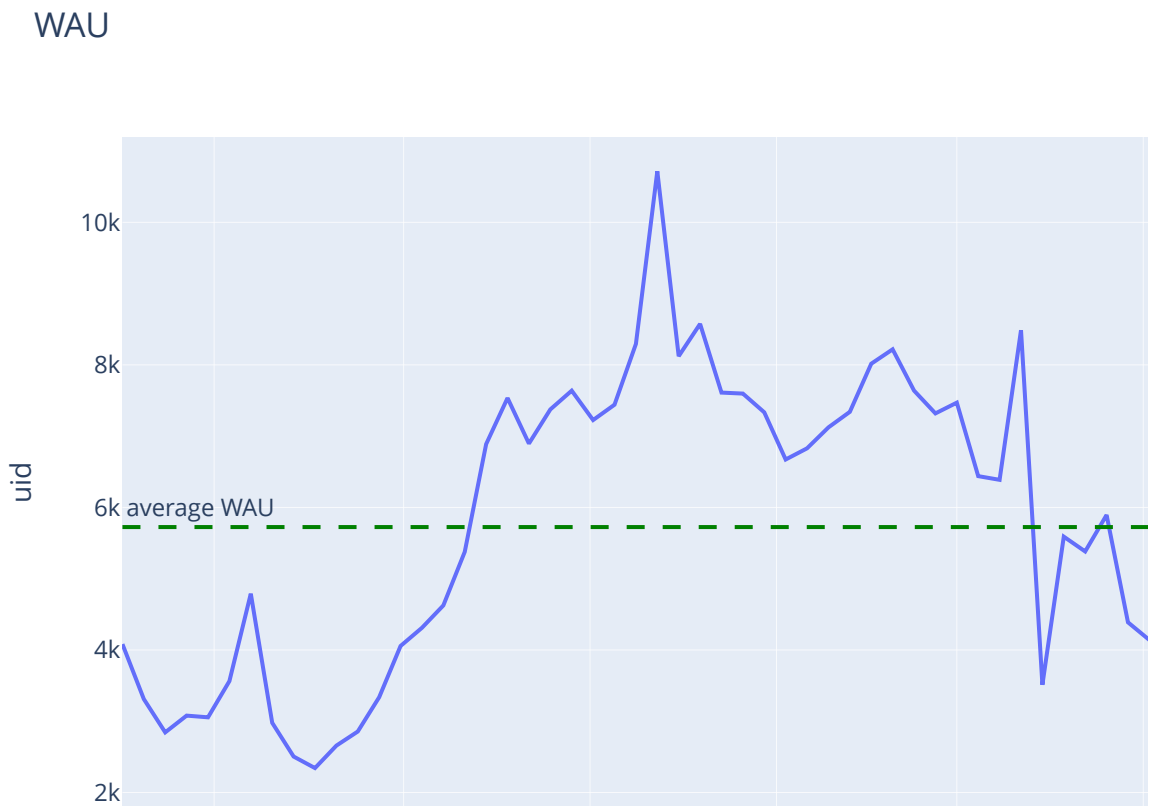
In [15]:

```
# a graph describing the daily visitors
fig = px.line(dau, x = "session_date", y = "uid", title = 'DAU')

fig.add_hline(y = dau['uid'].mean(), line_dash = "dash", line_color = "purple", annotation_
fig.show()
```



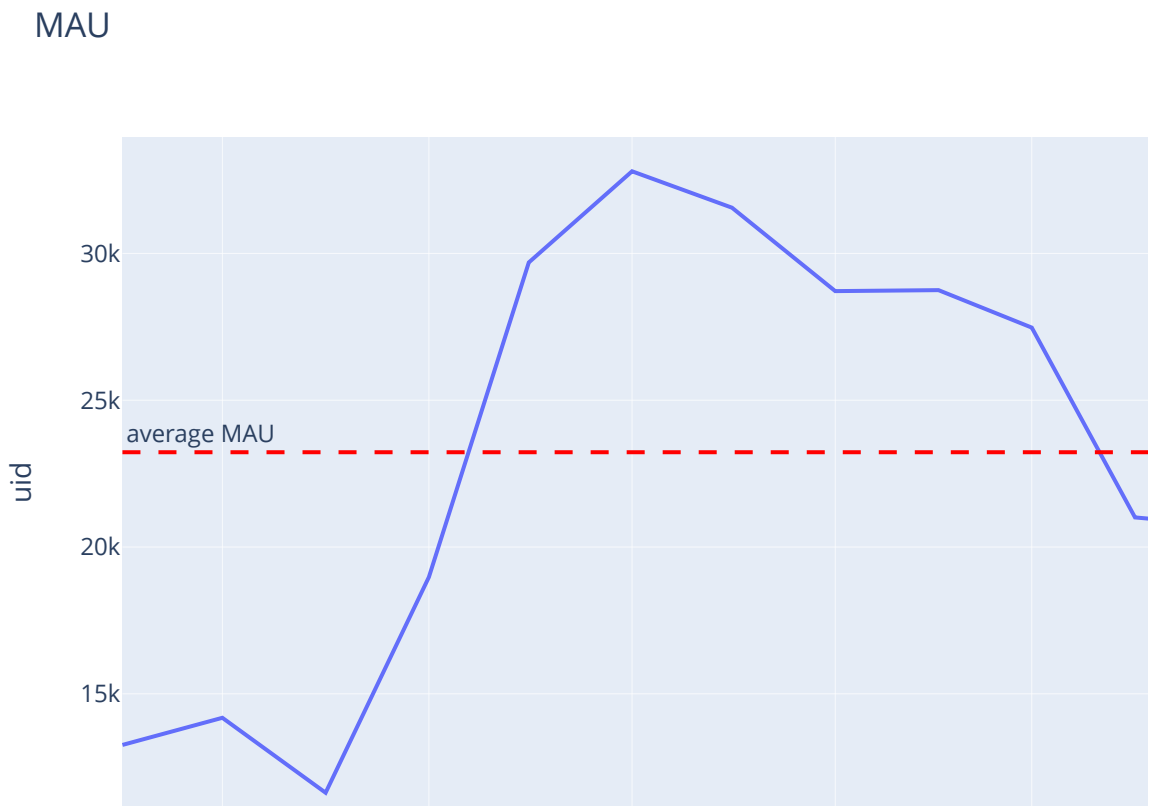
In [16]:

```
# a graph describing the weekly visitors
fig = px.line(wau, x = "week", y = "uid", title = 'WAU')

fig.add_hline(y = wau['uid'].mean(), line_dash = "dash", line_color = "green", annotation_t
fig.show()
```



In [17]:

```
# a graph describing the monthly visitors
fig = px.line(mau, x = "month", y = "uid", title = 'MAU')

fig.add_hline(y = mau['uid'].mean(), line_dash = "dash", line_color = "red", annotation_tex
fig.show()
```



Reviewer's comment

All right here, very good graphs👍

We can see that in Nov, 2017 we have the highest surge of visitors on our service, peaking in all graphs. What could've been the reason for this surge, maybe because of the season, or maybe more money were spent on promotions during this time? We'll answer this question later on.

2.1.2 How many sessions are there per day?

To calculate how many sessions are there per day, a good way to calculate how many sessions there are per day per user. This way we get info about sessions per user and about sessions per day alltogether!

In [18]:

```
dau_per_user = visits.groupby(visits['session_date']).agg({'start_ts': 'count','uid':'nuniq
dau_per_user['sessions_per_user']=dau_per_user['start_ts']/dau_per_user['uid']
dau_per_user
```

Out[18]:

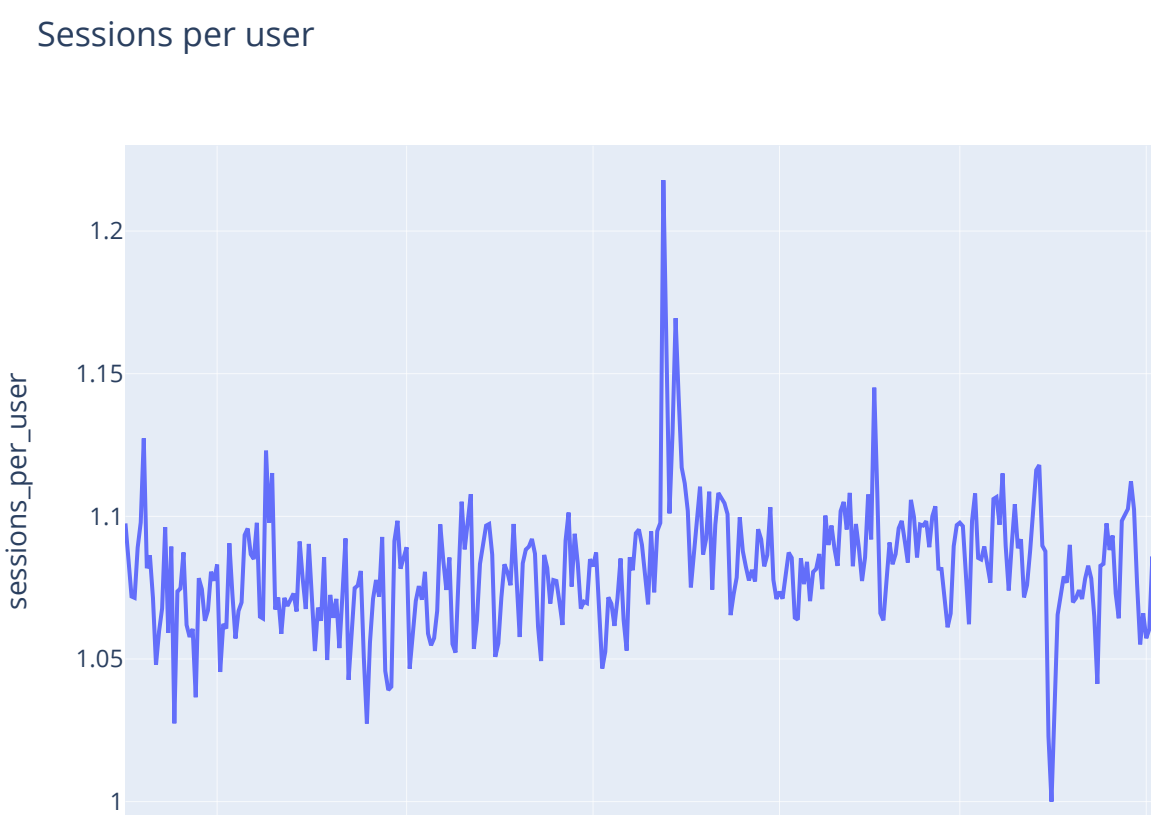
	session_date	start_ts	uid	sessions_per_user
0	2017-06-01	664	605	1.097521
1	2017-06-02	658	608	1.082237
2	2017-06-03	477	445	1.071910
3	2017-06-04	510	476	1.071429
4	2017-06-05	893	820	1.089024
...	...	...	...	...
359	2018-05-27	672	620	1.083871
360	2018-05-28	1156	1039	1.112608
361	2018-05-29	1035	948	1.091772
362	2018-05-30	1410	1289	1.093871
363	2018-05-31	2256	1997	1.129695

364 rows × 4 columns

Results displayed in a graph below:

In [19]:

```
fig = px.line(dau_per_user, x = "session_date", y = "sessions_per_user", title = 'Sessions
fig.show()
```



Similar to our previous graphs, this graph also has a great surge in Nov of 2017. This means, in Nov of 2017, we had the highest **sessions_per_user** rate which means many users visited the service more than once. Why would this happen? As we stated earlier, we need more studying to actually answer this question, there could be many reasons.

Reviewer's comment

Right again

2.1.3 What is the length of each session?

The length of a session can be calculated via the visits dataframe, by subtracting the **end_ts** from the **start_ts**. Why does the session length matter? This could tell us how immersive was our service, did the users spend much time using the service? and may also be a contributer to more sales/revenue.

In [20]:

```
visits['session_length'] = visits['end_ts'] - visits['start_ts']
```

In [21]:

```
session_length = visits.groupby(['start_ts', 'uid'])['session_length'].sum().reset_index()
display(session_length)
```

	start_ts	uid	session_length
0	2017-06-01 00:01:00	13890188992670018146	0 days 00:01:00
1	2017-06-01 00:02:00	16152015161748786004	0 days 00:00:00
2	2017-06-01 00:02:00	16706502037388497502	0 days 00:14:00
3	2017-06-01 00:04:00	8842918131297115663	0 days 00:00:00
4	2017-06-01 00:09:00	10329302124590727494	0 days 00:02:00
...	...	...	...
359394	2018-05-31 23:59:00	83872787173869366	0 days 00:05:00
359395	2018-05-31 23:59:00	3720373600909378583	0 days 00:12:00
359396	2018-05-31 23:59:00	4906562732540547408	0 days 00:26:00
359397	2018-05-31 23:59:00	10406407303624848652	0 days 00:00:00
359398	2018-05-31 23:59:00	10723414689244282024	0 days 00:13:00

359399 rows × 3 columns

In [22]:

```
display(session_length['session_length'].mean())
display(session_length['session_length'].median())
```

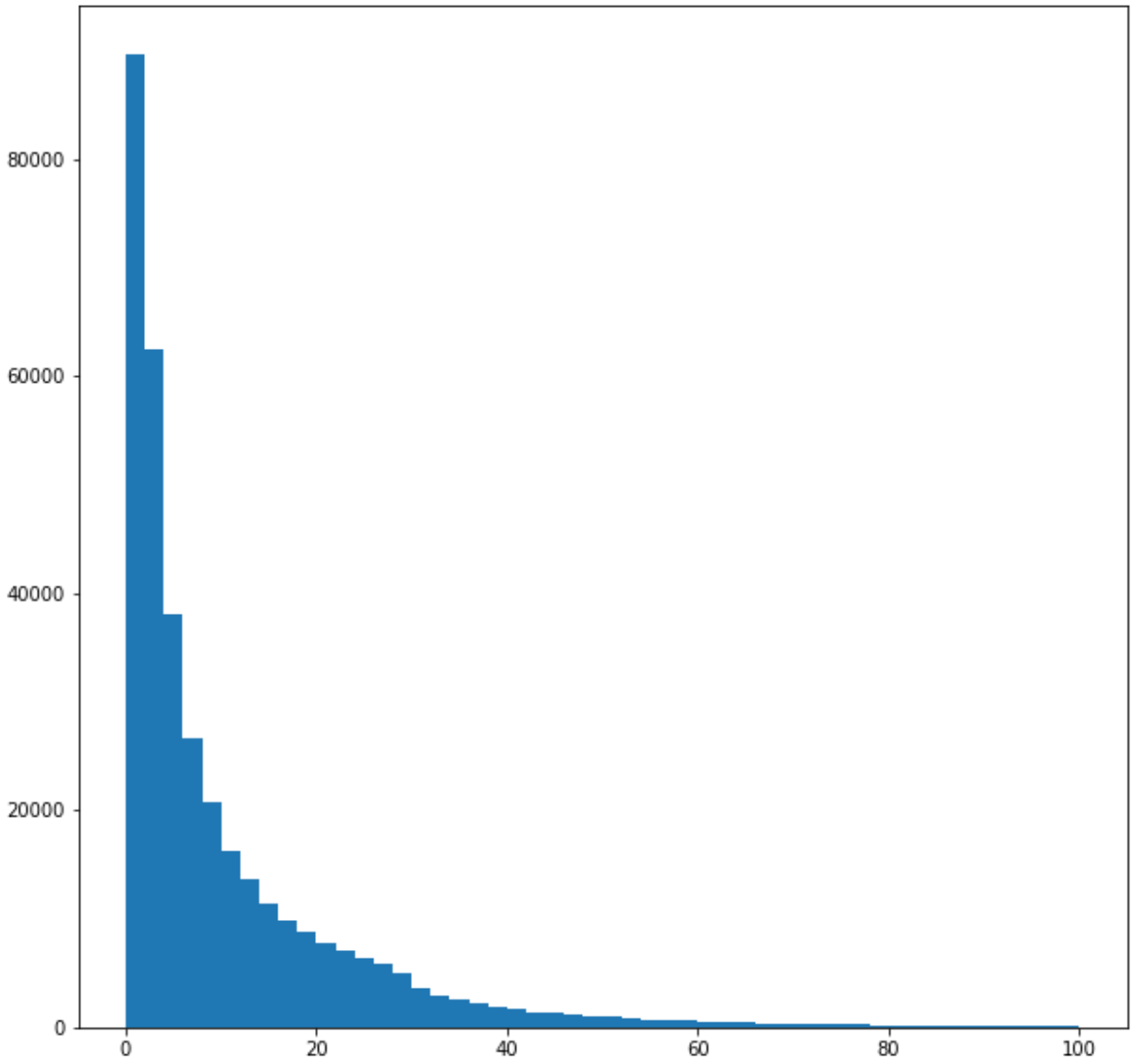
```
Timedelta('0 days 00:10:43.027476425')
```

```
Timedelta('0 days 00:05:00')
```

The average session length was around 11 minutes, while the median is at exactly 5 minutes. Here is a histogram showing the distribution of the session lengths:

In [23]:

```
plt.figure(figsize=(10,10))
plt.hist(session_length['session_length'].dt.seconds/60, bins = 50, range = (0,100))
plt.show()
```



Reviewer's comment

You can add plt.show() to remove useless array output

Student answer.

Thanks, didn't know about this! (I knew about plt.show, just didn't ever notice that it hides these text lines)

CORRECTED

Reviewer's comment 2

👍

There is a large amount of sessions with a length of 0 to 5 minutes.

Reviewer's comment

Good work!

2.1.4 How often do users come back?

This is basically the retention rate, and we know how to calculate the retention rate: we get the first session of every user/visitor and divide the visitors into cohorts accordingly and calculate the retention rate according to these cohorts.

In [24]:

```
# getting the first session date of each user to divide into cohorts later on
min_visit = visits.groupby(['uid'])['start_ts'].min().reset_index()
min_visit.columns = ['uid', 'first_session']
min_visit.head()
```

Out[24]:

	uid	first_session
0	11863502262781	2018-03-01 17:27:00
1	49537067089222	2018-02-06 15:55:00
2	297729379853735	2017-06-07 18:47:00
3	313578113262317	2017-09-18 22:49:00
4	325320750514679	2017-09-30 14:29:00

In [25]:

```
# adding the first_session column to the visits dataframe
visits = visits.merge(min_visit, how='inner', on=['uid'])
visits.head()
```

Out[25]:

	device	end_ts	source_id	start_ts	uid	week	month	session_date	s
0	touch	2017-12-20 17:38:00	4	2017-12-20 17:20:00	16879256277535980062	2017-12-14	2017-12-01	2017-12-20	(
1	desktop	2018-02-19 17:21:00	2	2018-02-19 16:53:00	104060357244891740	2018-02-15	2018-02-01	2018-02-19	(
2	touch	2017-07-01 01:54:00	5	2017-07-01 01:54:00	7459035603376831527	2017-06-29	2017-07-01	2017-07-01	(
3	desktop	2018-05-20 11:23:00	9	2018-05-20 10:59:00	16174680259334210214	2018-05-17	2018-05-01	2018-05-20	(
4	desktop	2018-03-09 20:33:00	4	2018-03-09 20:05:00	16174680259334210214	2018-03-08	2018-03-01	2018-03-09	(

In [26]:

```
# attaching each uid to a cohort and specifying the age of the current user visit cohort
visits['cohort'] = visits['first_session'].astype('datetime64[M]')
visits['age'] = ((pd.to_datetime(visits['session_date']) - pd.to_datetime(visits['first_ses
visits.head()
```

Out[26]:

	device	end_ts	source_id	start_ts	uid	week	month	session_date	s
0	touch	2017-12-20 17:38:00	4	2017-12-20 17:20:00	16879256277535980062	2017-12-14	2017-12-01	2017-12-20	(
1	desktop	2018-02-19 17:21:00	2	2018-02-19 16:53:00	104060357244891740	2018-02-15	2018-02-01	2018-02-19	(
2	touch	2017-07-01 01:54:00	5	2017-07-01 01:54:00	7459035603376831527	2017-06-29	2017-07-01	2017-07-01	(
3	desktop	2018-05-20 11:23:00	9	2018-05-20 10:59:00	16174680259334210214	2018-05-17	2018-05-01	2018-05-20	(
4	desktop	2018-03-09 20:33:00	4	2018-03-09 20:05:00	16174680259334210214	2018-03-08	2018-03-01	2018-03-09	(

In [27]:

```
cohorts = visits.pivot_table(index='cohort',
                  columns='age',
                  values='uid',
                  aggfunc='nunique').fillna(0)
display(cohorts)
```

age	0	1	2	3	4	5	6	7	8	9	10	11
cohort												
2017-06-01	13259.0	976.0	704.0	814.0	915.0	886.0	850.0	732.0	749.0	658.0	561.0	521.0
2017-07-01	13140.0	743.0	677.0	712.0	741.0	641.0	589.0	638.0	492.0	376.0	320.0	65.0
2017-08-01	10181.0	710.0	621.0	604.0	515.0	430.0	416.0	343.0	296.0	233.0	29.0	0.0
2017-09-01	16704.0	1289.0	1096.0	827.0	668.0	660.0	556.0	424.0	333.0	39.0	0.0	0.0
2017-10-01	25977.0	1912.0	1346.0	1001.0	948.0	805.0	546.0	460.0	81.0	0.0	0.0	0.0
2017-11-01	27248.0	1888.0	1246.0	1060.0	860.0	646.0	502.0	66.0	0.0	0.0	0.0	0.0
2017-12-01	25268.0	1301.0	1010.0	737.0	546.0	402.0	67.0	0.0	0.0	0.0	0.0	0.0
2018-01-01	22624.0	1212.0	809.0	580.0	387.0	37.0	0.0	0.0	0.0	0.0	0.0	0.0
2018-02-01	22197.0	1061.0	581.0	402.0	42.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2018-03-01	20589.0	854.0	503.0	75.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2018-04-01	15709.0	624.0	62.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2018-05-01	15273.0	95.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

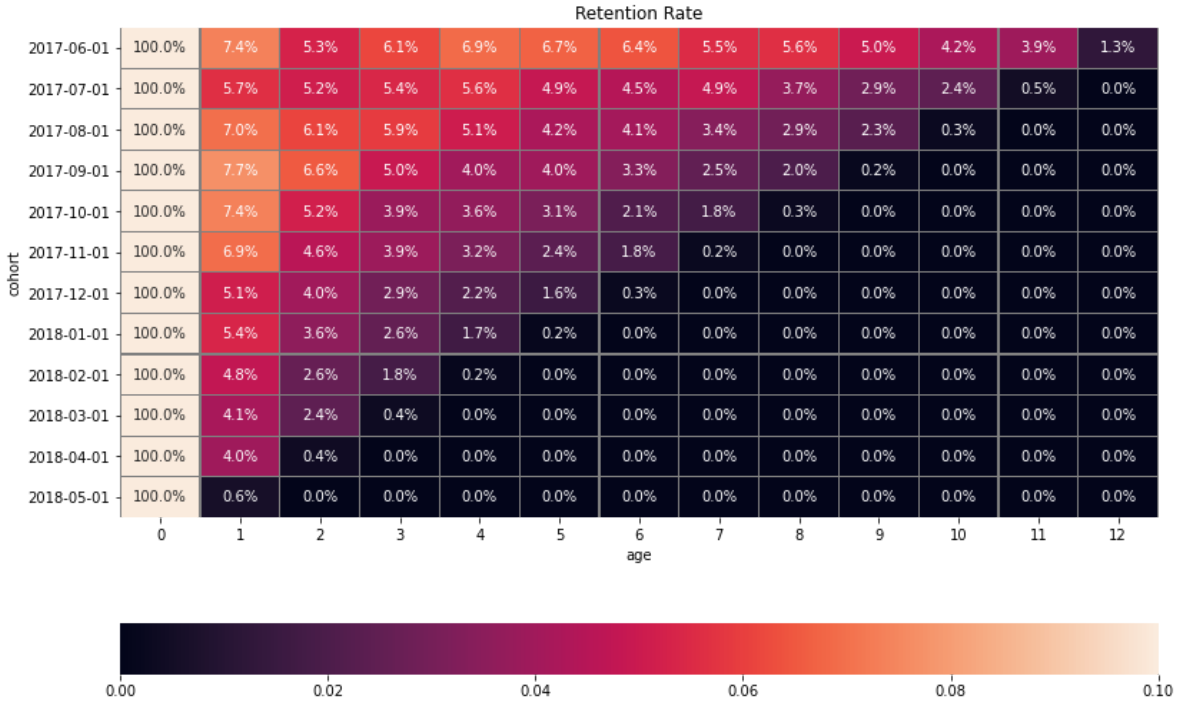
Below we show the results in a heatmap, for clearler conclusions:

In [28]:

```
retention=cohorts.iloc[:,0:].div(cohorts[0], axis=0)
```

In [29]:

```
retention.index=retention.index.astype(str)
plt.figure(figsize=(13, 9))
sns.heatmap(retention, annot=True, fmt='.1%', linewidths=0.1, linecolor='grey',  vmax=0.1,
            ).set(title = 'Retention Rate')
plt.show()
```



Judging by the heatmap above, we can say that the older cohorts had slightly higher retention rate (difference of 2%), but not only that, also slightly higher churn rates but it's not that great.

2.1.5 Summary

To sum up on **product** metrics:

- First, we calculated daily, monthly and weekly users/visitors, what we noticed and was very obvious is that daily, weekly and monthly number of visitors was increasing day by day, week by week and month by month up until Nov of 2017. Same month we got our peak visitors, after that visitors numbers started to slowly but surely go down.
- Second, we calculated how many sessions we had per day (per user) with nothing much special other than the fact that also in Nov of 2017 we had the highest sessions per day per user. Sessions per day per users was steady since the beginning of the time period with a minimum of 1 visits per day per user and a maximum of around 1.2 visits per day per user.

- Lastly, we calculated the lengths of each sessions, peaking from 0 to 5 minutes per session (which explains our median: 5 minutes) and averaging 10 to 11 minutes, due to the great outliers we had.

Reviewer's comment

Beautiful heatmap!

2.2 Sales

2.2.1 When do people start buying?

AKA Conversion rate. To get the conversion rate, we have find the users that started as visitors and ended up as customers. Such users are users that are both in the **visits** and **orders** dateframe.

Lets calculate how long does it take the average user to become a customer and how many of our users actually become customers, meaning, make their first purchase:

In [30]:

```
# Getting only orders with revenue higher than 0 because why not
purchase = orders[orders['revenue'] > 0]
purchase
```

Out[30]:

	buy_ts	revenue	uid
0	2017-06-01 00:10:00	17.00	10329302124590727494
1	2017-06-01 00:25:00	0.55	11627257723692907447
2	2017-06-01 00:27:00	0.37	17903680561304213844
3	2017-06-01 00:29:00	0.55	16109239769442553005
4	2017-06-01 07:58:00	0.37	14200605875248379450
...	...	...	...
50410	2018-05-31 23:50:00	4.64	12296626599487328624
50411	2018-05-31 23:50:00	5.80	11369640365507475976
50412	2018-05-31 23:54:00	0.30	1786462140797698849
50413	2018-05-31 23:56:00	3.67	3993697860786194247
50414	2018-06-01 00:02:00	3.42	83872787173869366

50364 rows × 3 columns

In [31]:

```
# Getting the first order date for each customer
first_order = purchase.groupby(['uid'])['buy_ts'].min().reset_index()
first_order.columns = ['uid', 'first_order']
first_order.head()
```

Out[31]:

	uid	first_order
0	313578113262317	2018-01-03 21:51:00
1	1575281904278712	2017-06-03 10:13:00
2	2429014661409475	2017-10-11 18:33:00
3	2464366381792757	2018-01-28 15:54:00
4	2551852515556206	2017-11-24 10:14:00

In [32]:

```
# Merging the purchase with first_order df and min_visit df (first visits as well)
purchase = purchase.merge(first_order, how='left', on=['uid'])
purchase = purchase.merge(min_visit, how='left', on=['uid'])
purchase.head()
```

Out[32]:

	buy_ts	revenue	uid	first_order	first_session
0	2017-06-01 00:10:00	17.00	10329302124590727494	2017-06-01 00:10:00	2017-06-01 00:09:00
1	2017-06-01 00:25:00	0.55	11627257723692907447	2017-06-01 00:25:00	2017-06-01 00:14:00
2	2017-06-01 00:27:00	0.37	17903680561304213844	2017-06-01 00:27:00	2017-06-01 00:25:00
3	2017-06-01 00:29:00	0.55	16109239769442553005	2017-06-01 00:29:00	2017-06-01 00:14:00
4	2017-06-01 07:58:00	0.37	14200605875248379450	2017-06-01 07:58:00	2017-06-01 07:31:00

In [33]:

```
purchase.info()
```

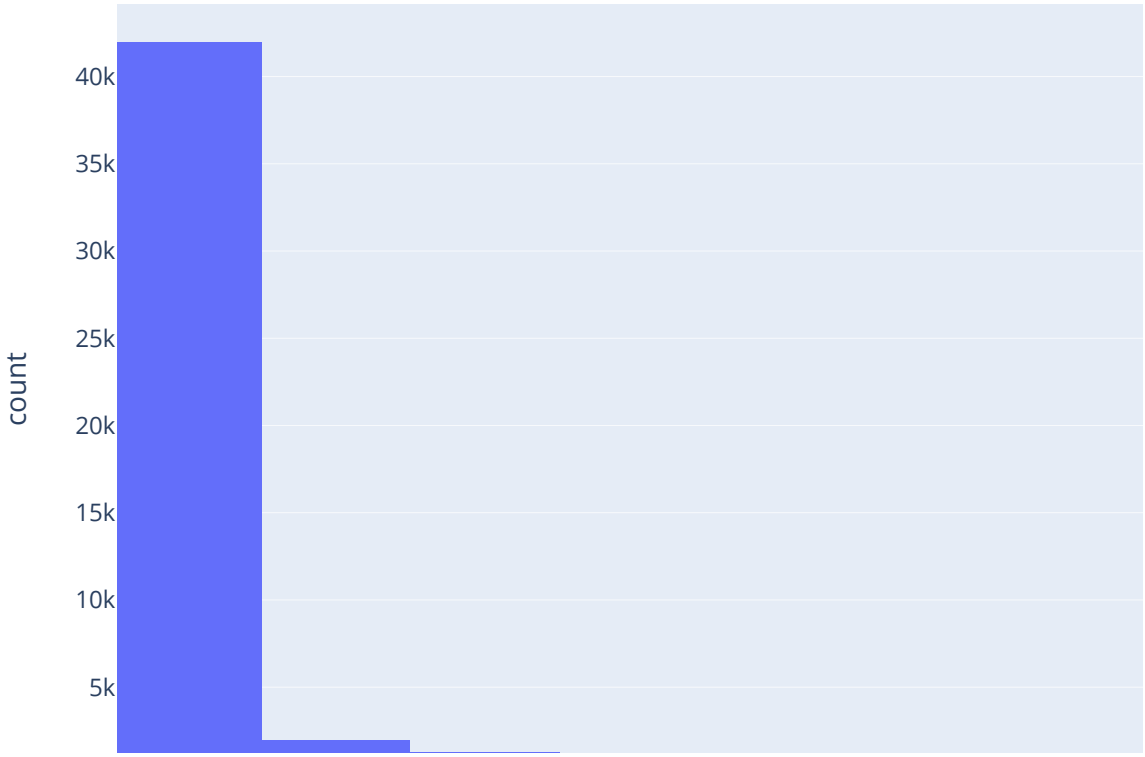
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 50364 entries, 0 to 50363
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype         
---  ------         --------------  -----         
 0   buy_ts         50364 non-null  datetime64[ns]
 1   revenue        50364 non-null  float64       
 2   uid            50364 non-null  uint64        
 3   first_order    50364 non-null  datetime64[ns]
 4   first_session  50364 non-null  datetime64[ns]
dtypes: datetime64[ns](3), float64(1), uint64(1)
memory usage: 2.3 MB
```

In [34]:

```
# calculating how much does it take for a user to convert into a customer
purchase['conversion'] = ((pd.to_datetime(purchase['first_order'])-pd.to_datetime(purchase[
```

In [35]:

```
# the distibution of customers' conversion time
fig = px.histogram(purchase, x = "conversion", nbins = 30)
fig.update_xaxes(range=[0,150])
fig.show()
```



In [36]:

```
display(purchase['conversion'].mean())
display(purchase['conversion'].median())
```

```
17.413549360654436
```

```
0.0
```

In [37]:

```
print('The user conversion rate is {:.2%}'.format(purchase['uid'].nunique()/visits['uid'].n
```

```
The user conversion rate is 16.00%
```

On average, it takes the user around 17 days to become a customer and make his first order, but in reality the numbers are much much lower than that and this can be supported by the histogram above and the value of the median (0 days). Other than that, we found out that 16% of all the users became customers and made their first order.

Reviewer's comment

You are right

2.2.2 How many orders do they make during a given period of time?

Similarly to how we calculated sessions per day (per user) , we can calculate the orders amount (per month per customer). But this time we chose to go with a month period of time instead of daily, since we don't have as much orders as visits.

In [38]:

```
# extracting the first order month
purchase['first_order_month'] = purchase['first_order'].astype('datetime64[M]')
purchase['month'] = purchase['buy_ts'].astype('datetime64[M]')
purchase.head()
```

Out[38]:

	buy_ts	revenue	uid	first_order	first_session	conversion	first_order_r
0	2017-06-01 00:10:00	17.00	10329302124590727494	2017-06-01 00:10:00	2017-06-01 00:09:00	0	2017-
1	2017-06-01 00:25:00	0.55	11627257723692907447	2017-06-01 00:25:00	2017-06-01 00:14:00	0	2017-
2	2017-06-01 00:27:00	0.37	17903680561304213844	2017-06-01 00:27:00	2017-06-01 00:25:00	0	2017-
3	2017-06-01 00:29:00	0.55	16109239769442553005	2017-06-01 00:29:00	2017-06-01 00:14:00	0	2017-
4	2017-06-01 07:58:00	0.37	14200605875248379450	2017-06-01 07:58:00	2017-06-01 07:31:00	0	2017-

In [39]:

```
# calculating the sizes of each cohort
cohort_sizes = purchase.groupby('first_order_month').agg({'uid': 'nunique'}).reset_index()
cohort_sizes.columns=['first_order_month','cohort_size']
cohort_sizes.head()
```

Out[39]:

	first_order_month	cohort_size
0	2017-06-01	2022
1	2017-07-01	1922
2	2017-08-01	1369
3	2017-09-01	2579
4	2017-10-01	4340

In [40]:

```
# calculating number of purchases for cohort and month
cohort = purchase.groupby(['first_order_month','month'])['revenue'].count().reset_index()
cohort.columns = ['first_order_month','month','orders']
```

In [41]:

```
# merge cohort with month_cohort to attain cohort_sizes
cohort = cohort.merge(cohort_sizes,on = ['first_order_month'])
cohort['age_month'] = ((cohort['month'] - cohort['first_order_month'])/np.timedelta64(1,'M'
cohort['orders_per_buyer'] = cohort['orders']/cohort['cohort_size']
cohort.head()
```

Out[41]:

	first_order_month	month	orders	cohort_size	age_month	orders_per_buyer
0	2017-06-01	2017-06-01	2353	2022	0.0	1.163699
1	2017-06-01	2017-07-01	177	2022	1.0	0.087537
2	2017-06-01	2017-08-01	171	2022	2.0	0.084570
3	2017-06-01	2017-09-01	224	2022	3.0	0.110781
4	2017-06-01	2017-10-01	291	2022	4.0	0.143917

In [42]:

```
cohort_piv=cohort.pivot_table(
    index='first_order_month',
    columns='age_month',
    values='orders_per_buyer',
    aggfunc='sum'
).cumsum(axis=1)

cohort_piv.round(2).fillna('')
```

Out[42]:

age_month	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0
first_order_month												
2017-06-01	1.16	1.25	1.34	1.45	1.59	1.7	1.83	1.92	2.03	2.1	2.15	2.19
2017-07-01	1.14	1.19	1.25	1.31	1.34	1.39	1.42	1.44	1.47	1.49	1.51	
2017-08-01	1.12	1.2	1.27	1.33	1.39	1.44	1.47	1.53	1.56	1.6		
2017-09-01	1.14	1.22	1.28	1.35	1.37	1.41	1.46	1.48	1.5			
2017-10-01	1.14	1.22	1.25	1.28	1.31	1.34	1.35	1.38				
2017-11-01	1.18	1.27	1.32	1.37	1.4	1.42	1.44					
2017-12-01	1.15	1.21	1.26	1.3	1.32	1.34						
2018-01-01	1.12	1.19	1.24	1.25	1.28							
2018-02-01	1.12	1.18	1.2	1.22								
2018-03-01	1.17	1.22	1.27									
2018-04-01	1.09	1.18										
2018-05-01	1.09											
2018-06-01	1.00											

The table above shows us the number of orders per month per customer for each cohort for each age month. This way we have attained so much useful information into one table.

Reviewer's comment

Perfect calculations

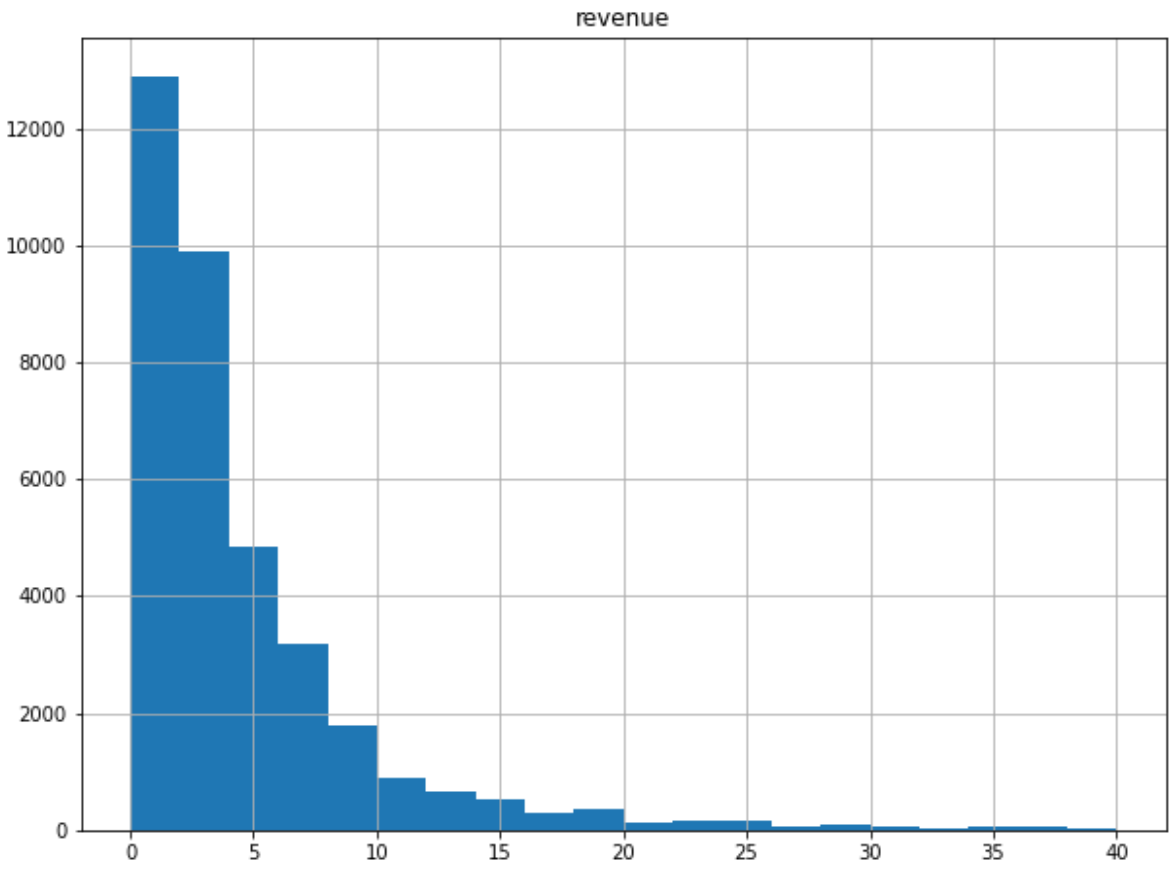
2.2.3 What is the average purchase size?

To calculate the average purchase size, we are going to do that by calculating the average purchase size per month per cohorts. That means, we are going to divide our customers into cohorts again, and calculate the monthly purchase size for each cohort lifetime.

First, lets have a look at the revenue distribution:

In [43]:

```
# distribution of revenue by uids
avg_check = purchase.groupby(['uid'])['revenue'].sum().reset_index()
fig = avg_check.hist("revenue", bins=20,figsize=(10,7.5), range=(0,40))
```



Now lets calculate the average purchase size: we'll group the revenue by cohort and month (lifetimes) and see what was the average purchase size for each lifetime of each cohort.

In [44]:

```
# grouping revenue by cohort and month
avg_cohort = purchase.groupby(['first_order_month','month'])['revenue'].mean().reset_index(
avg_cohort['age_month'] = ((avg_cohort['month'] - avg_cohort['first_order_month']) / np.tim
avg_cohort.head(24)
```

Out[44]:

	first_order_month	month	revenue	age_month
0	2017-06-01	2017-06-01	4.061832	0.0
1	2017-06-01	2017-07-01	5.547006	1.0
2	2017-06-01	2017-08-01	5.177427	2.0
3	2017-06-01	2017-09-01	8.621875	3.0
4	2017-06-01	2017-10-01	7.108522	4.0
5	2017-06-01	2017-11-01	6.825321	5.0
6	2017-06-01	2017-12-01	6.986545	6.0
7	2017-06-01	2018-01-01	6.761839	7.0
8	2017-06-01	2018-02-01	5.279009	8.0
9	2017-06-01	2018-03-01	8.009869	9.0
10	2017-06-01	2018-04-01	12.038125	10.0
11	2017-06-01	2018-05-01	6.042093	11.0
12	2017-07-01	2017-07-01	5.289542	0.0
13	2017-07-01	2017-08-01	6.446100	1.0
14	2017-07-01	2017-09-01	9.992083	2.0
15	2017-07-01	2017-10-01	6.637596	3.0
16	2017-07-01	2017-11-01	4.721806	4.0
17	2017-07-01	2017-12-01	3.659512	5.0
18	2017-07-01	2018-01-01	3.789508	6.0
19	2017-07-01	2018-02-01	5.454400	7.0
20	2017-07-01	2018-03-01	5.345690	8.0
21	2017-07-01	2018-04-01	11.790000	9.0
22	2017-07-01	2018-05-01	5.648302	10.0
23	2017-08-01	2017-08-01	4.718557	0.0

In [45]:

```
avg_cohort_piv=avg_cohort.pivot_table(
    index='first_order_month',
    columns='age_month',
    values='revenue',
    aggfunc='mean'
)
avg_cohort_piv.round(2).fillna('')
```

Out[45]:

age_month	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.(
first_order_month												
2017-06-01	4.06	5.55	5.18	8.62	7.11	6.83	6.99	6.76	5.28	8.01	12.04	6.0
2017-07-01	5.29	6.45	9.99	6.64	4.72	3.66	3.79	5.45	5.35	11.79	5.65	
2017-08-01	4.72	5.99	6.28	6.62	7.96	6.27	5.89	7.11	8.7	5.6		
2017-09-01	4.97	13.23	8.38	62.57	15.43	15.32	16.92	11.21	7.79			
2017-10-01	4.37	7.41	5.13	5.59	5.1	5.07	4.28	4.01				
2017-11-01	4.38	4.17	4.47	6.28	4.47	3.73	4.6					
2017-12-01	4.11	4.23	20.07	26.08	15.95	14.11						
2018-01-01	3.69	4.44	6.45	7.52	2.71							
2018-02-01	3.71	4.58	3.45	3.87								
2018-03-01	4.14	5.97	6.33									
2018-04-01	4.26	6.2										
2018-05-01	4.29											
2018-06-01	3.42											

Looking at the table, averages look very similar through the different cohorts and lifetimes. One thing that we notice, is the excpetional averages for the Sep 2017 cohort, what could be the reason for this very higher average? With a simple test below, we can see what's so special about this cohort:

In [46]:

```
# displaying info about Sep and August 2017 cohorts and the rest of the df to compare
display(purchase[purchase['first_order_month'] == '2017-09-01']['revenue'].describe())
display(purchase[purchase['first_order_month'] == '2017-08-01']['revenue'].describe())
display(purchase['revenue'].describe())
```

```
count    3873.000000
mean        8.952419
std        53.042118
min         0.030000
25%         1.220000
50%         2.570000
75%         6.110000
max      2633.280000
Name: revenue, dtype: float64

count    2187.000000
mean        5.306932
std         7.110621
min         0.060000
25%         1.530000
50%         3.050000
75%         6.110000
max        97.780000
Name: revenue, dtype: float64

count    50364.000000
mean         5.004710
std         21.828823
min          0.010000
25%          1.220000
50%          2.500000
75%          4.890000
max       2633.280000
Name: revenue, dtype: float64
```

What we can see here is; the reason the Sep 2017 cohort had exceptionally high averages (higher than the rest of the cohorts) was due to having orders with really high revenue, that pushed the average higher.

Reviewer's comment

You are right again👍

2.2.4 How much money does the people bring? (LTV - Lifetime value)

To find out, yet again, we'll use cohorts, based on the month of the first order of people and the month of the order itself to calculate the cumulative value each cohort has brought to the business

In [47]:

```
#get the revenue per cohort in each month
ltv_cohort=purchase.groupby(['first_order_month','month'])['revenue'].sum().reset_index()
ltv_cohort.columns = ['first_order_month','month','revenue']
#merge with the cohort size
ltv_cohort=ltv_cohort.merge(cohort_sizes,on=['first_order_month'])
ltv_cohort['age']=((ltv_cohort['month'] - ltv_cohort['first_order_month']) / np.timedelta64
ltv_cohort['ltv']=ltv_cohort['revenue']/ltv_cohort['cohort_size']
ltv_cohort
```

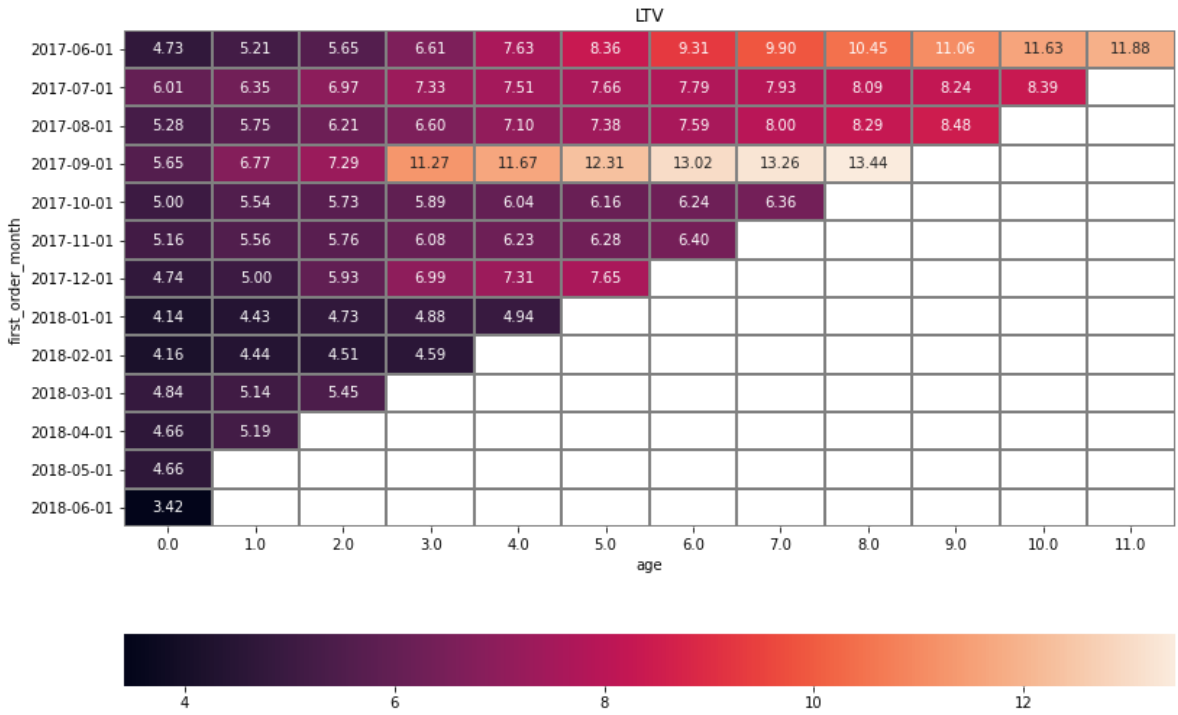
Out[47]:

	first_order_month	month	revenue	cohort_size	age	ltv
0	2017-06-01	2017-06-01	9557.49	2022	0.0	4.726751
1	2017-06-01	2017-07-01	981.82	2022	1.0	0.485569
2	2017-06-01	2017-08-01	885.34	2022	2.0	0.437854
3	2017-06-01	2017-09-01	1931.30	2022	3.0	0.955143
4	2017-06-01	2017-10-01	2068.58	2022	4.0	1.023037
...	...	...	...	...	...	...
74	2018-03-01	2018-05-01	1114.87	3534	2.0	0.315470
75	2018-04-01	2018-04-01	10600.69	2274	0.0	4.661693
76	2018-04-01	2018-05-01	1209.92	2274	1.0	0.532067
77	2018-05-01	2018-05-01	13925.76	2986	0.0	4.663684
78	2018-06-01	2018-06-01	3.42	1	0.0	3.420000

79 rows × 6 columns

In [48]:

```
ltv_cohort_piv=ltv_cohort.pivot_table(
    index='first_order_month',
    columns='age',
    values='ltv',
    aggfunc='sum'
).cumsum(axis=1)
plt.figure(figsize=(13, 9))
ltv_cohort_piv.index=ltv_cohort_piv.index.astype(str)
sns.heatmap(ltv_cohort_piv, annot=True, fmt='.2f', linewidths=1, linecolor='grey', cbar_kws
            ).set(title ='LTV')
plt.show()
```



Above, we can see similar results to those we had on the 'average purchase size' section. But here what we actually see is the average LTV per customer in each cohort. Some cohorts were more successful than other cohorts, an example of such cohorts are the Sep 2017 and Dec 2017 cohorts, what could be the reasons for a certain cohort to be more profitable than other cohorts? This could be totally coincidenctal or there could actually be a real reason, to find out, we have to do further studying.

2.2.5 To sum up

- In this section, we calculate the conversion rate, which turned out to be 16%. Meaning, 16% of visitors make their first order and become customers. Is this enough? We can only find out after we study the expenses, afterall, to decide if that's enough or not, we should at least be gaining as much as we're losing.
- We also calculated the average number of orders each cohort makes per month. What we got was pretty reasonable, each cohort had the most number of orders at the first lifetime (**age_month**) and the rest of the lifetimes had significantly lower order rates. This is very reasonable because the retention rates were very low (varying from 4% to 7%) and with only this percentage coming back month after month, order numbers will accordingly be lower.
- Away from customers and orders, we then start talking about what really matters, **Revenue**. Most of our revenue comes from low-cost orders (from 0 to 10 dollars). Among the different cohorts, we had the successful cohorts, like the first, forth and seventh cohorts. And the less successful cohorts, like the 5th

cohort or the 8th cohort. What do we learn from this? We should study the successful cohorts, and determine what led to their success, if we want more revenue, maybe we can find something.

Reviewer's comment

And here, everything is perfect :)

2.3 Marketing

Now that we are done talking about revenue from customers, its time to talk about costs and expenses. Afterall, that's the main topic of the project, how can we optimize these expenses, to maximize the winnings.

2.3.1 How much money was spent? Overall/per source/over time

Lets start by calculating how much expenses we had;

First, overall expenses:

In [49]:

```
# how much money was spent (Overall)
display(costs['costs'].sum())
display(orders['revenue'].sum())
```

```
329131.62
```

```
252057.19999999998
```

In [50]:

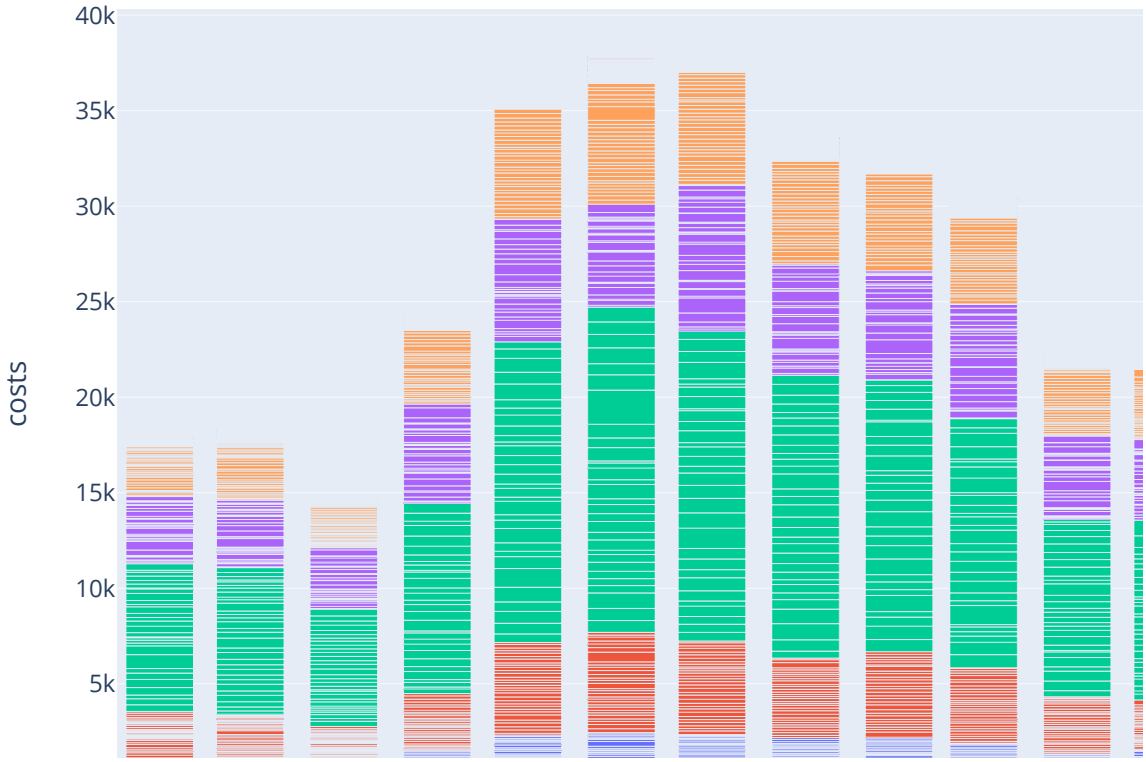
```
# how much money was spent (Per source)
costs_per_source = costs.groupby('source_id')['costs'].sum()
costs_per_source
```

Out[50]:

```
source_id
1      20833.27
2      42806.04
3     141321.63
4      61073.60
5      51757.10
9       5517.49
10      5822.49
Name: costs, dtype: float64
```

In [51]:

```
# adding a month column to costs df
costs['month'] = costs['dt'].astype('datetime64[M]')
costs_per_month = costs.groupby('month')['costs'].sum().reset_index()
costs_per_month

fig = px.bar(costs, x="month", y="costs", color='source_id')
fig.show()
```



Reviewer's comment

Let's try to create a stacked plot by month. It will be more understandable here😉

Student answer.

This defintely describes the costs of each month, by each source in a more elegant and readable way. Thanks for the tip.

CORRECTED

Reviewer's comment 2

Here is an error :(

Student answer.

I am really sorry, such a newbie mistake. I ran the whole project then made the necessary changes therefore the code worked for me and I was able to see the graph of costs per months. (Since I added a 'month' column to the costs DF later in the project)

Reviewer's comment 3

Everything is fine now!

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_53/3594570279.py in <module>
----> 1 fig = px.bar(costs, x="month", y="costs", color='source_id')
      2 fig.show()

/opt/conda/lib/python3.9/site-packages/plotly/express/_chart_types.py in bar(data_frame, x, y, color, pattern_shape, facet_row,
facet_col, facet_col_wrap, facet_row_spacing, facet_col_spacing, hover_name, hover_data, custom_data, text, base, error_x, erro
r_x_minus, error_y, error_y_minus, animation_frame, animation_group, category_orders, labels, color_discrete_sequence, color_di
screte_map, color_continuous_scale, pattern_shape_sequence, pattern_shape_map, range_color, color_continuous_midpoint, opacity,
orientation, barmode, log_x, log_y, range_x, range_y, title, template, width, height)
    363     mark.
    364     """
--> 365     return make_figure(
    366         args=locals(),
    367         constructor=go.Bar,

/opt/conda/lib/python3.9/site-packages/plotly/express/_core.py in make_figure(args, constructor, trace_patch, layout_patch)
   1931     apply_default_cascade(args)
   1932
-> 1933     args = build_dataframe(args, constructor)
   1934     if constructor in [go.Treemap, go.Sunburst, go.Icicle] and args["path"] is not None:
   1935         args = process_dataframe_hierarchy(args)

/opt/conda/lib/python3.9/site-packages/plotly/express/_core.py in build_dataframe(args, constructor)
   1403     # now that things have been prepped, we do the systematic rewriting of `args`
   1404
-> 1405     df_output, wide_id_vars = process_args_into_dataframe(
   1406         args, wide_mode, var_name, value_name
   1407     )

/opt/conda/lib/python3.9/site-packages/plotly/express/_core.py in process_args_into_dataframe(args, wide_mode, var_name, value_
name)
   1205                         if argument == "index":
   1206                             err_msg += "\n To use the index, pass it in directly as `df.index`."
-> 1207                         raise ValueError(err_msg)
   1208                 elif length and len(df_input[argument]) != length:
   1209                     raise ValueError(

ValueError: Value of 'x' is not the name of a column in 'data_frame'. Expected one of ['source_id', 'dt', 'costs'] but receive
d: month
```

This graph looks similar to the DAU, WAU and MAU graphs we had earlier. How are they similar? they all peak in Nov 2017 (Aside from Nov, here it also peaks at Dec 2017. But this seems to have had less impact on DAU/MAU/WAU) . This actually explains everything, the reason DAU, MAU and WAU peaked at this period of time was due to the fact that the business had spent the highest amount of expenses at this time.

First thing we noticed was the green peak, source #3. Does over-spending on this source mean we are going to get more visitors and more buyers? is it the way to go? Not necessarily, to find out lets continue.

Second, over time expenses:

In [52]:

```
# grouping buyers per month by the first order month to get the size of each cohort
buyers_per_month = purchase.groupby(['first_order_month'])['uid'].nunique().reset_index()
buyers_per_month.columns = ['month','buyers']
buyers_per_month.head()
```

Out[52]:

	month	buyers
0	2017-06-01	2022
1	2017-07-01	1922
2	2017-08-01	1369
3	2017-09-01	2579
4	2017-10-01	4340

In [53]:

```
# merging the dfs we got above to calculate the CAC (customer acquistion cost)
CAC_per_month = costs_per_month.merge(buyers_per_month, how='left', on='month')
CAC_per_month['CAC'] = CAC_per_month['costs']/CAC_per_month['buyers']
CAC_per_month
```

Out[53]:

	month	costs	buyers	CAC
0	2017-06-01	18015.00	2022	8.909496
1	2017-07-01	18240.59	1922	9.490421
2	2017-08-01	14790.54	1369	10.803901
3	2017-09-01	24368.91	2579	9.448976
4	2017-10-01	36322.88	4340	8.369327
5	2017-11-01	37907.88	4078	9.295704
6	2017-12-01	38315.35	4380	8.747797
7	2018-01-01	33518.52	3373	9.937302
8	2018-02-01	32723.03	3651	8.962758
9	2018-03-01	30415.27	3534	8.606471
10	2018-04-01	22289.38	2274	9.801838
11	2018-05-01	22224.27	2986	7.442823

In [54]:

```
fig = px.line(CAC_per_month, x="month", y="CAC", title='CAC')
fig.show()
```

Reviewer's comment

Good

CAC is the result of dividing **costs** by **buyers**. This means, to decrease the CAC itself, we either need to have lower **costs** or higher **buyers**. Most of the cohorts had a similar cost/buyers ratio therefore we ended up with close CACs on most of the months. One way to optimize our expenses, is to follow the steps taken in cohorts with low CACs, this way we keep our costs low and keep getting the same number of buyers per cohort.

Third and finally, per source expenses:

In [55]:

```
# adding a first source (main source) column for each customer
first_source = visits.sort_values('start_ts').groupby('uid').first()['source_id'].reset_ind
first_source.columns = ['uid','first_source']
first_source.head()
```

Out[55]:

	uid	first_source
0	11863502262781	3
1	49537067089222	2
2	297729379853735	3
3	313578113262317	2
4	325320750514679	5

In [56]:

```
# merging the first_source df with purchase df
purchase = purchase.merge(first_source, on=['uid'],how='left')
purchase.head()
```

Out[56]:

	buy_ts	revenue	uid	first_order	first_session	conversion	first_order_r
0	2017-06-01 00:10:00	17.00	10329302124590727494	2017-06-01 00:10:00	2017-06-01 00:09:00	0	2017-
1	2017-06-01 00:25:00	0.55	11627257723692907447	2017-06-01 00:25:00	2017-06-01 00:14:00	0	2017-
2	2017-06-01 00:27:00	0.37	17903680561304213844	2017-06-01 00:27:00	2017-06-01 00:25:00	0	2017-
3	2017-06-01 00:29:00	0.55	16109239769442553005	2017-06-01 00:29:00	2017-06-01 00:14:00	0	2017-
4	2017-06-01 07:58:00	0.37	14200605875248379450	2017-06-01 07:58:00	2017-06-01 07:31:00	0	2017-

In [57]:

```
# grouping costs df by month and source id to get costs for each source by month
costs_by_month_source = costs.groupby(['month','source_id'])['costs'].sum().reset_index()
costs_by_month_source.head(20)
```

Out[57]:

	month	source_id	costs
0	2017-06-01	1	1125.61
1	2017-06-01	2	2427.38
2	2017-06-01	3	7731.65
3	2017-06-01	4	3514.80
4	2017-06-01	5	2616.12
5	2017-06-01	9	285.22
6	2017-06-01	10	314.22
7	2017-07-01	1	1072.88
8	2017-07-01	2	2333.11
9	2017-07-01	3	7674.37
10	2017-07-01	4	3529.73
11	2017-07-01	5	2998.14
12	2017-07-01	9	302.54
13	2017-07-01	10	329.82
14	2017-08-01	1	951.81
15	2017-08-01	2	1811.05
16	2017-08-01	3	6143.54
17	2017-08-01	4	3217.36
18	2017-08-01	5	2185.28
19	2017-08-01	9	248.93

In [58]:

```
# calculating how much buyers each source brought per month
buyers_per_month_source = purchase.groupby(['first_order_month','first_source'])['uid'].nun
buyers_per_month_source.columns = ['month','source_id','buyers']
buyers_per_month_source.head(24)
```

Out[58]:

	month	source_id	buyers
0	2017-06-01	1	190
1	2017-06-01	2	234
2	2017-06-01	3	638
3	2017-06-01	4	413
4	2017-06-01	5	384
5	2017-06-01	6	0
6	2017-06-01	7	0
7	2017-06-01	9	68
8	2017-06-01	10	95
9	2017-07-01	1	160
10	2017-07-01	2	208
11	2017-07-01	3	511
12	2017-07-01	4	517
13	2017-07-01	5	423
14	2017-07-01	6	0
15	2017-07-01	7	0
16	2017-07-01	9	52
17	2017-07-01	10	51
18	2017-08-01	1	113
19	2017-08-01	2	122
20	2017-08-01	3	337
21	2017-08-01	4	338
22	2017-08-01	5	360
23	2017-08-01	6	0

In [59]:

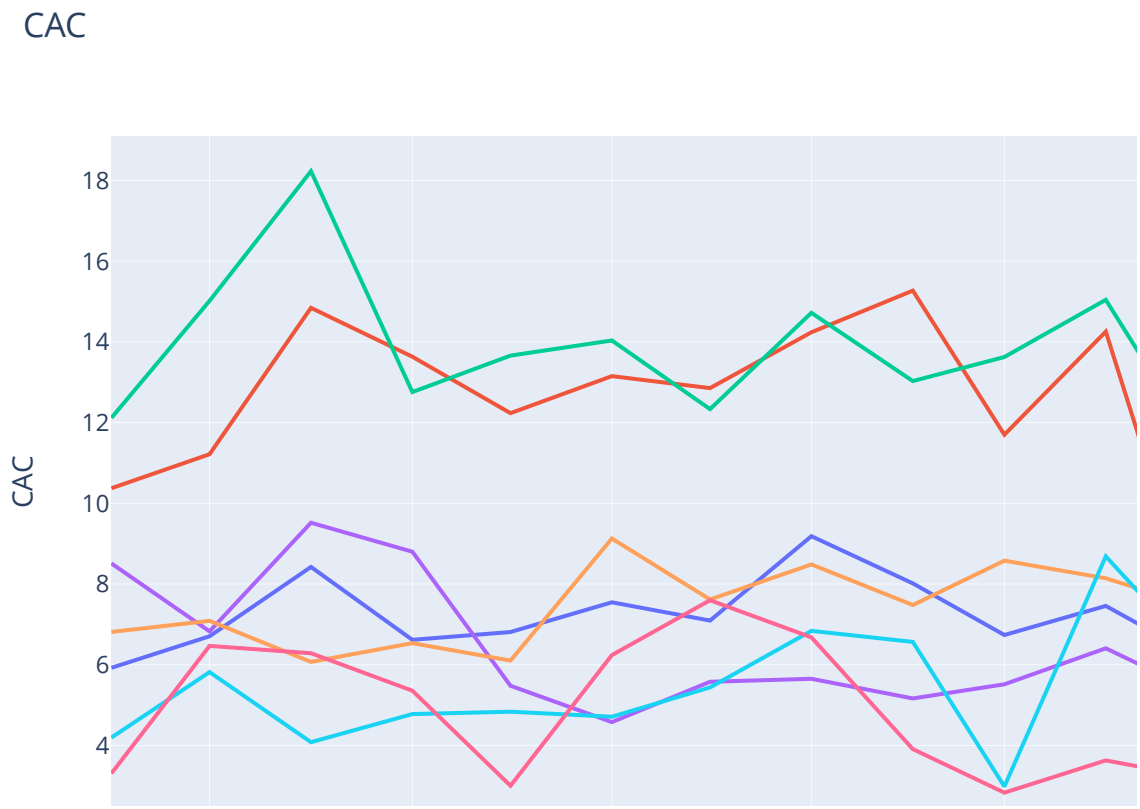
```
#actual cac calculations
CAC_per_month_source = costs_by_month_source.merge(buyers_per_month_source,how='left',on=['
CAC_per_month_source['CAC']=CAC_per_month_source['costs']/CAC_per_month_source['buyers']
CAC_per_month_source.head(25)
```

Out[59]:

	month	source_id	costs	buyers	CAC
0	2017-06-01	1	1125.61	190	5.924263
1	2017-06-01	2	2427.38	234	10.373419
2	2017-06-01	3	7731.65	638	12.118574
3	2017-06-01	4	3514.80	413	8.510412
4	2017-06-01	5	2616.12	384	6.812813
5	2017-06-01	9	285.22	68	4.194412
6	2017-06-01	10	314.22	95	3.307579
7	2017-07-01	1	1072.88	160	6.705500
8	2017-07-01	2	2333.11	208	11.216875
9	2017-07-01	3	7674.37	511	15.018337
10	2017-07-01	4	3529.73	517	6.827331
11	2017-07-01	5	2998.14	423	7.087801
12	2017-07-01	9	302.54	52	5.818077
13	2017-07-01	10	329.82	51	6.467059
14	2017-08-01	1	951.81	113	8.423097
15	2017-08-01	2	1811.05	122	14.844672
16	2017-08-01	3	6143.54	337	18.230089
17	2017-08-01	4	3217.36	338	9.518817
18	2017-08-01	5	2185.28	360	6.070222
19	2017-08-01	9	248.93	61	4.080820
20	2017-08-01	10	232.57	37	6.285676
21	2017-09-01	1	1502.01	227	6.616784
22	2017-09-01	2	2985.66	219	13.633151
23	2017-09-01	3	9963.55	781	12.757426
24	2017-09-01	4	5192.26	590	8.800441

In [60]:

```
#plotting cac dynamics
fig = px.line(CAC_per_month_source, x="month", y="CAC",color='source_id',title='CAC')
fig.show()
```



Judging by the graph above, our first suggestion to optimize expenses would be to give up the sources with ids 2 and 3. As we look at the graph, we can see that the sources 2 and 3 had the highest CACs. What this means? This means that's these sources performed slower than the rest of sources. We basically spent too much money to only get less buyers than we usually get with the other sources. Therefore, we should work on these sources or just give them up entierly.

Reviewer's comment

Perfect

2.3.2 How worthwhile where the investments? (ROI)

Finally, we are at that point where we want to find out if we are **winning**, if the business is worth it and if it is a healthy business. We are at the point where we calculate the ROI/ROMI.

We are going to calculate ROI per cohorts, find out which cohorts were worthwhile and payed off and which aren't/still arent.

In [61]:

```
CAC_per_month_ROI = CAC_per_month[['month','CAC']]
CAC_per_month_ROI.columns = ['first_order_month','CAC']
ROI = ltv_cohort.merge(CAC_per_month_ROI,on = ['first_order_month'], how = 'left')
ROI.head()
```

Out[61]:

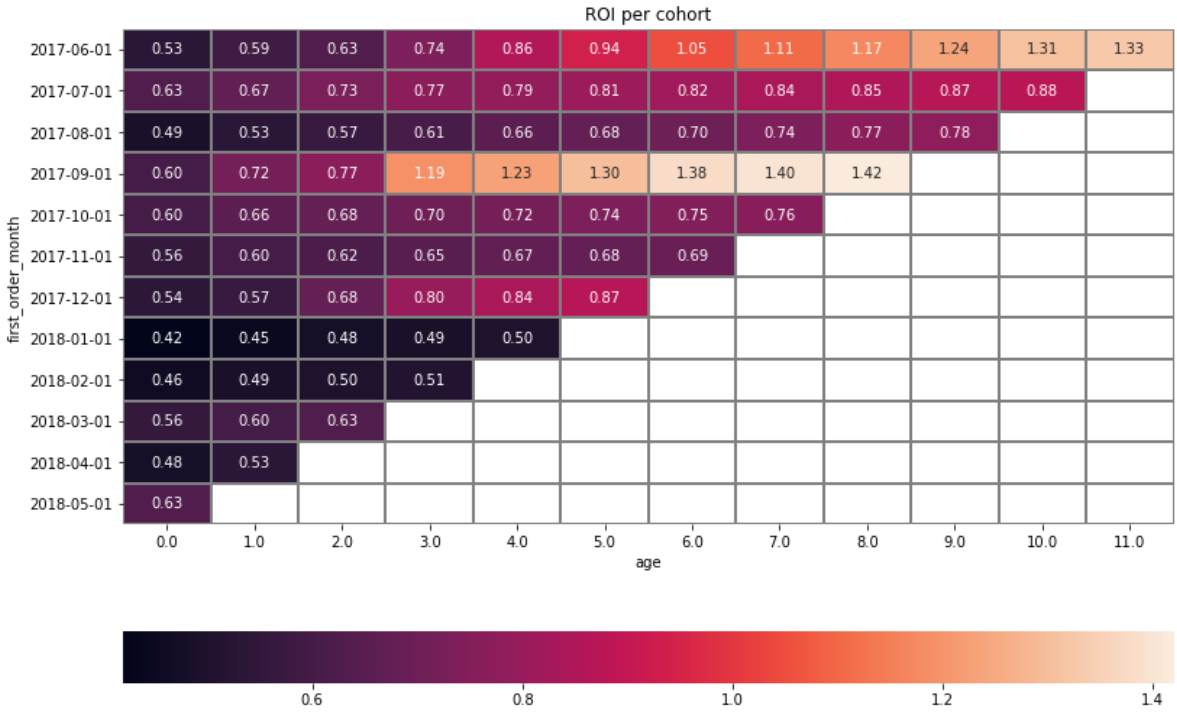
	first_order_month	month	revenue	cohort_size	age	ltv	CAC
0	2017-06-01	2017-06-01	9557.49	2022	0.0	4.726751	8.909496
1	2017-06-01	2017-07-01	981.82	2022	1.0	0.485569	8.909496
2	2017-06-01	2017-08-01	885.34	2022	2.0	0.437854	8.909496
3	2017-06-01	2017-09-01	1931.30	2022	3.0	0.955143	8.909496
4	2017-06-01	2017-10-01	2068.58	2022	4.0	1.023037	8.909496

In [62]:

```
ROI['ROI'] = ROI['ltv']/ROI['CAC']
roi_piv = ROI.pivot_table(
    index = 'first_order_month', columns = 'age', values = 'ROI', aggfunc = 'mean'
).cumsum(axis=1).round(2)
```

In [63]:

```
roi_piv.index = roi_piv.index.astype(str)
plt.figure(figsize = (13, 9))
sns.heatmap(roi_piv, annot = True, fmt = '.2f', linewidths = 1, linecolor = 'grey', cbar_kw
            ).set(title = 'ROI per cohort')
plt.show()
```



To compare between cohorts, we look at how well they're doing in similar lifetimes. For instance, the first cohort wasn't paying off in the first 4 months. while on the other hand, the forth cohort has already payed off on the 4th

month. This means that the forth cohort was more worthwhile than the first cohort.

In my opinion, any cohorts that has payed off within the given period (10-12 lifetimes) of time by at least 5% to 10% is a worthwhile cohort.

Now that we calculate ROI per cohort, lets continue and calculate the ROI per source, since the sources are our investments afterall:

In [64]:

```
ltv_per_source = purchase.groupby(['first_source'])['uid','revenue'].agg({'uid':'nunique','
ltv_per_source.columns = ['source_id','buyers','revenue']
ltv_per_source['ltv'] = ltv_per_source['revenue']/ltv_per_source['buyers']
ltv_per_source
```

Out[64]:

	source_id	buyers	revenue	ltv
0	1	2896	31090.55	10.735687
1	2	3503	46923.61	13.395264
2	3	10467	54511.24	5.207914
3	4	10296	56696.83	5.506685
4	5	6931	52624.02	7.592558
5	6	0	0.00	NaN
6	7	1	1.22	1.220000
7	9	1088	5759.40	5.293566
8	10	1327	4450.33	3.353677

In [65]:

```
roi_per_source=costs_by_month_source.merge(ltv_per_source,on=['source_id'])
roi_per_source['cac']=roi_per_source['costs']/roi_per_source['buyers']
roi_per_source['romi']=roi_per_source['ltv']/roi_per_source['cac']
roi_per_source
```

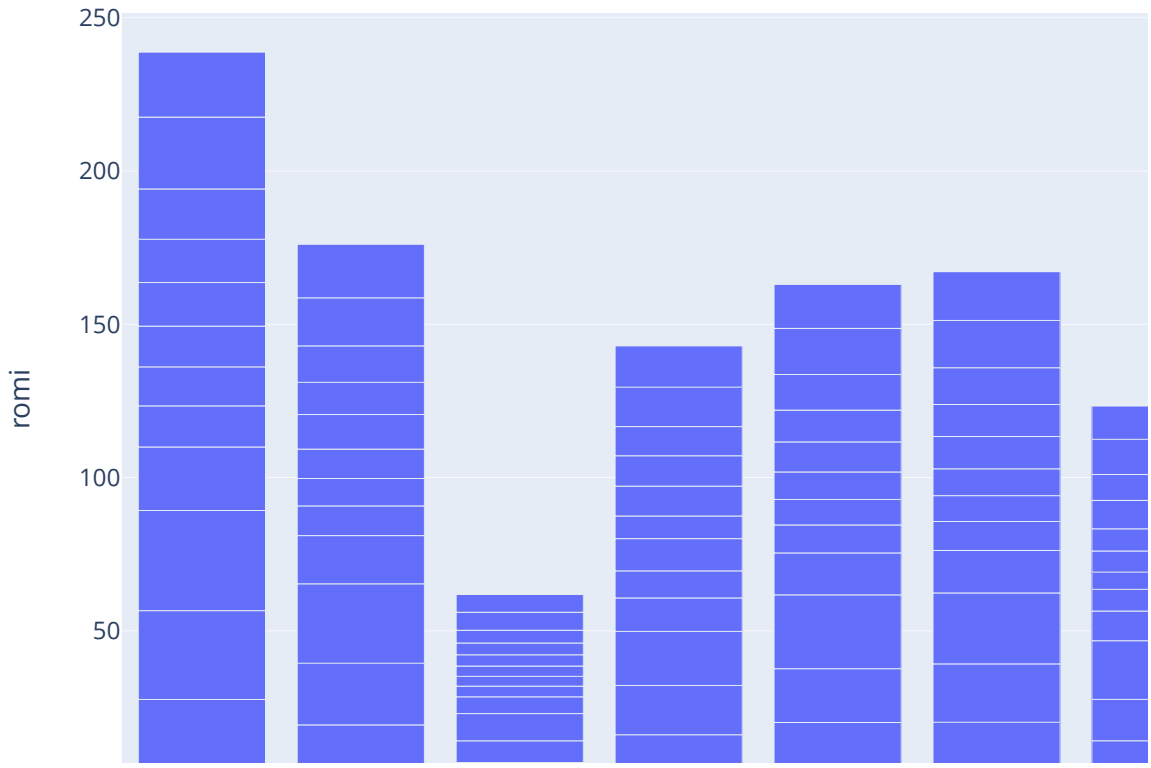
Out[65]:

	month	source_id	costs	buyers	revenue	ltv	cac	romi
0	2017-06-01	1	1125.61	2896	31090.55	10.735687	0.388677	27.621068
1	2017-07-01	1	1072.88	2896	31090.55	10.735687	0.370470	28.978590
2	2017-08-01	1	951.81	2896	31090.55	10.735687	0.328664	32.664660
3	2017-09-01	1	1502.01	2896	31090.55	10.735687	0.518650	20.699296
4	2017-10-01	1	2315.75	2896	31090.55	10.735687	0.799637	13.425694
...	...	...	...	...	...	...	...	...
79	2018-01-01	10	614.35	1327	4450.33	3.353677	0.462962	7.243965
80	2018-02-01	10	480.88	1327	4450.33	3.353677	0.362381	9.254554
81	2018-03-01	10	526.41	1327	4450.33	3.353677	0.396692	8.454114
82	2018-04-01	10	388.25	1327	4450.33	3.353677	0.292577	11.462537
83	2018-05-01	10	409.86	1327	4450.33	3.353677	0.308862	10.858171

84 rows × 8 columns

In [66]:

```
fig = px.bar(roi_per_source, x='source_id', y='romi')
fig.update_xaxes(type='category')
fig.show()
```



2.3.3 Conclusion

This graph is another proof that source #3 was a big failure as an investiment. But what we can also obtain from this graph is that the first source (source_id = 1) had the highest ROI/ROMI, meaning it was the most worthy of them all.

Reviewer's comment

Great work

2.4 General Conclusion

We have dug up and dived in our datasets and ended up with enough information and conclusions to help us optimize our expenses.

- First, we studied our product. How many people use it and the retention rate. At this point, we can't reach to much conclusions, we are basically getting familiar with our research subject. We've encounted some

weird cases that sparked some questions, questions that we'll answer in the next sections.
- Highlights: We had visitor count peaks at Nov of 2017 and session lengths peak was from 5 to 10 minutes.

- Second, we studied our sales. After we encountered the questions from section 1, we had to find some answers. At this we calculated the conversion rate and revenue. We are not interested in visitors but rather in customers.
- Highlights: The conversion rate was at 16% (16% of visitors become customers), The retention rates weren't among the highest (ranging from 4% to 7%) and most of our revenue was from orders ranging from 0 to 10 dollars.
- Finally, we studing our expenses. How much money did we spend and on what, when did it pay off and what sources were the most successful.
- Highlights: We had cohorts that were more successful than other, paying off early, way earlier than other cohorts (cohorts 1,4 and 7 for instance). Similarly, we had successful sources and less succesful sources. Source #1 had the highest ROMI while the source #3 had the lowest ROMI.

To sum up: In order to optimize our expenses, I advice the experts to give up on sources that scored low in ROMI. Such sources is source #3. Despite the fact that it brought the most visitors, but not a high percent of these visitors made their first order, and those who made their orders weren't enough to pay off for the expenses of this investment, maybe this source just didn't target the right audience. Similarlly, I advice the expers to spend more money on souce #1 and #2, for they were the most successful sources in terms of bringing revenue and buyers. One more advice that I want to give to our experts is to pay attention to the successful cohorts, learn why they succeed and determine the factors that contributed to their success and apply what they learnt on the new cohorts. Beside that, I also advice them on investing on new sources, sources like #6, #7 and #8. Those sources proved to bring revenue even without spending a dime on them, I think that they deserve a shot.

That's it and that's all, thanks for reading. I hope I was helpful enough!

Reviewer's comment

Structured and informative general conclusion👍

Reviewer's conclusion

Ameer, thank you for your project! You have great knowledge in visualization and analysis. I wrote some tips to improve your work. I will be waiting for your project😊

Student answer.

Thanks for the tips and thanks for the feedback ofcourse!

Reviewer's comment 2

You are welcome😉

In [ ]:

```
```