Project: User Churn Project (Waze app)

Introduction /Overview:

Waze app is a subsidiary company of Google that provides satellite navigation software on smartphones and other computers that support the Global Positioning System.

This project aimed at increasing overall growth by preventing monthly user churn on the app. The purpose of this project is to find factors that drive user churn and The goal is to build a model to predict whether or not a Waze user is retained or churned.

The Project milestones' :

1. Explore and analyze Waze's user data.
2. Data cleaning and EDA.
3. Create data visualizations.
4. Conduct a hypothesis test.
5. Binomial logistic regression model.
6. Build and test two tree-based models: random forest and XGBoost.

Data Source:

This project uses a dataset called waze_dataset.csv. It contains synthetic data created for this project in partnership with Waze. MetaData: The dataset contains: 14,999 rows – each row represents one unique user 12 columns

Column name	Type	Description
label	obj	Binary target variable ("retained" vs "churned") for if a user has churned anytime during the course of the month
sessions	int	The number of occurrence of a user opening the app during the month
drives	int	An occurrence of driving at least 1 km during the month
device	obj	The type of device a user starts a session with
total_sessions	float	A model estimate of the total number of sessions since a user has onboarded
n_days_after_onboarding	int	The number of days since a user signed up for the app
total_navigations_fav1	int	Total navigations since onboarding to the user's favorite place 1
total_navigations_fav2	int	Total navigations since onboarding to the user's favorite place 2
driven_km_drives	float	Total kilometers driven during the month
duration_minutes_drives	float	Total duration driven in minutes during the month
activity_days	int	Number of days the user opens the app during the month
driving_days	int	Number of days the user drives (at least 1 km) during the month

Objective:

The goal is to build a model that predicts whether a Waze user will be retained or will churn, and to identify the factors contributing to user churn, thereby aiding in the improvement of retention strategies.

Setting up the environment, importing packages and load the dataset :

In [2]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
```

In [3]:
```
## Read in the data and store it as a dataframe object called data.
data = pd.read_csv('C:/Users/engmo/OneDrive/Desktop/Google Advanced Dat
```

EDA and Data cleaning

In [4]:
```
## Explore the Data
data.head()
```

Out[4]:

	ID	label	sessions	drives	total_sessions	n_days_after_onboarding	total_navigations_fa
0	0	retained	283	226	296.748273	2276	2
1	1	retained	133	107	326.896596	1225	
2	2	retained	114	95	135.522926	2651	
3	3	retained	49	40	67.589221	15	3
4	4	retained	84	68	168.247020	1562	'

In [5]:
```
##Lets understand the data and the data type
print(data.shape)
print(data.info())
```

```
(14999, 13)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14999 entries, 0 to 14998
Data columns (total 13 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   ID                       14999 non-null  int64  
 1   label                    14299 non-null  object 
 2   sessions                 14999 non-null  int64  
 3   drives                   14999 non-null  int64  
 4   total_sessions           14999 non-null  float64
 5   n_days_after_onboarding  14999 non-null  int64  
 6   total_navigations_fav1   14999 non-null  int64  
 7   total_navigations_fav2   14999 non-null  int64  
 8   driven_km_drives         14999 non-null  float64
 9   duration_minutes_drives  14999 non-null  float64
 10  activity_days            14999 non-null  int64  
 11  driving_days             14999 non-null  int64  
 12  device                   14999 non-null  object 
dtypes: float64(3), int64(8), object(2)
memory usage: 1.5+ MB
None
```

In [6]:
```
## let's generate a statistics summary of the data.
data.describe()
```

Out[6]:

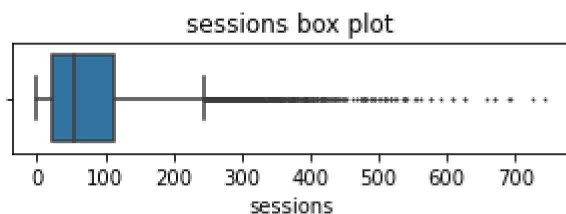
	ID	sessions	drives	total_sessions	n_days_after_onboarding	to
count	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000	
mean	7499.000000	80.633776	67.281152	189.964447	1749.837789	
std	4329.982679	80.699065	65.913872	136.405128	1008.513876	
min	0.000000	0.000000	0.000000	0.220211	4.000000	
25%	3749.500000	23.000000	20.000000	90.661156	878.000000	
50%	7499.000000	56.000000	48.000000	159.568115	1741.000000	
75%	11248.500000	112.000000	93.000000	254.192341	2623.500000	
max	14998.000000	743.000000	596.000000	1216.154633	3500.000000	

In [7]:
```
## Lets look for data missing and outliers, if any.
data.isna().sum()
```

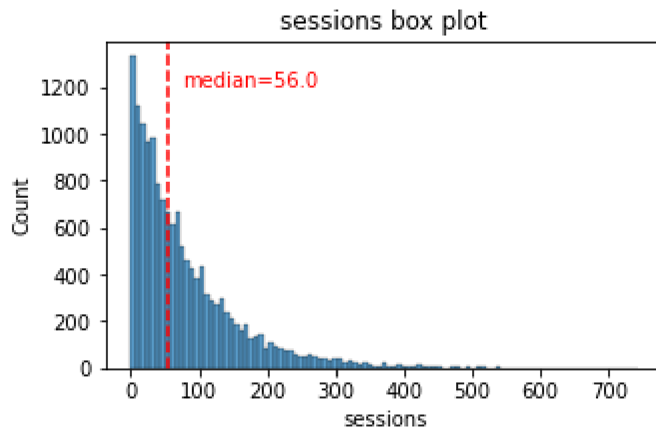
Out[7]:
```
ID                         0
label                    700
sessions                   0
drives                     0
total_sessions             0
n_days_after_onboarding    0
total_navigations_fav1     0
total_navigations_fav2     0
driven_km_drives           0
duration_minutes_drives    0
activity_days              0
driving_days               0
device                     0
dtype: int64
```

In [42]:
```
## Let's drop the ID column, its not usable
data = data.drop("ID",axis=1)
```

In [8]:
```
## The label which is the outcome variable is missing 700 enteries, let
## Let's check for outliers.
## Sessions are the numbers of occurrences of a user opening the app du
# Box plot.
plt.figure(figsize=(5,1))
sns.boxplot(x=data['sessions'],fliersize=1)
plt.title('sessions box plot')
plt.show()
```

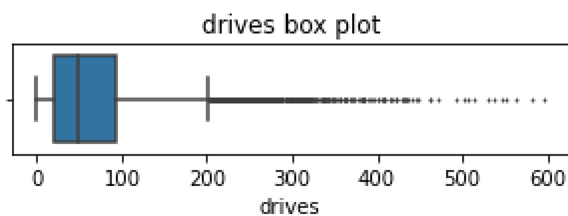


In [9]:
```
## Let's visulaize the same varibale in histogram to understand the dat
# Histogram
plt.figure(figsize=(5,3))
sns.histplot(x=data['sessions'])
median = data['sessions'].median()
plt.axvline(median, color='red', linestyle='--')
plt.text(75,1200, 'median=56.0', color='red')
plt.title('sessions box plot');
```



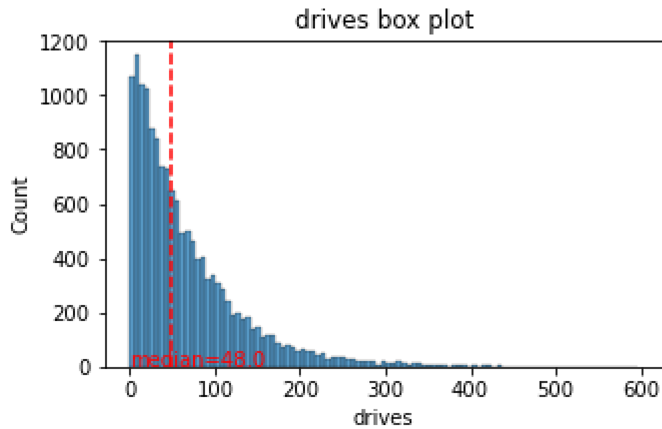
The sessions variable is a right-skewed distribution with half of the observations having 56 or fewer sessions. However, as indicated by the boxplot, some users have more than 700.

In [10]:
```
## Let's also review drives.
# Box plot
plt.figure(figsize=(5,1))
sns.boxplot(x=data['drives'], fliersize=1)
plt.title('drives box plot');
```



In [11]:

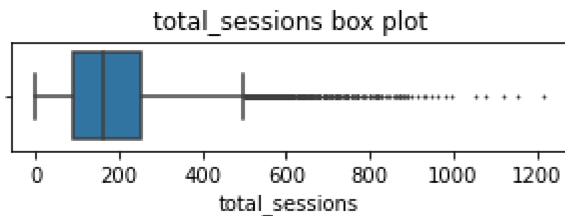
```
# Histogram
plt.figure(figsize=(5,3))
sns.histplot(x=data['drives'])
median = data['drives'].median()
plt.axvline(median, color='red', linestyle='--')
plt.text(0.25,0.95 ,'median=48.0', color='red')
plt.title('drives box plot');
```



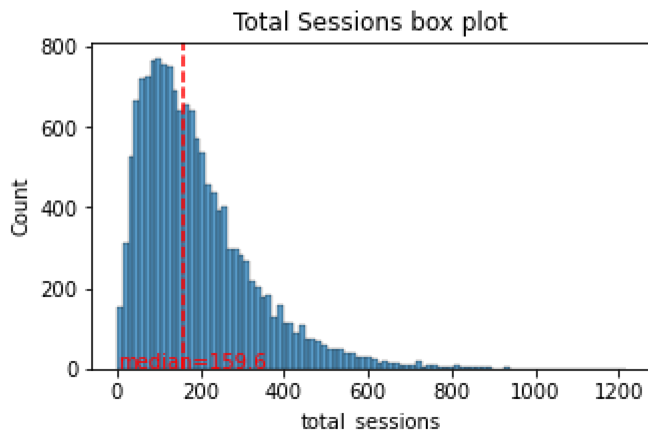
The drives information follows a distribution similar to the sessions variable. It is right-skewed, approximately log-normal, with a median of 48. However, some drivers had over 400 drives in the last month.

In [12]:

```
## Let's review the total sessions.
plt.figure(figsize=(5,1))
sns.boxplot(x=data['total_sessions'], fliersize=1)
plt.title('total_sessions box plot');

```

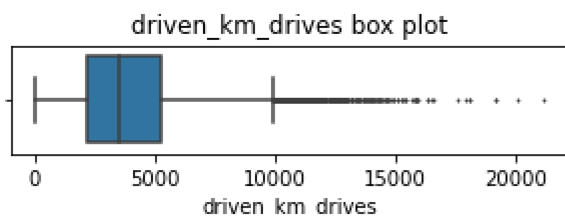


In [13]:
```
# Histogram
plt.figure(figsize=(5,3))
sns.histplot(x=data['total_sessions'])
median = data['total_sessions'].median()
plt.axvline(median, color='red', linestyle='--')
plt.text(0.25,0.85,'median=159.6', color='red',ha='left')
plt.title('Total Sessions box plot');
```

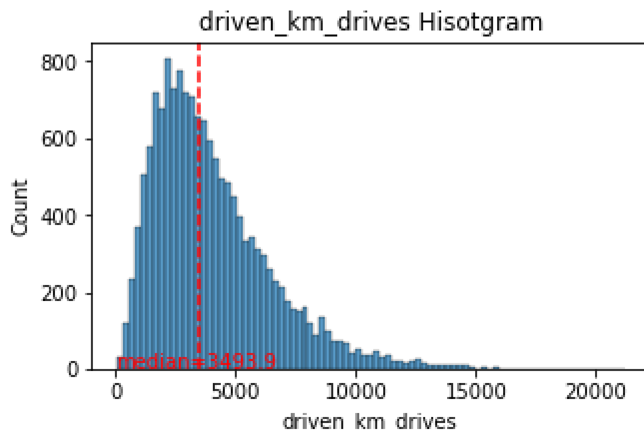


The `total_sessions` is a right-skewed distribution. The median total number of sessions is 159.6. This is interesting information because, if the median number of sessions in the last month was 56 and the median total sessions was ~160, then it seems that a large proportion of a user's (estimated) total drives might have taken place in the last month. This is something we can examine more closely later.

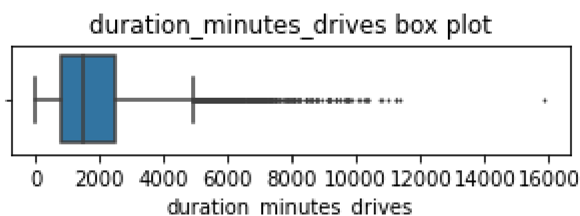
In [14]:
```
## Let's review the driven_km_drive total KM driven during the month.
# Box plot
plt.figure(figsize=(5,1))
sns.boxplot(x=data['driven_km_drives'], fliersize=1)
plt.title('driven_km_drives box plot');
```



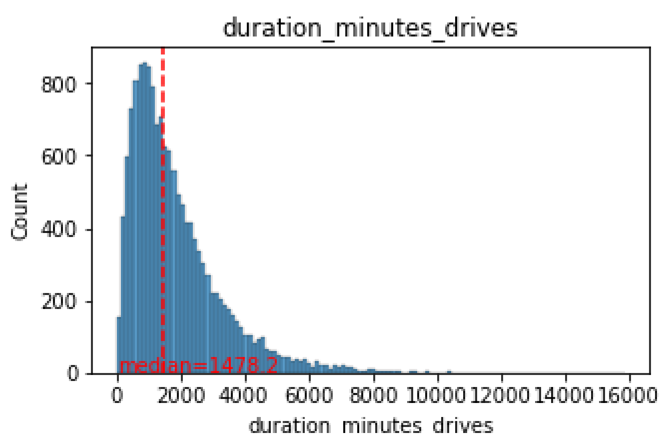
In [15]:
```
# Histogram
plt.figure(figsize=(5,3))
sns.histplot(x=data['driven_km_drives'])
median = data['driven_km_drives'].median()
plt.axvline(median, color='red', linestyle='--')
plt.text(0.25,0.85,'median=3493.9', color='red',ha='left')
plt.title('driven_km_drives Hisotgram');
```



In [16]:
```
## duration_minutes_drives
# Box plot
plt.figure(figsize=(5,1))
sns.boxplot(x=data['duration_minutes_drives'], fliersize=1)
plt.title('duration_minutes_drives box plot');
```

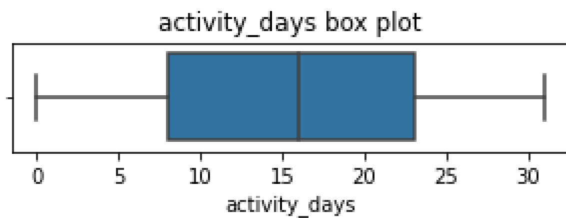


In [17]:
```
# Histogram
plt.figure(figsize=(5,3))
sns.histplot(x=data['duration_minutes_drives'])
median = data['duration_minutes_drives'].median()
plt.axvline(median, color='red', linestyle='--')
plt.text(0.85,0.85,'median=1478.2', color='red')
plt.title('duration_minutes_drives');
```

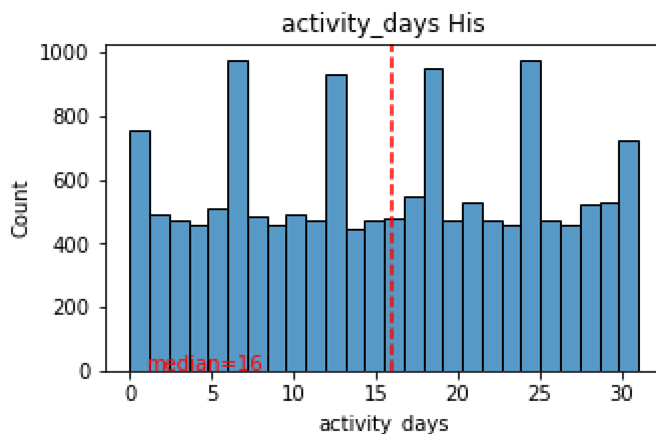


The `duration_minutes_drives` variable has a heavily skewed right tail. Half of the users drove less than ~1,478 minutes (~25 hours), but some users clocked over 250 hours over the month.

In [18]:
```
#### activity_days.
# Box plot
plt.figure(figsize=(5,1))
sns.boxplot(x=data['activity_days'], fliersize=1)
plt.title('activity_days box plot');
```



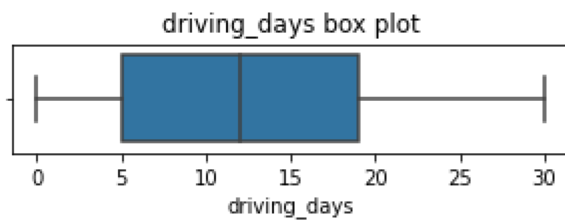
In [19]:
```
# Histogram
plt.figure(figsize=(5,3))
sns.histplot(x=data['activity_days'])
median = data['activity_days'].median()
plt.axvline(median, color='red', linestyle='--')
plt.text(0.95,0.25,'median=16', color='red')
plt.title('activity_days His');
```



Within the last month, users opened the app a median of 16 times. The box plot reveals a centered distribution. The histogram shows a nearly uniform distribution of ~500 people opening the app on each count of days. This distribution is noteworthy because it does not mirror the `sessions` distribution, which you might think would be closely correlated with `activity_days`.

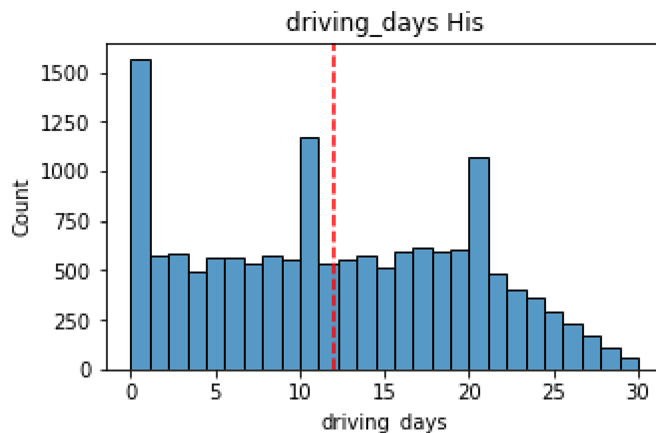
In [20]:

```
####driving_days:Number of days the user drives (at least 1 km) during
# Box plot
plt.figure(figsize=(5,1))
sns.boxplot(x=data['driving_days'], fliersize=1)
plt.title('driving_days box plot');

```



In [21]:

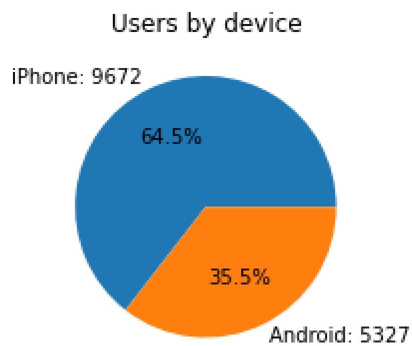
```
# Histogram
plt.figure(figsize=(5,3))
sns.histplot(x=data['driving_days'])
median = data['driving_days'].median()
plt.axvline(median, color='red', linestyle='--')

plt.title('driving_days His');
```



The number of days users drove each month is almost uniform, and it largely correlates with the number of days they opened the app that month, except the `driving_days` distribution tails off on the right.

In [22]:

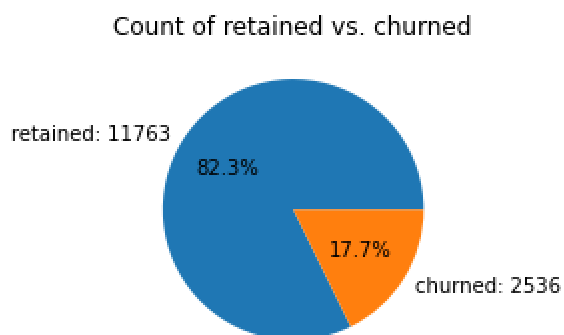
```
##device(Android/Iphone):
##The type of device a user starts a session with This is a categorical
# Pie chart
fig = plt.figure(figsize=(3,3))
dat=data['device'].value_counts()
plt.pie(dat,
        labels=[f'{dat.index[0]}: {dat.values[0]}',
                f'{dat.index[1]}: {dat.values[1]}'],
        autopct='%1.1f%%'
        )
plt.title('Users by device');
```



There are nearly twice as many iPhone users as Android users represented in this data.

In [23]:

```
##label:Binary target variable (“retained” vs “churned”)
fig = plt.figure(figsize=(3,3))
Mega=data['label'].value_counts()
plt.pie(Mega,
        labels=[f'{Mega.index[0]}: {Mega.values[0]}',
                f'{Mega.index[1]}: {Mega.values[1]}'],
        autopct='%1.1f%%'
        )
plt.title('Count of retained vs. churned');
```



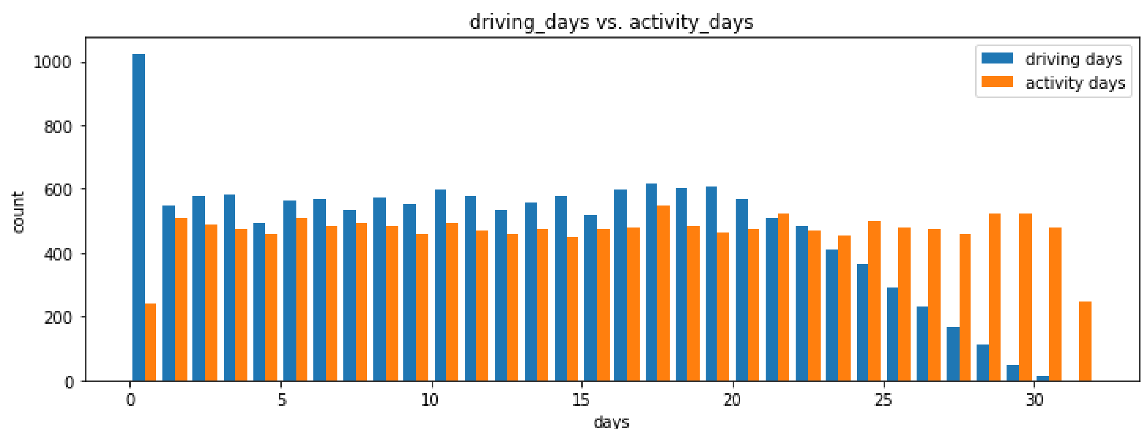
Less than 18% of the users churned.

`driving_days` vs. `activity_days`

Because both `driving_days` and `activity_days` represent counts of days over a month and they're also closely related, we can plot them together on a single histogram. This will help to better understand how they relate to each other without having to scroll back and forth comparing histograms in two different places.

Let's Plot a histogram that, for each day, has a bar representing the counts of `driving_days` and `user_days`.

In [24]:
```
# Histogram
plt.figure(figsize=(12,4))
label=['driving days', 'activity days']
plt.hist([data['driving_days'], data['activity_days']],
         bins=range(0,33),
         label=label)
plt.xlabel('days')
plt.ylabel('count')
plt.legend()
plt.title('driving_days vs. activity_days');
```



As observed previously, this might seem counterintuitive. After all, why are there *fewer* people who didn't use the app at all during the month and *more* people who didn't drive at all during the month?

On the other hand, it could just be illustrative of the fact that, while these variables are related to each other, they're not the same. People probably just open the app more than they use the app to drive—perhaps to check drive times or route information, to update settings, or even just by mistake.

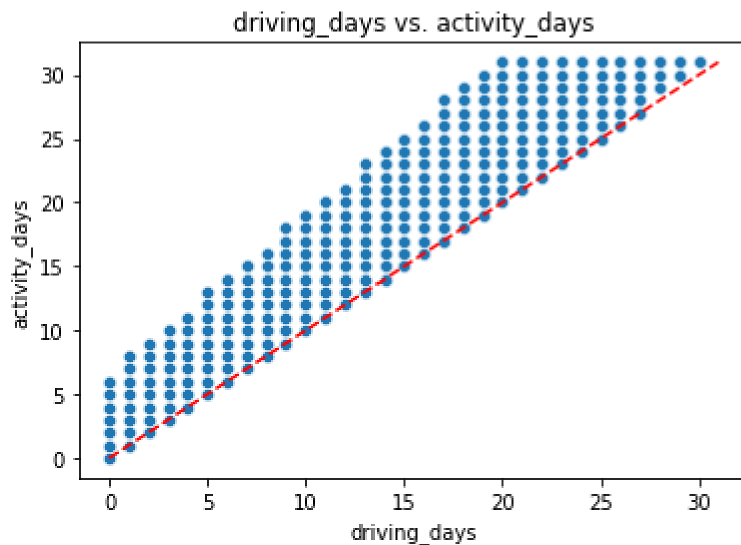
In [25]:
```
##Let's confirm the maximum number of days for each variable;`driving_d
print(data['driving_days'].max())
print(data['activity_days'].max())
```

```
30
31
```

It's true. Although it's possible that not a single user drove all 31 days of the month, it's highly unlikely, considering there are 15,000 people represented in the dataset.

Let's use another way to check the validity of these variables by plotting a simple scatter plot with the x-axis representing one variable and the y-axis representing the other.

In [26]:
```
# Scatter plot
sns.scatterplot(data=data, x='driving_days', y='activity_days')
plt.title('driving_days vs. activity_days')
plt.plot([0,31], [0,31], color='red', linestyle='--');
```

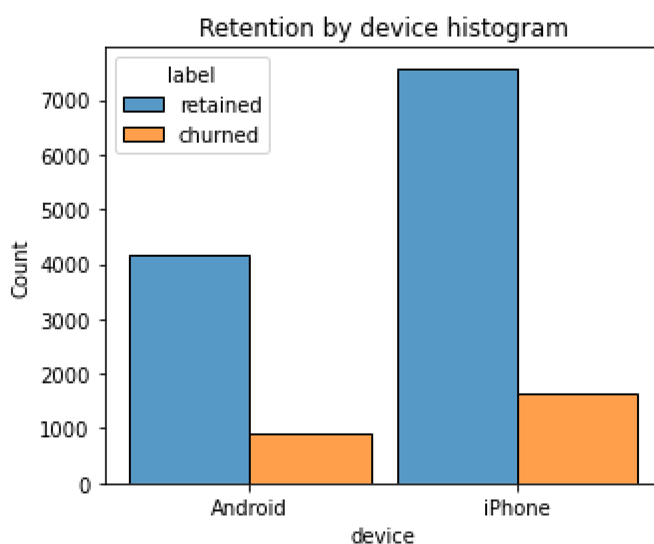


Notice that there is a theoretical limit. If you use the app to drive, then by definition it must count as a day-use as well. In other words, you cannot have more drive-days than activity-days. None of the samples in this data violate this rule, which is good.

Retention by device

Let's plota histogram to understand the churn rate by device type.

In [27]:
```
# Histogram
plt.figure(figsize=(5,4))
sns.histplot(data=data,
             x='device',
             hue='label',
             multiple='dodge',
             shrink=0.9
             )
plt.title('Retention by device histogram');
```



The proportion of churned users to retained users is consistent between device types.

Retention by kilometers driven per driving day

Let's examine retnetion by KM driving per day.

In [28]:
```
## Let's create `km_per_driving_day` column
data['km_per_driving_day'] = data['driven_km_drives'] / data['driving_d

# Let's pull the statistic description of the new column
data['km_per_driving_day'].describe()
```

Out[28]:
```
count    1.499900e+04
mean              inf
std               NaN
min      3.022063e+00
25%      1.672804e+02
50%      3.231459e+02
75%      7.579257e+02
max               inf
Name: km_per_driving_day, dtype: float64
```

Here the mean value is infinity, the standard deviation is NaN, and the max value is infinity.

This is the result of there being values of zero in the `driving_days` column.

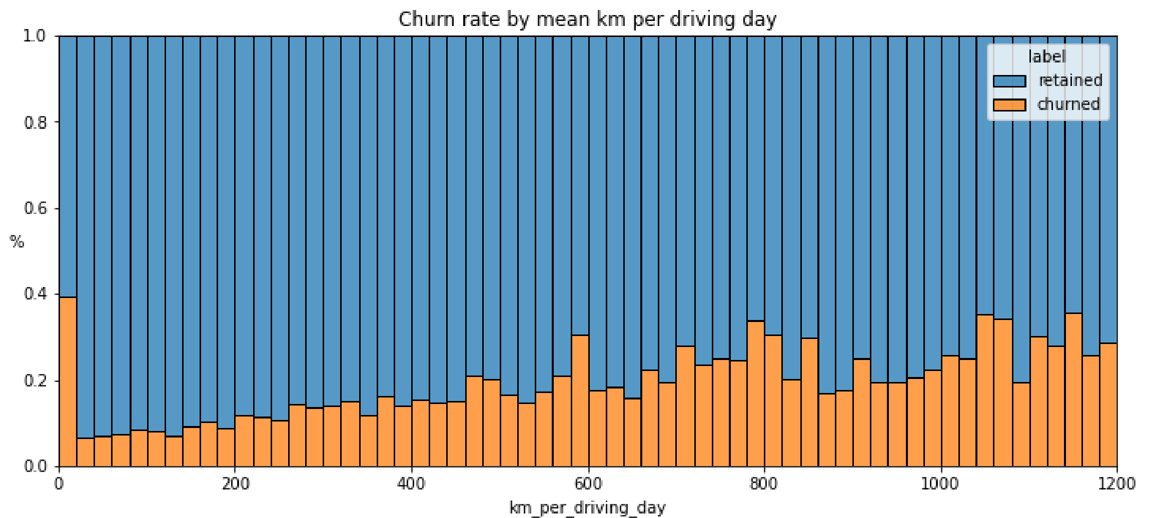
Let's convert these values from infinity to zero and recheck it

In [29]:
```
# 1. Convert infinite values to zero
data.loc[data['km_per_driving_day']==np.inf, 'km_per_driving_day'] = 0

# 2. Confirm that it worked
data['km_per_driving_day'].describe()
```

Out[29]:
```
count    14999.000000
mean       578.963113
std       1030.094384
min          0.000000
25%        136.238895
50%        272.889272
75%        558.686918
max      15420.234110
Name: km_per_driving_day, dtype: float64
```

The maximum value is 15,420 kilometers *per drive day*. This is physically impossible. Driving 100 km/hour for 12 hours is 1,200 km. It's unlikely many people averaged more than this each day they drove, so, for now, lets disregard rows where the distance in this column is greater than 1,200 km and plot a histogram to understand the distribution versus churned and retained users.

In [30]:
```
# Histogram
plt.figure(figsize=(12,5))
sns.histplot(data=data,
             x='km_per_driving_day',
             bins=range(0,1201,20),
             hue='label',
             multiple='fill')
plt.ylabel('%', rotation=0)
plt.title('Churn rate by mean km per driving day');
```

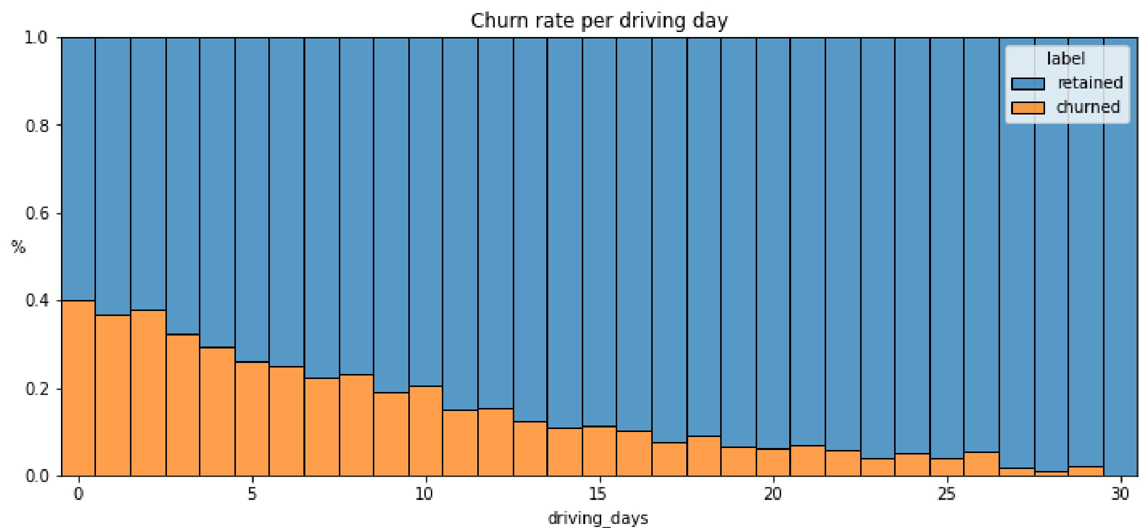


The churn rate tends to increase as the mean daily distance driven increases.

Churn rate per number of driving days

lets investegate the number of driving days vs the churn rate

In [31]:
```
# Histogram
plt.figure(figsize=(12,5))
sns.histplot(data=data,
             x='driving_days',
             bins=range(1,32),
             hue='label',
             multiple='fill',
             discrete=True)
plt.ylabel('%', rotation=0)
plt.title('Churn rate per driving day');
```



The churn rate is highest for people who didn't use Waze much during the last month. The more times they used the app, the less likely they were to churn. nobody who used the app 30 days churned.

This isn't surprising. If people who used the app a lot churned, it would likely indicate dissatisfaction. When people who don't use the app churn, it might be the result of dissatisfaction in the past, or it might be indicative of a lesser need for a navigational app. Maybe they moved to a city with good public transportation and don't need to drive anymore and etc.

Lets engineer a feature to understand the percent of each user's total sessions that were logged in their last month of use(percent_sessions_in_last_month).

In [32]:
```
data['percent_sessions_in_last_month'] = data['sessions'] / data['total
```

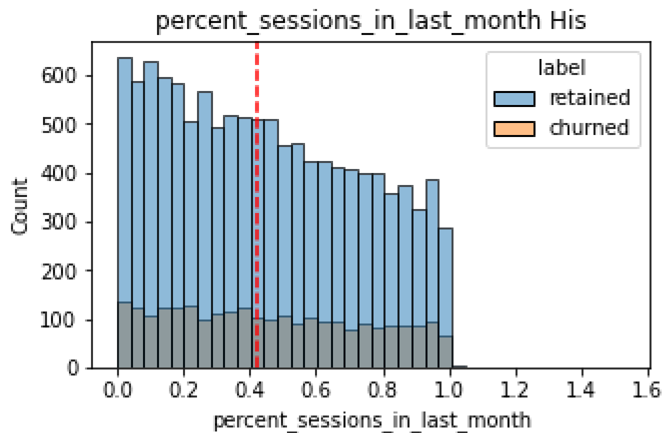
now lets find the median

In [33]:
```
data['percent_sessions_in_last_month'].median()
```

Out[33]: 0.42309702992763176

Type *Markdown* and LaTeX: α^2

In [34]:
```
1 ## Now, let's create a histogram depicting the distribution of values i
2
3 # Histogram
4 plt.figure(figsize=(5,3))
5 sns.histplot(x=data['percent_sessions_in_last_month'],hue=data['label']
6 median = data['percent_sessions_in_last_month'].median()
7 plt.axvline(median, color='red', linestyle='--')
8
9 plt.title('percent_sessions_in_last_month His');
```



Type *Markdown* and LaTeX: α^2

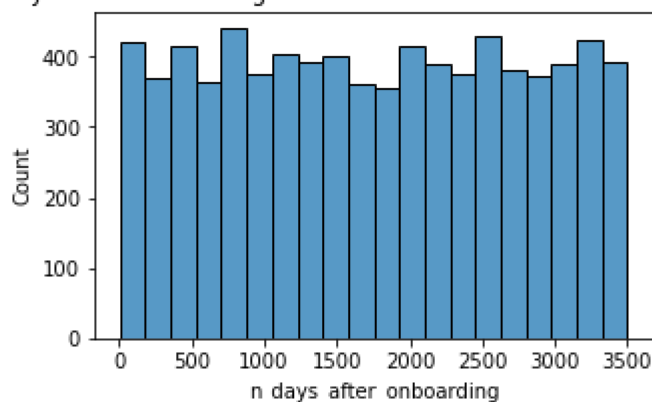
In [35]:
```
1 ##Let's check the median value of the `n_days_after_onboarding` variabl
2 data['n_days_after_onboarding'].median()
```

Out[35]: 1741.0

Half of the people in the dataset had 40% or more of their sessions in just the last month, yet the overall median time since onboarding is almost five years.

Lets make a histogram of `n_days_after_onboarding` for just the people who had 40% or more of their total sessions in the last month.

In [36]:
```
1 # Histogram
2 df = data.loc[data['percent_sessions_in_last_month']>=0.4]
3 plt.figure(figsize=(5,3))
4 sns.histplot(x=df['n_days_after_onboarding'])
5 plt.title('Num. days after onboarding for users with >=40% sessions in
```



The number of days since onboarding for users with 40% or more of their total sessions occurring in just the last month is a uniform distribution. This is very strange. It's worth asking Waze why so many long-time users suddenly used the app so much in the last month.

Observations:

- Analysis revealed that the overall churn rate is ~17%, and that this rate is consistent between iPhone users and Android users.
- EDA has revealed that users who drive very long distances on their driving days are *more* likely to churn, but users who drive more often are *less* likely to churn.
- There is missing data in the user churn label, so we might need further data processing before further analysis.
- There are many outlying observations for drives, so we might consider a variable transformation to stabilize the variation.
- The number of drives and the number of sessions are both strongly correlated, so they might provide redundant information when we incorporate both in a model.
- On average, retained users have fewer drives than churned users.
- several variables had highly improbable or perhaps even impossible outlying values, such as driven_km_drives.
- Users of all tenures from brand new to ~10 years were relatively evenly represented in the data. This is borne out by the histogram for `n_days_after_onboarding` , which reveals a uniform distribution for this variable.*

Descriptive statistics and hypothesis testing:

Let's focus on the device type, by examining the number of derives in both devices. basically, Do drivers who open the application using an iPhone have the same number of drives on average as drivers who use Android devices?

In [37]:
```
## Let's import the stats lib.
from scipy import stats
```

In [38]:
```
data_stat = data.copy()
data_stat['device_type'] = data['device']

## lets convert the devices type to numerical value.
map_dictionary = {'Android':2,'iPhone':1}

data_stat['device_type'] = data_stat['device_type'].map(map_dictionary)
data_stat['device_type'].head()
```

Out[38]:
```
0    2
1    1
2    2
3    1
4    2
Name: device_type, dtype: int64
```

In [39]:
```
##Let's look at the average number of drives for each device type
data_stat.groupby('device_type')['drives'].mean()
```

Out[39]:
```
device_type
1    67.859078
2    66.231838
Name: drives, dtype: float64
```

Based on the averages shown, it appears that drivers who use an iPhone device to interact with the application have a higher number of drives on average. However, this difference might arise from random sampling, rather than being a true difference in the number of drives. To assess whether the difference is statistically significant, Let's conduct a hypothesis test.

Let's conduct a t-test for two independent samples. the test should be appropriate since the groups are independent. Lets state our hypothesis:

H_0: There is no difference in average number of drives between drivers who use iPhone devices and drivers who use Androids.

H_A: There is a difference in average number of drives between drivers who use iPhone devices and drivers who use Androids.

Our significance level is 5%.

In [40]:
```
## Let's isolate the device types from the column.
iPhone = data_stat[data_stat['device_type'] == 1]['drives']

# 2. Isolate the `drives` column for Android users.
Android = data_stat[data_stat['device_type'] == 2]['drives']

# 3. Perform the t-test
stats.ttest_ind(a=iPhone, b=Android, equal_var=False)
```

Out[40]:
```
Ttest_indResult(statistic=1.4635232068852353, pvalue=0.1433519726802059)
```

Since the p-value is larger than the chosen significance level (5%), we fail to reject the null hypothesis. and we conclude that there is **not** a statistically significant difference in the average number of drives between drivers who use iPhones and drivers who use Androids.

Modeling Approaches:

Approach A: Binomial Logisitic regression:

Let's build binomial logistic regression model which helps in estimating the probability of an outcome, and evaulating its performance.

In [43]:
```
##Preparing the environment and Load the packages.
# Packages for Logistic Regression & Confusion Matrix
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score, prec
recall_score, f1_score, confusion_matrix, ConfusionMatrixDisplay
from sklearn.linear_model import LogisticRegression
```

In [45]:
```
## Let's check the balance of the outcome variable:
df['label'].value_counts(normalize=True)
```

Out[45]:
```
retained    0.819112
churned     0.180888
Name: label, dtype: float64
```

The balance of the outcome variable to the independent variable is decent, it hleps to make sure we use stratify to represent the outcome variable minority in both dataset (Train & Test).

In [46]:
```
##Lets call the describe function to quickly examine if there is an out
data.describe()
```

Out[46]:

	sessions	drives	total_sessions	n_days_after_onboarding	total_navigations
count	14999.000000	14999.000000	14999.000000	14999.000000	14999.0(
mean	80.633776	67.281152	189.964447	1749.837789	121.6(
std	80.699065	65.913872	136.405128	1008.513876	148.1:
min	0.000000	0.000000	0.220211	4.000000	0.0(
25%	23.000000	20.000000	90.661156	878.000000	9.0(
50%	56.000000	48.000000	159.568115	1741.000000	71.0(
75%	112.000000	93.000000	254.192341	2623.500000	178.0(
max	743.000000	596.000000	1216.154633	3500.000000	1236.0(

The following column they all seem to have outliers:

- `sessions`
- `drives`
- `total_sessions`
- `total_navigations_fav1`
- `total_navigations_fav2`
- `driven_km_drives`
- `duration_minutes_drives` All of these columns have max values that are multiple standard deviations above the 75th percentile. This could indicate outliers in these variables.

For this analysis, impute the outlying values for these columns. Lets calculate the **95th percentile** of each column and change to this value any value in the column that exceeds it.

In [48]:
```
## Lets create a dataset for the logistic model.
data_lg= data.copy()
```

In [49]:
```
# Impute outliers
for column in ['sessions', 'drives', 'total_sessions', 'total_navigatio
               'total_navigations_fav2', 'driven_km_drives', 'duration_
    threshold = data_lg[column].quantile(0.95)
    data_lg.loc[data_lg[column] > threshold, column] = threshold
```

In [51]:
```
##Lets call the describe function to check the previous step.
data_lg.describe()
```

Out[51]:

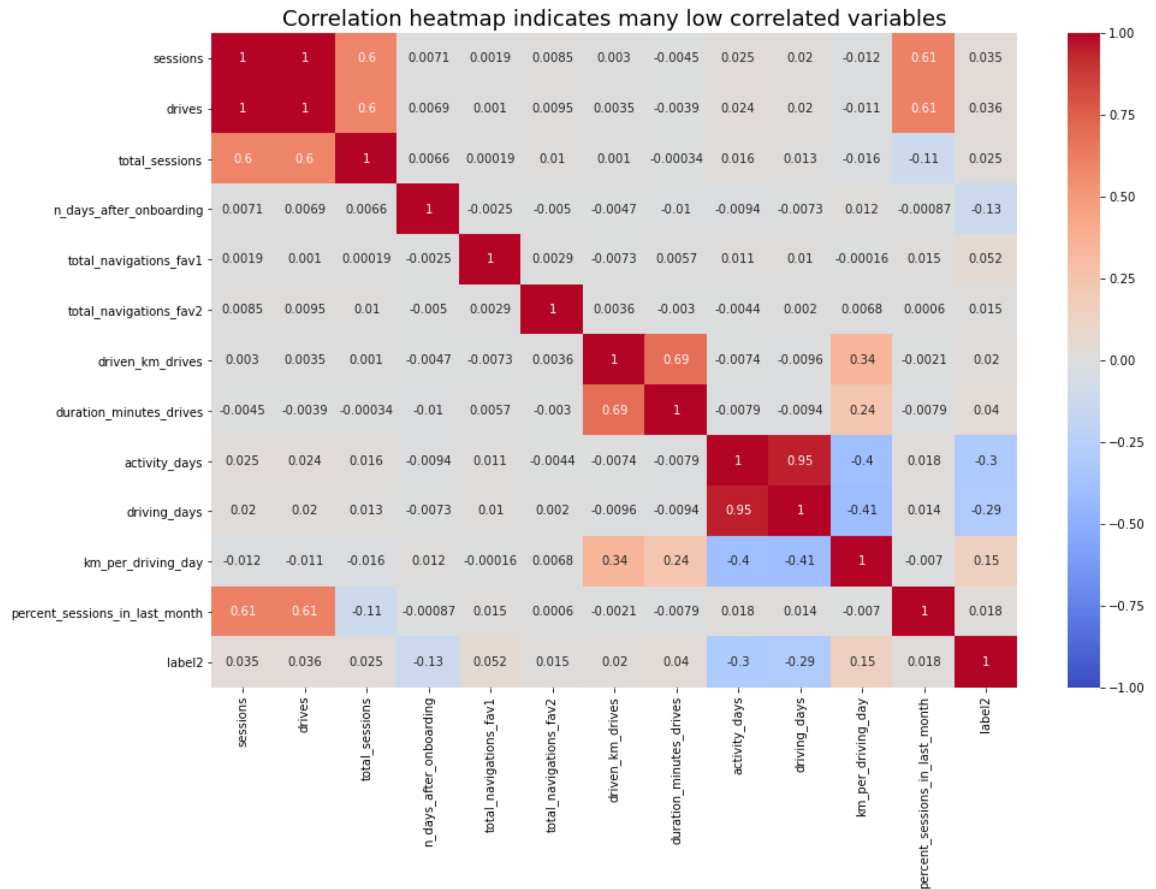
	sessions	drives	total_sessions	n_days_after_onboarding	total_navigations_
count	14999.000000	14999.000000	14999.000000	14999.000000	14999.0(
mean	76.568705	64.058204	184.031320	1749.837789	114.4
std	67.297958	55.306924	118.600463	1008.513876	124.6{
min	0.000000	0.000000	0.220211	4.000000	0.0(
25%	23.000000	20.000000	90.661156	878.000000	9.0(
50%	56.000000	48.000000	159.568115	1741.000000	71.0(
75%	112.000000	93.000000	254.192341	2623.500000	178.0(
max	243.000000	201.000000	454.363204	3500.000000	424.0(

In [52]:
```
##Previously we have found that 700 of label column is na, lets drop th
data_lg = data_lg.dropna(subset=['label'])
```

In [56]:
```
## Let's change the data type on label & device to binary.
data_lg['label2']= np.where(data_lg['label']=='churned',1,0)
data_lg['device2'] = np.where(data_lg['device']=='Android', 0, 1)
```

Binomial logistic regression has assumptions to work, independent observations, no extreme outliers which are met, now lets confirm the multicollineraity that it should not exist in the X predictors.

In [55]:
```
## Lets check collinearity between the variables by plotting a heatmap.
plt.figure(figsize=(15,10))
sns.heatmap(data_lg.corr(method='pearson'), vmin=-1, vmax=1, annot=True
plt.title('Correlation heatmap indicates many low correlated variables'
          fontsize=18)
plt.show();
```



If there are predictor variables that have a Pearson correlation coefficient value greater than the **absolute value of 0.7**, these variables are strongly multicollinear. Therefore, only one of these variables should be used in your model.

The following variables are multicolinear with each other:

> - *`sessions` and `drives` : 1.0*
> - *`driving_days` and `activity_days` : 0.95*

In [57]:
```
## Let's build the model, assign the predictors to X and the target var
X = data_lg.drop(columns = ['label', 'label2', 'device', 'sessions', 'd
```

Notice that `sessions` and `driving_days` were selected to be dropped, rather than `drives` and `activity_days`. The reason for this is that the features that were kept for modeling had slightly stronger correlations with the target variable than the features that were dropped.

In [58]:
```
# Isolating target variable
y = data_lg['label2']
```

In [59]:
```
## Let's split our data to train and test dataset.
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, r
```

In [60]:
```
##Lets fit the model and use penalty to 'none' since our predictors are
model_lg = LogisticRegression(penalty='none', max_iter=400)

model_lg.fit(X_train, y_train)

```

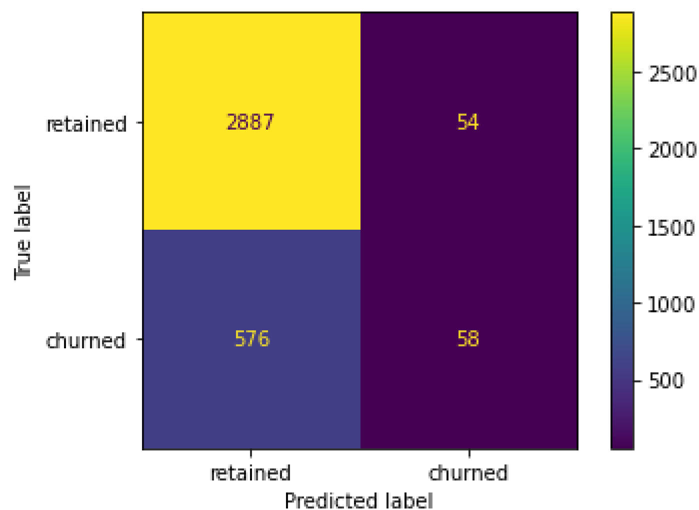
Out[60]:
```
LogisticRegression(max_iter=400, penalty='none')
```

In [62]:
```
# let generate predictions on X_test
y_preds = model_lg.predict(X_test)
```

In [64]:
```
# Let's score the model (accuracy) on the test data
model_lg.score(X_test, y_test)
```

Out[64]:
```
0.8237762237762237
```

In [65]:
```
## lets show the result using a confusion matrix.
cm = confusion_matrix(y_test, y_preds)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=['retained', 'churned'],
                              )
disp.plot();
```



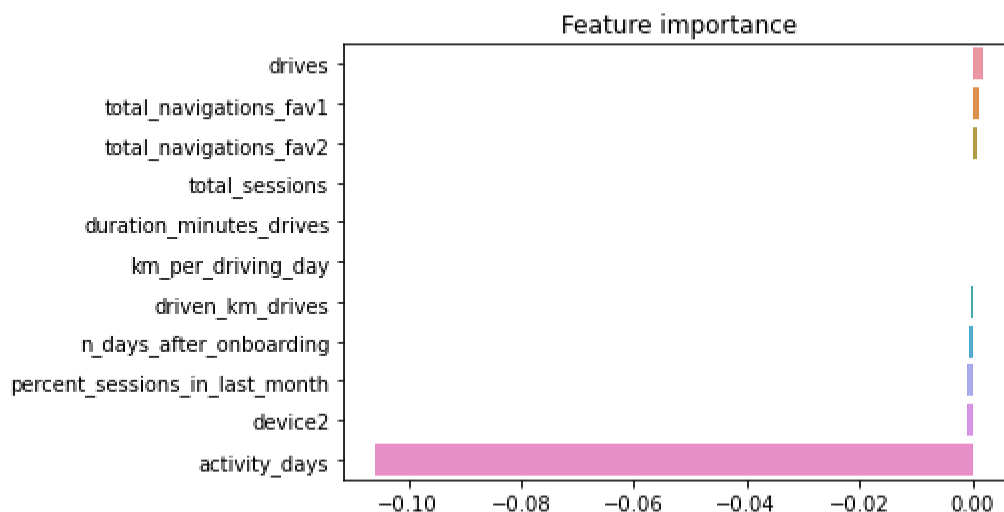
In [66]:
```
## Let's create a classification report.
target_labels = ['retained','churned']
print(classification_report(y_test,y_preds,target_names=target_labels))
```

```
              precision    recall  f1-score   support

    retained       0.83      0.98      0.90      2941
     churned       0.52      0.09      0.16       634

    accuracy                           0.82      3575
   macro avg       0.68      0.54      0.53      3575
weighted avg       0.78      0.82      0.77      3575
```

The model has mediocre precision and very low recall, which means that it makes a lot of false negative predictions and fails to capture users who will churn.

In [73]:
```
## Lets find the features importance of this model.
# Lets create a list of (column_name, coefficient) tuples
feature_importance = list(zip(X_train.columns, model_lg.coef_[0]))

# Sort the list by coefficient value
feature_importance = sorted(feature_importance, key=lambda x: x[1], rev
```

In [74]:
```
##Let's plot the feature importance.
sns.barplot(x=[x[1] for x in feature_importance],
            y=[x[0] for x in feature_importance],
            orient='h')
plt.title('Feature importance');
```



`activity_days` was by far the most important feature in the model. It had a negative correlation with user churn. This was not surprising, as this variable was very strongly correlated with `driving_days`, which was known from EDA to have a negative correlation with churn.

The model is not a strong enough predictor, as made clear by its poor recall score.

Approach B: Random Forest and XGBoost:

In [76]:
```
# Import packages for data modeling
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_auc_score, roc_curve, auc


from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier

# This is the function that helps plot feature importance
from xgboost import plot_importance
# This module lets us save our models once we fit them.
import pickle
```

So far we have engineered two features, lets create the following features

- `total sessions per day` , it represent the mean sessions per day since the onboarding.
- `KM/hr` , represent the mean km per hour driven in the last month.
- `KM/drive` ,mean number of kilometers per drive made in the last month for each user.
- `percent_of_sessions_to_favorite` ,represents the percentage of total sessions that were used to navigate to one of the users' favorite places.

In [78]:
```
# `total_sessions_per_day` feature
data['total_sessions_per_day'] = data['total_sessions'] / data['n_days_
```

In [84]:
```
# Lets review the total session per day by calling the describe functio
data['total_sessions_per_day'].describe()
```

Out[84]:
```
count    14999.000000
mean         0.338698
std          1.314333
min          0.000298
25%          0.051037
50%          0.100775
75%          0.216269
max         39.763874
Name: total_sessions_per_day, dtype: float64
```

In [81]:
```
## km_per_hour feature
data['km_per_hour'] = data['driven_km_drives'] / (data['duration_minute
data['km_per_hour'].describe()
```

Out[81]:
```
count    14999.000000
mean       190.394608
std        334.674026
min         72.013095
25%         90.706222
50%        122.382022
75%        193.130119
max      23642.920871
Name: km_per_hour, dtype: float64
```

there is huge descripency here, as the max value exceeds the month period.

In [82]:
```
1 ##`km_per_drive` feature
2 data['km_per_drive'] = data['driven_km_drives'] / data['drives']
```

In [83]:
```
1 ##Lets call the describe function to review the values.
2 data['km_per_drive'].describe()
```

Out[83]:
```
count    1.499900e+04
mean              inf
std               NaN
min      1.008775e+00
25%      3.323065e+01
50%      7.488006e+01
75%      1.854667e+02
max               inf
Name: km_per_drive, dtype: float64
```

In [85]:
```
1 # this feature has inf values, lets convert these values to zero.
2 # 1. Convert infinite values to zero
3 data.loc[data['km_per_drive']==np.inf, 'km_per_drive'] = 0
4 
5 # 2. Confirm that it worked
6 data['km_per_drive'].describe()
```

Out[85]:
```
count    14999.000000
mean       232.817946
std        620.622351
min          0.000000
25%         32.424301
50%         72.854343
75%        179.347527
max      15777.426560
Name: km_per_drive, dtype: float64
```

In [86]:
```
1 data.columns
```

Out[86]:
```
Index(['label', 'sessions', 'drives', 'total_sessions',
       'n_days_after_onboarding', 'total_navigations_fav1',
       'total_navigations_fav2', 'driven_km_drives', 'duration_minutes_dri
ves',
       'activity_days', 'driving_days', 'device', 'km_per_driving_day',
       'percent_sessions_in_last_month', 'total_sessions_per_day',
       'km_per_hour', 'km_per_drive'],
      dtype='object')
```

In [87]:
```
1 #`percent_of_sessions_to_favorite` feature
2 data['percent_of_drives_to_favorite'] = (
3     data['total_navigations_fav1'] + data['total_navigations_fav2']) /
```

In [89]:
```
# let descriptive stats
data['percent_of_drives_to_favorite'].describe()
```

Out[89]:
```
count    14999.000000
mean         1.665439
std          8.865666
min          0.000000
25%          0.203471
50%          0.649818
75%          1.638526
max        777.563629
Name: percent_of_drives_to_favorite, dtype: float64
```

In [92]:
```
## Dropping the missing values and create an allias to the dataset.
f_data = data.copy()
```

In [93]:
```
f_data = f_data.dropna(subset=['label'])
```

Outliers, there is no need to correct the outliers due to the tree based models are reilient to outliers

In [94]:
```
##Lets convert the categoral variables to binary.
f_data['device2'] = np.where(f_data['device']=='Android', 0, 1)
f_data['label2'] = np.where(f_data['label']=='churned', 1, 0)

```

Earlier we have discovered that 18% of the users in this dataset churned. This is an unbalanced dataset, but not extremely so. It can be modeled without any class rebalancing. Therefore, accuracy might not be the best gauge of performance because a model can have high accuracy on an imbalanced dataset and still fail to predict the minority class. And, It was already determined that the risks involved in making a false positive prediction are minimal. No one stands to get hurt, lose money, or suffer any other significant consequence if they are predicted to churn. Lets select the model based on the recall score.

In [95]:
```
f_data.shape
```

Out[95]:
```
(14299, 20)
```

Steps to take for the treebased models.

1. Split the data into train/validation/test sets (60/20/20).
2. Fit models and tune hyperparameters on the training set
3. Perform final model selection on the validation set
4. Assess the champion model's performance on the test set

In [97]:
```
1  # 1. Isolate X variables
2  X = f_data.drop(columns=['label', 'label2', 'device'])
3
4  # 2. Isolate y variable
5  y = f_data['label2']
6
7  # 3. Split into train and test sets
8  X_tr, X_test, y_tr, y_test = train_test_split(X, y, stratify=y,
9                                                test_size=0.2, random_sta
10
11 # 4. Split into train and validate sets
12 X_train, X_val, y_train, y_val = train_test_split(X_tr, y_tr, stratify=
13                                                   test_size=0.25, rando
```

Verify the number of samples in the partitioned data.

In [98]:
```
1  for x in [X_train, X_val, X_test]:
2      print(len(x))
```

```
8579
2860
2860
```

In [110]:
```
1  # 1. Instantiating the random forest classifier
2  rf = RandomForestClassifier(random_state=42)
3
4  # 2. Lets create a dictionary of hyperparameters to tune
5  cv_params = {'max_depth': [None],
6               'max_features': [1.0],
7               'max_samples': [0.5]
8               'min_samples_leaf': [2],
9               'min_samples_split': [2],
10              'n_estimators': [300,500]
11              }
12
13 # 3. Let's define a dictionary of scoring metrics to capture
14 scoring = {'accuracy', 'precision', 'recall', 'f1'}
15
16 # 4. Instantiating the GridSearchCV object
17 rf_cv = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='recal
```

```
  File "<ipython-input-110-c27aeecefc81>", line 8
    'min_samples_leaf': [2],
    ^
SyntaxError: invalid syntax
```

Now Let's fit the model to the training data.

In [108]:
```
%%time
rf_cv.fit(X_train, y_train)
```

```
Wall time: 45.1 s
```

Out[108]:
```
GridSearchCV(cv=4, estimator=RandomForestClassifier(random_state=42),
             param_grid={'max_depth': [None], 'max_features': [1.0],
                         'max_samples': [0.5], 'min_samples_leaf': [2],
                         'min_samples_split': [2], 'n_estimators': [300]},
             refit='recall', scoring={'f1', 'precision', 'recall', 'accura
cy'})
```

In [109]:
```
# Examine best score
rf_cv.best_score_
```

Out[109]:
```
0.10904993783671778
```

In [111]:
```
# Examine best hyperparameter combo
rf_cv.best_params_
```

Out[111]:
```
{'max_depth': None,
 'max_features': 1.0,
 'max_samples': 0.5,
 'min_samples_leaf': 2,
 'min_samples_split': 2,
 'n_estimators': 300}
```

In [112]:
```
## Let's create make_results() function to output all of the scores of
def make_results(model_name:str, model_object, metric:str):
    '''
    Arguments:
        model_name (string): what you want the model to be called in th
        model_object: a fit GridSearchCV object
        metric (string): precision, recall, f1, or accuracy

    Returns a pandas df with the F1, recall, precision, and accuracy sc
    for the model with the best mean 'metric' score across all validati
    '''

    # Create dictionary that maps input metric to actual metric name in
    metric_dict = {'precision': 'mean_test_precision',
                   'recall': 'mean_test_recall',
                   'f1': 'mean_test_f1',
                   'accuracy': 'mean_test_accuracy',
                   }

    # Get all the results from the CV and put them in a df
    cv_results = pd.DataFrame(model_object.cv_results_)

    # Isolate the row of the df with the max(metric) score
    best_estimator_results = cv_results.iloc[cv_results[metric_dict[met

    # Extract accuracy, precision, recall, and f1 score from that row
    f1 = best_estimator_results.mean_test_f1
    recall = best_estimator_results.mean_test_recall
    precision = best_estimator_results.mean_test_precision
    accuracy = best_estimator_results.mean_test_accuracy

    # Create table of results
    table = pd.DataFrame({'model': [model_name],
                          'precision': [precision],
                          'recall': [recall],
                          'F1': [f1],
                          'accuracy': [accuracy],
                          },
                         )

    return table
```

In [113]:
```
results = make_results('RF cv', rf_cv, 'recall')
results
```

Out[113]:

	model	precision	recall	F1	accuracy
0	RF cv	0.490608	0.10905	0.178026	0.82189

Asside from the accuracy, the scores aren't that good. However, the logistic regression model was ~0.09, which means that this model has 33% better recall and about the same accuracy, and it was trained on less data.

XGBoost:

Lets improve our scores using an XGBoost model.

In [114]:
```
# 1. Instantiate the XGBoost classifier
xgb = XGBClassifier(objective='binary:logistic', random_state=42)

# 2. Create a dictionary of hyperparameters to tune
cv_params = {'max_depth': [6, 12],
             'min_child_weight': [3, 5],
             'learning_rate': [0.01, 0.1],
             'n_estimators': [300]
             }

# 3. Define a dictionary of scoring metrics to capture
scoring = {'accuracy', 'precision', 'recall', 'f1'}

# 4. Instantiate the GridSearchCV object
xgb_cv = GridSearchCV(xgb, cv_params, scoring=scoring, cv=4, refit='rec
```

In [115]:
```
%%time
xgb_cv.fit(X_train, y_train)
```

```
Wall time: 20.8 s
```

Out[115]:
```
GridSearchCV(cv=4,
             estimator=XGBClassifier(base_score=None, booster=None,
                                     callbacks=None, colsample_bylevel=Non
e,
                                     colsample_bynode=None,
                                     colsample_bytree=None, device=None,
                                     early_stopping_rounds=None,
                                     enable_categorical=False, eval_metric
=None,
                                     feature_types=None, gamma=None,
                                     grow_policy=None, importance_type=Non
e,
                                     interaction_constraints=None,
                                     learning_rate=None,...
                                     max_delta_step=None, max_depth=None,
                                     max_leaves=None, min_child_weight=Non
e,
                                     missing=nan, monotone_constraints=Non
e,
                                     multi_strategy=None, n_estimators=Non
e,
                                     n_jobs=None, num_parallel_tree=None,
                                     random_state=42, ...),
             param_grid={'learning_rate': [0.01, 0.1], 'max_depth': [6, 1
2],
                         'min_child_weight': [3, 5], 'n_estimators': [30
0]},
             refit='recall', scoring={'f1', 'precision', 'recall', 'accura
cy'})
```

In [116]:
```
# Examine best score
xgb_cv.best_score_
```

Out[116]:
```
0.17411244647050697
```

In [117]:
```
# Examine best parameters
xgb_cv.best_params_
```

Out[117]:
```
{'learning_rate': 0.1,
 'max_depth': 6,
 'min_child_weight': 5,
 'n_estimators': 300}
```

In [118]:
```
# Call 'make_results()' on the GridSearch object
xgb_cv_results = make_results('XGB cv', xgb_cv, 'recall')
results = pd.concat([results, xgb_cv_results], axis=0)
results
```

Out[118]:

	model	precision	recall	F1	accuracy
0	RF cv	0.490608	0.109050	0.178026	0.821890
0	XGB cv	0.436090	0.174112	0.248679	0.813614

This model fit the data even better than the random forest model. The recall score is nearly double the recall score from the logistic regression model from the previous course, and it's almost 50% better than the random forest model's recall score, with a minor drop in the precision and accuracy score

Model selection:

Let's use the best random forest model and the best XGBoost model to predict on the validation data. Whichever performs better will be selected as the champion model.

In [120]:
```
# Random forest model to predict on validation data
rf_val_preds = rf_cv.best_estimator_.predict(X_val)
```

In [121]:

```
##Let's create the get_test_scores() function to generate a table of sc
def get_test_scores(model_name:str, preds, y_test_data):
    '''
    Generate a table of test scores.

    In:
        model_name (string): Your choice: how the model will be named i
        preds: numpy array of test predictions
        y_test_data: numpy array of y_test data

    Out:
        table: a pandas df of precision, recall, f1, and accuracy score
    '''
    accuracy = accuracy_score(y_test_data, preds)
    precision = precision_score(y_test_data, preds)
    recall = recall_score(y_test_data, preds)
    f1 = f1_score(y_test_data, preds)

    table = pd.DataFrame({'model': [model_name],
                          'precision': [precision],
                          'recall': [recall],
                          'F1': [f1],
                          'accuracy': [accuracy]
                          })

    return table
```

In [122]:

```
# Get validation scores for RF model
rf_val_scores = get_test_scores('RF val', rf_val_preds, y_val)

# Append to the results table
results = pd.concat([results, rf_val_scores], axis=0)
results
```

Out[122]:

	model	precision	recall	F1	accuracy
0	RF cv	0.490608	0.109050	0.178026	0.821890
0	XGB cv	0.436090	0.174112	0.248679	0.813614
0	RF val	0.477273	0.124260	0.197183	0.820629

Notice that the scores went up from the training scores across all metrics except precision, but only by very little. This means that the model did not overfit the training data.

In [123]:

```
# Use XGBoost model to predict on validation data
xgb_val_preds = xgb_cv.best_estimator_.predict(X_val)

# Get validation scores for XGBoost model
xgb_val_scores = get_test_scores('XGB val', xgb_val_preds, y_val)

# Append to the results table
results = pd.concat([results, xgb_val_scores], axis=0)
results
```

Out[123]:

	model	precision	recall	F1	accuracy
0	RF cv	0.490608	0.109050	0.178026	0.821890
0	XGB cv	0.436090	0.174112	0.248679	0.813614
0	RF val	0.477273	0.124260	0.197183	0.820629
0	XGB val	0.435233	0.165680	0.240000	0.813986

the XGBoost model's validation scores were lower, but only very slightly. It is still the clear champion.

In [124]:

```
# use XGBoost model to predict on test data
xgb_test_preds = xgb_cv.best_estimator_.predict(X_test)

# get test scores for XGBoost model
xgb_test_scores = get_test_scores('XGB test', xgb_test_preds, y_test)

# Append to the results table
results = pd.concat([results, xgb_test_scores], axis=0)
results
```

Out[124]:

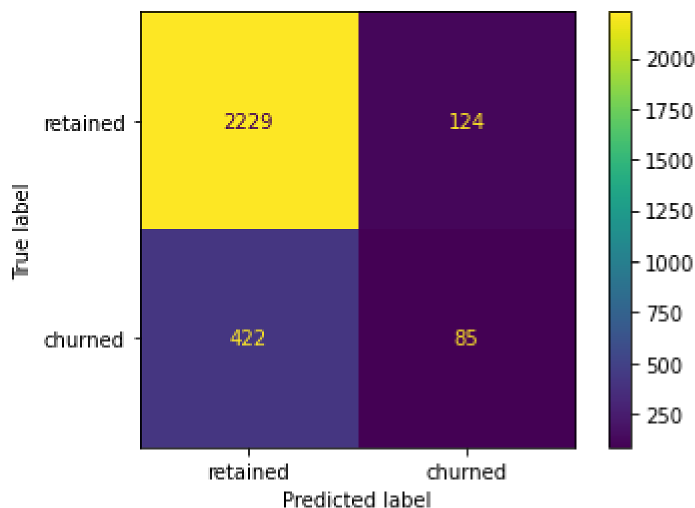
	model	precision	recall	F1	accuracy
0	RF cv	0.490608	0.109050	0.178026	0.821890
0	XGB cv	0.436090	0.174112	0.248679	0.813614
0	RF val	0.477273	0.124260	0.197183	0.820629
0	XGB val	0.435233	0.165680	0.240000	0.813986
0	XGB test	0.406699	0.167653	0.237430	0.809091

The recall was exactly the same as it was on the validation data, but the precision declined notably, which caused all of the other scores to drop slightly. Nonetheless, this is stil within the acceptable range for performance discrepancy between validation and test scores.

Confusion matrix

In [128]:
```
# generate array of values for confusion matrix
plt.figure(figsize=(20,8))
cm = confusion_matrix(y_test, xgb_test_preds, labels=xgb_cv.classes_)

# Plot confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=['retained', 'churned'])
disp.plot();
```

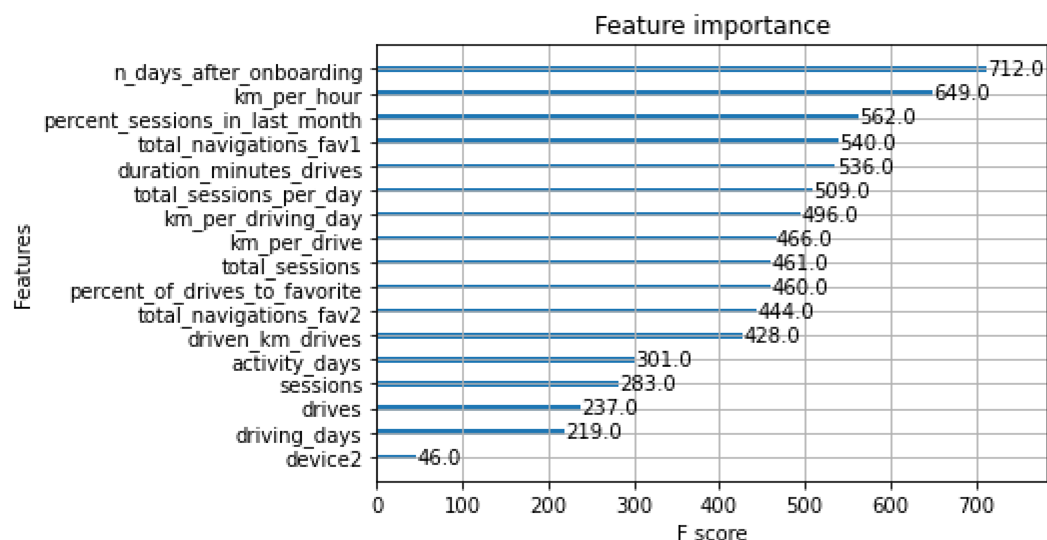
```
<Figure size 1440x576 with 0 Axes>
```



The model predicted three times as many false negatives than it did false positives, and it correctly identified only 16.5% of the users who actually churned.

Feature importance

In [129]:
```
plot_importance(xgb_cv.best_estimator_);
```



Conclusion:

- The model is not a strong enough predictor, as made clear by its poor recall score. However, if the model is only being used to guide further exploratory efforts, then it can have value.

- The default decision threshold for most implementations of classification algorithms—including scikit-learn's—is 0.5. This means that, in the case of the Waze models, if they predicted that a given user had a 50% probability or greater of churning, then that user was assigned a predicted value of 1—the user was predicted to churn. With imbalanced datasets where the response class is a minority, this threshold might not be ideal. a lower threshold will increase the model performance.

In [ ]:
```
1
```