CUSTOMER CHURN

by : Ana Farida

The goals are predicting whether or not a user will churn, and identifying the reasons that lead to churn. The model will allow Waze to prevent or at least reduce churn, hence improving retention and driving business growth.

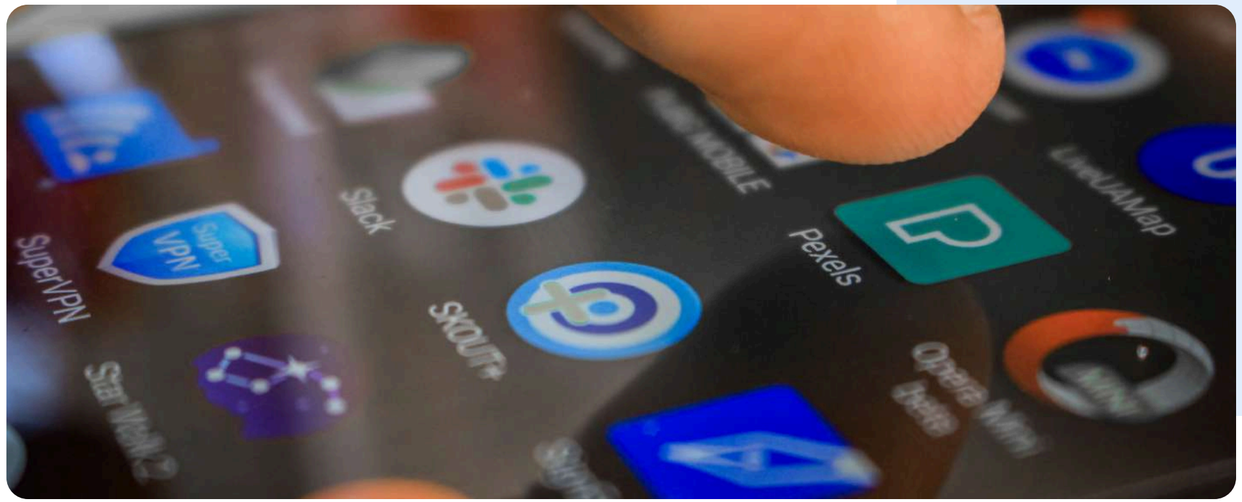
Impact of Model Errors:
False Negative: The model predicts a user won't churn, but they do. Waze may miss the chance to retain the user. Proactive actions, like app notifications or surveys, could help address dissatisfaction.

False Positive: The model predicts a user will churn, but they don't. Waze might take unnecessary actions to retain loyal users, potentially annoying them and harming their experience with the app.

Predicting Customer Churn

Key Predictors of Customer Churn

A/B Test
Comparing iPhone and Android as Users' Device Types

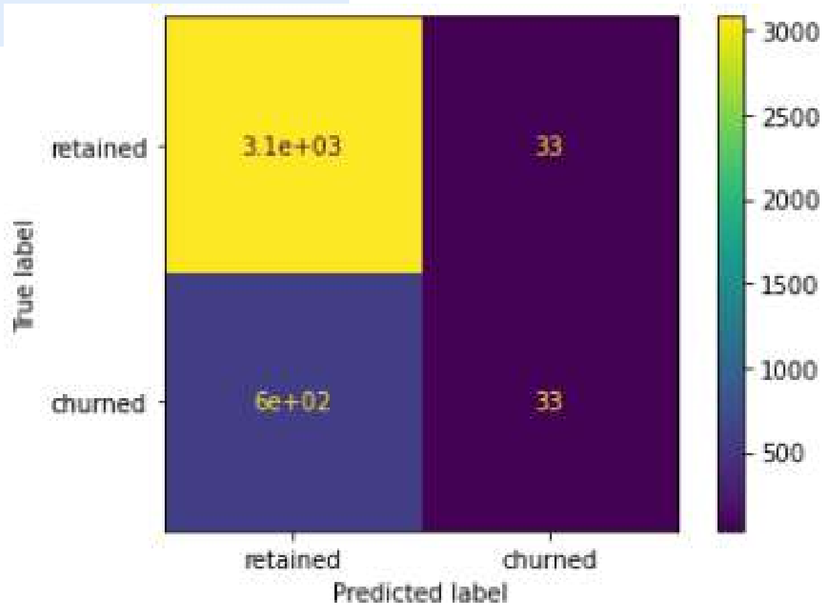


A. Logistic Regression

```
model = LogisticRegression(penalty='none', max_iter=400)
model.fit(X_train, y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=400,
                   multi_class='auto', n_jobs=None, penalty='none',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

	precision	recall	f1-score	support
retained	0.84	0.99	0.91	3116
churned	0.50	0.05	0.09	634
accuracy			0.83	3750
macro avg	0.67	0.52	0.50	3750
weighted avg	0.78	0.83	0.77	3750



B. Tree-Based (Random Forest & XGBoost)

```
rf = RandomForestClassifier(random_state=42)
rf_cv = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='recall')
```

```
xgb = XGBClassifier(objective= 'binary:logistic', random_state=42)
xgb_cv = GridSearchCV(xgb, cv_params, scoring=scoring, cv=4, refit='recall')
```

rf_cv.best_params_

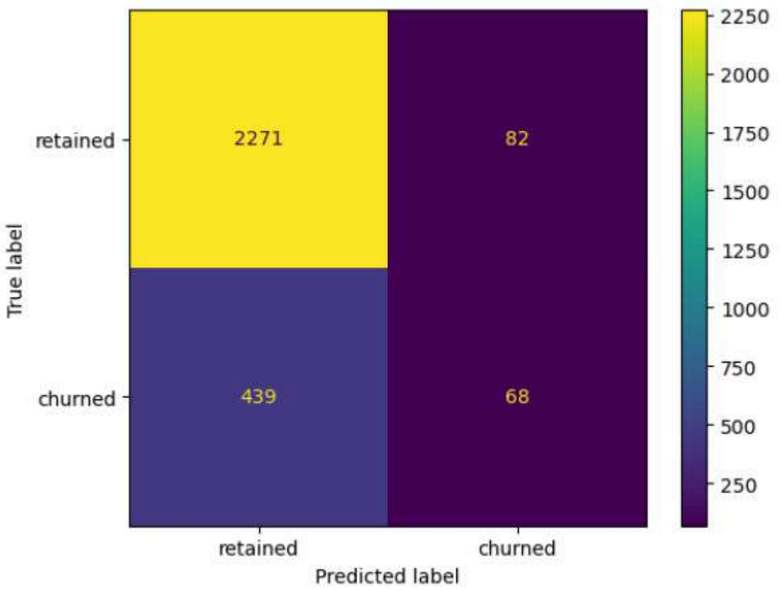
```
{'max_depth': None,
 'max_features': 1.0,
 'max_samples': 1.0,
 'min_samples_leaf': 2,
 'min_samples_split': 2,
 'n_estimators': 300}
```

xgb_cv.best_params_

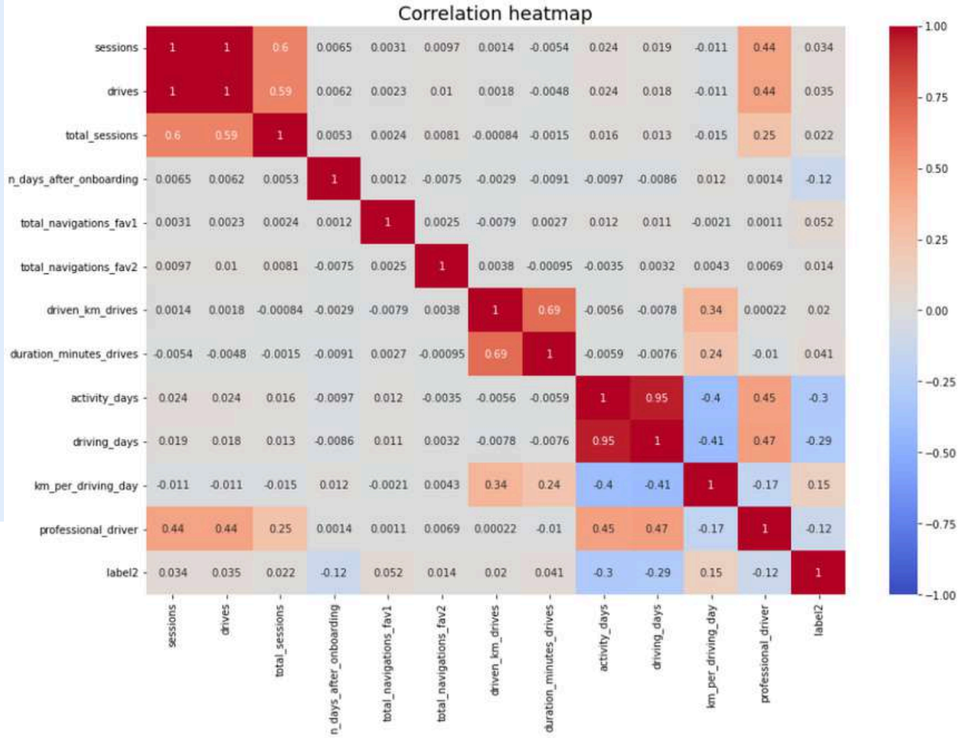
```
{'learning_rate': 0.01,
 'max_depth': 8,
 'min_child_weight': 3,
 'n_estimators': 30}
```

	model	precision	recall	F1	accuracy
0	RF cv	0.457163	0.126782	0.198445	0.818510
0	XGB cv	0.447306	0.139273	0.212365	0.816761
0	RF val	0.445255	0.120316	0.189441	0.817483
0	XGB val	0.461538	0.153846	0.230769	0.818182
0	XGB test	0.453333	0.134122	0.207002	0.817832

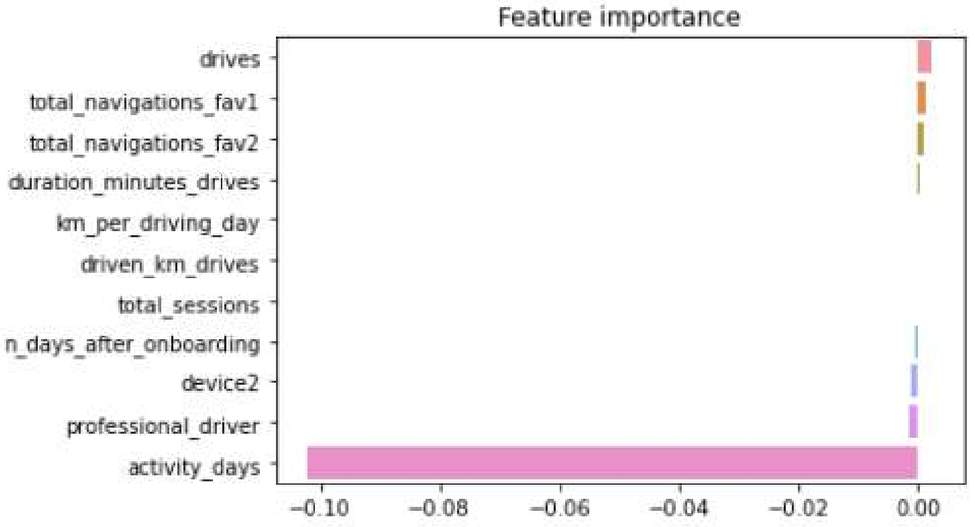
- The XGB model fits the data better than the random forest model. Its recall score (0.13) is higher than the logistic regression (0.05) and random forest model's (0.12), while keeping similar accuracy (0.18) and precision (0.45).
- The XGB model's recall and precision slightly declined, indicating a performance discrepancy between the validation and test scores.
- The XGB model predicted four times as many false negatives (439) than it did false positives (82).



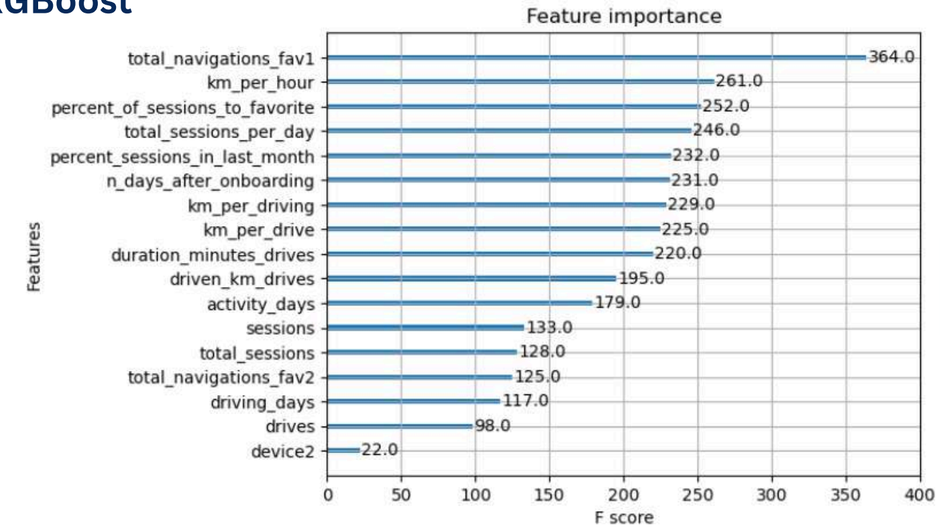
A. Pearson Correlation



B. Logistic Regression



C. XGBoost



- In the exploratory data analysis, km_per_driving_day had the strongest positive correlation **with user churn while activity_days and driving_days had the most negative**.
- Feature importance may vary across different models. Never abandon a feature without full insight into its relationship with the target variable. Different results can be caused by complex interactions between features.
- In the logistic regression model, activity_days was the most important feature, and it had a negative correlation with user churn. A new feature called professional_driver ranked third regarding predictive power.
- Whereas XGBoost incorporated more features in its use, logistic regression did rely highly on the activity_days feature in most of the predictions.

Comparing iPhone and Android as Users' Device Types

- **Do iPhone and Android Users Take the Same Number of Rides on Average?**
- **Purpose:** The goal is to determine if iPhone users take the same average number of rides as Android users.

```
map_dictionary = {'Android': 2, 'iPhone':1}
df["device_type"] = df["device"]
df["device_type"] = df["device_type"].map(map_dictionary)
```

```
df.groupby("device_type")["drives"].mean()
```

```
device_type
1    67.859078
2    66.231838
Name: drives, dtype: float64
```

```
iPhone = df[df["device_type"] == 1]["drives"]
Android = df[df["device_type"] == 2]["drives"]
stats.ttest_ind(a=iPhone, b=Android, equal_var=False)
```

```
Ttest_indResult(statistic=1.4635232068852353, pvalue=0.1433519726802059)
```

1. Descriptive Statistics:

- Calculate the average (mean) number of rides for each device type to identify initial differences.
- 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 could be due to random chance. To confirm, perform a two-sample t-test to check if the difference is statistically significant.

2. Hypothesis Test:

- Null Hypothesis (H_0): There is no difference in average number of drives between drivers who use iPhone devices and drivers who use Androids
- Alternative Hypothesis (H_1): There is a difference in average number of drives between drivers who use iPhone devices and drivers who use Androids.

3**. Conduct the t-test:**

- Since the p-value is (0.14) larger than the chosen significance level (5% or 0.05), it fail to reject the null hypothesis. It concludes that there is not a statistically significant difference in the average number of drives between drivers who use iPhones and drivers who use Androids.

- The key business insight is **that drivers who use iPhone devices on average have a similar number of drives as those who use Androids.**

Column name	Type	Description
ID	int	A sequential numbered index
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

```
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)
```

- There are 14,999 rows and 13 columns in the dataset. Each row represents a user.
- The dataset has 700 missing values in the `label` column. Other variables have no missing values.
- The variables `label` and `device` are of type `object`; `total_sessions`, `driven_km_drives`, and `duration_minutes_drives` are of type `float64`; the rest of the variables are of type `int64`.

```
null_df = df[df["label"].isnull()]
null_df.describe()
```

	ID	sessions	drives	total_sessions	n_days_after_onboarding	total_navigations_fav1	total_navigations_fav2	driven_km_drives	duration_minutes_drives	activity_days	driving_days
count	700.000000	700.000000	700.000000	700.000000	700.000000	700.000000	700.000000	700.000000	700.000000	700.000000	700.000000
mean	7405.584286	80.837143	67.798571	198.483348	1709.295714	118.717143	30.371429	3935.967029	1795.123358	15.382857	12.125714
std	4306.900234	79.987440	65.271926	140.561715	1005.306562	156.308140	46.306984	2443.107121	1419.242246	8.772714	7.626373
min	77.000000	0.000000	0.000000	5.582648	16.000000	0.000000	0.000000	290.119811	66.588493	0.000000	0.000000
25%	3744.500000	23.000000	20.000000	94.056340	869.000000	4.000000	0.000000	2119.344818	779.009271	8.000000	6.000000
50%	7443.000000	56.000000	47.500000	177.255925	1650.500000	62.500000	10.000000	3421.156721	1414.966279	15.000000	12.000000
75%	11007.000000	112.250000	94.000000	266.058022	2508.750000	169.250000	43.000000	5166.097373	2443.955404	23.000000	18.000000
max	14993.000000	556.000000	445.000000	1076.879741	3498.000000	1096.000000	352.000000	15135.391280	9746.253023	31.000000	30.000000

```
not_null_df = df[~df["label"].isnull()]
not_null_df.describe()
```

	ID	sessions	drives	total_sessions	n_days_after_onboarding	total_navigations_fav1	total_navigations_fav2	driven_km_drives	duration_minutes_drives	activity_days	driving_days
count	14299.000000	14299.000000	14299.000000	14299.000000	14299.000000	14299.000000	14299.000000	14299.000000	14299.000000	14299.000000	14299.000000
mean	7503.573117	80.623820	67.255822	189.547409	1751.822505	121.747395	29.638296	4044.401535	1864.199794	15.544653	12.182530
std	4331.207621	80.736502	65.947295	136.189764	1008.663834	147.713428	45.350890	2504.977970	1448.005047	9.016088	7.833835
min	0.000000	0.000000	0.000000	0.220211	4.000000	0.000000	0.000000	60.441250	18.282082	0.000000	0.000000
25%	3749.500000	23.000000	20.000000	90.457733	878.500000	10.000000	0.000000	2217.319909	840.181344	8.000000	5.000000
50%	7504.000000	56.000000	48.000000	158.718571	1749.000000	71.000000	9.000000	3496.545617	1479.394387	16.000000	12.000000
75%	11257.500000	111.000000	93.000000	253.540450	2627.500000	178.000000	43.000000	5299.972162	2466.928876	23.000000	19.000000
max	14998.000000	743.000000	596.000000	1216.154633	3500.000000	1236.000000	415.000000	21183.401890	15851.727160	31.000000	30.000000

- There is nothing particularly remarkable in the comparison of summary statistics between the observations with missing retention labels (retained vs churned) to those without any missing values. The means and standard deviations are reasonably consistent between the two sets.

```
print(df["label"].value_counts())
print()
print(df["label"].value_counts(normalize=True))
```

```
retained    11763
churned      2536
Name: label, dtype: int64

retained    0.822645
churned     0.177355
Name: label, dtype: float64
```

	driven_km_drives
count	14299.000000
mean	4044.401535
std	2504.977970
min	60.441250
25%	2217.319909
50%	3496.545617
75%	5299.972162
max	21183.401890

label	drives	driven_km_drives	duration_minutes_drives	activity_days	driving_days
churned	50.0	3652.655666	1607.183785	8.0	6.0
retained	47.0	3464.684614	1458.046141	17.0	14.0

- **This dataset contains 82% retained users and 18% churned users**.
- When comparing churned users to retained users, the median is calculated for each variable to avoid outliers distorting the representation of a typical user. For example, the maximum value in the column driven_km_drives is 21,183 km., which is more than half the circumference of the earth.

- Churned users took about three more drives in the last month than retained users (50 vs 47). However, retained users used the app on more than twice as many days during that time (17 vs 8).
- On average, churned users drove about 200 km more (3,653 vs 3,465) and spent 2.5 more hours driving than retained users last month (14 vs 6). It indicates that **churned users took more drives in fewer days with longer trips;** this pattern could indicate a certain user profile.

- Begin by dividing the `driven_km_drives` column by the `drives` column. Then, group the results by churned/retained and calculate the median km/drive of each group.

label	km_per_driving_day
churned	697.541999
retained	289.549333

label	drives_per_driving_day
churned	10.0000
retained	4.0625

label	km_per_drive
churned	74.109416
retained	75.014702

- These figures indicate that the customers in this dataset, who churned or retained, are serious drivers. This group likely does not represent the typical driver in general. In fact, **many of the users who churned might be long-haul truckers.**
- Considering how much these users drive, it could be worthwhile to gather more data about these "super-drivers." The reasons their heavy driving is occurring may explain why the app doesn't quite serve them, with very different needs than a normal commuter.

```
null_df["device"].value_counts(normalize=True)
```

```
iPhone     0.638571
Android    0.361429
Name: device, dtype: float64
```

```
df["device"].value_counts(normalize=True)
```

```
iPhone     0.644843
Android    0.355157
Name: device, dtype: float64
```

```
iPhone     447
Android    253
Name: device, dtype: int64
```

- **The percentage of missing values by each device is consistent** with their representation in the data overall.
- Of the 700 rows with null values, that is, retained vs. churned label, 447 correspond to iPhone users, and 253 correspond to Android users.

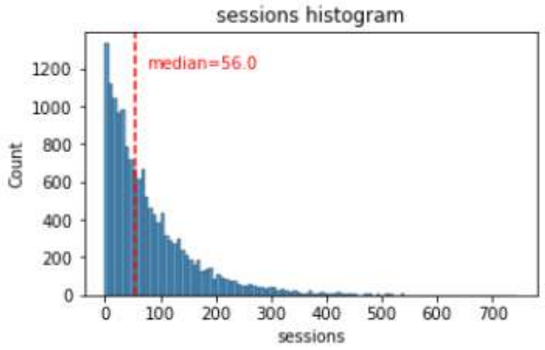
```
df.groupby(["label", "device"]).size()
```

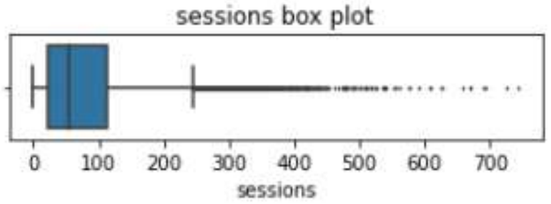
```
label     device
churned   Android     891
          iPhone     1645
retained  Android    4183
          iPhone     7580
dtype: int64
```

```
label     device
churned   iPhone     0.648659
          Android    0.351341
retained  iPhone     0.644393
          Android    0.355607
Name: device, dtype: float64
```

- **The ratio of iPhone users and Android users is consistent between the churned group and the retained group**, and those ratios are both consistent with the ratio found in the overall dataset.
- **Android users comprised approximately 36% of the sample, while iPhone users made up about 64%.** The churn rate for both iPhone and Android users was within one percentage point of each other. There is nothing suggestive of churn being correlated with device.

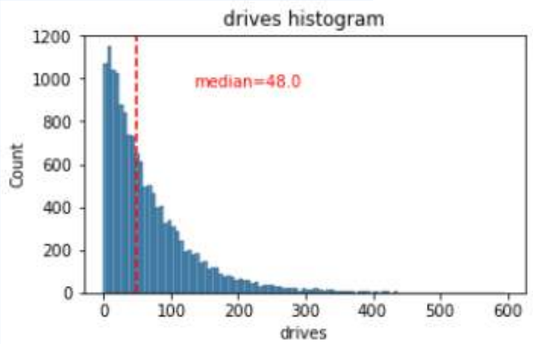
- The number of occurrence of a user opening the app during the month

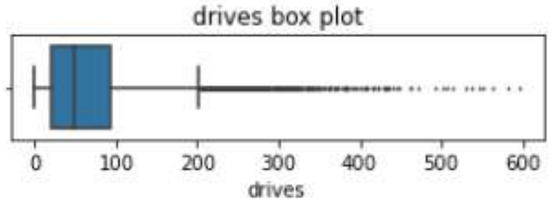




- The `**sessions**` variable is right-skewed, meaning most users have a relatively low number of sessions. **Half of the users had 56** or fewer sessions, but a few had over 700, as shown in the boxplot.

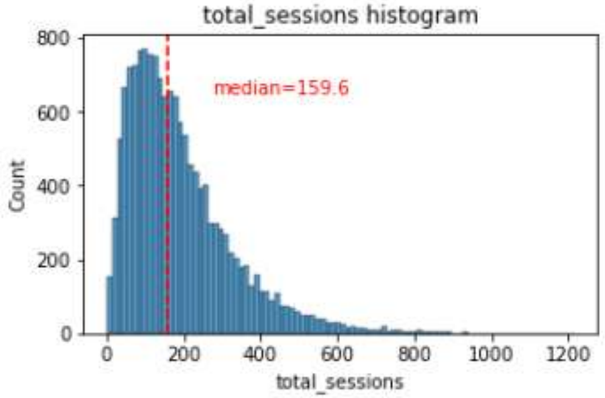
- An occurrence of driving at least 1 km during the month

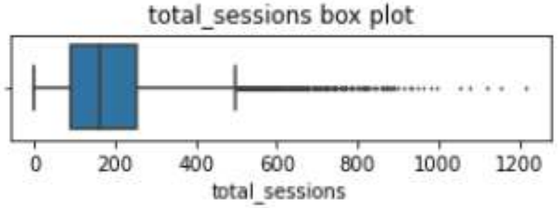




- The **`drives`** data is similar to the sessions variable, with a right-skewed distribution that looks roughly log-normal. **Most users had a median of 48 drives**, but some had over 400 drives in the last month.

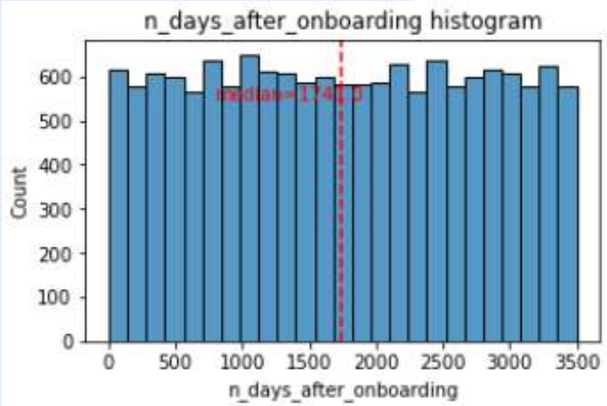
- The total number of sessions since a user has onboarded

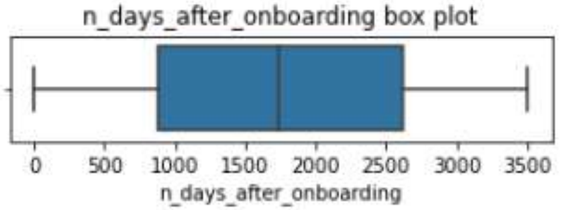




- The **`total_sessions`** is right-skewed. **The median is around 160 sessions**, whereas in the last month, it was 48, hence suggesting that the majority of a user's total sessions happened within the past month.

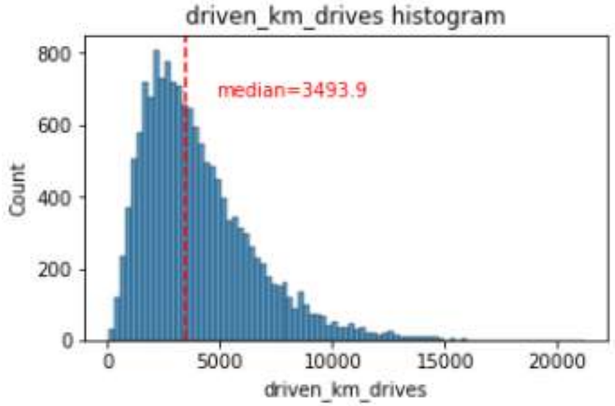
- The number of days since a user signed up for the app

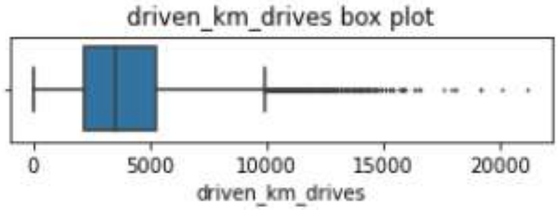




- User **tenure**, or days since onboarded are **evenly distributed** in a range from almost **0 days up to about 3,500 days** (around 9.5 years).

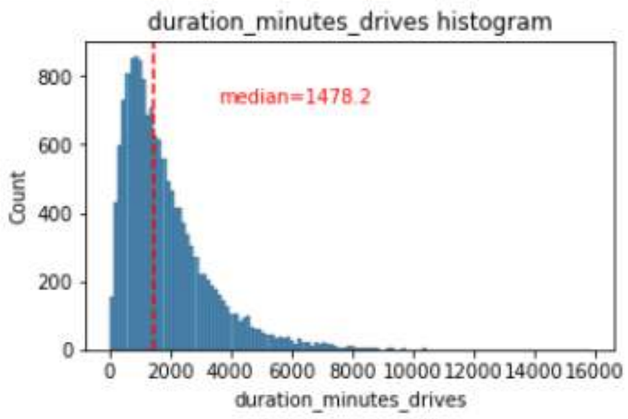
- Total kilometers driven during the month

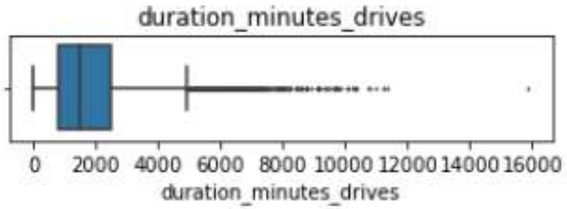




- The distribution of km driven by user in the last month is right-skewed, **half of the users drove less than 3,495 km**. These users drive a lot! The longest distance driven in the month was over half the circumferene of the earth.

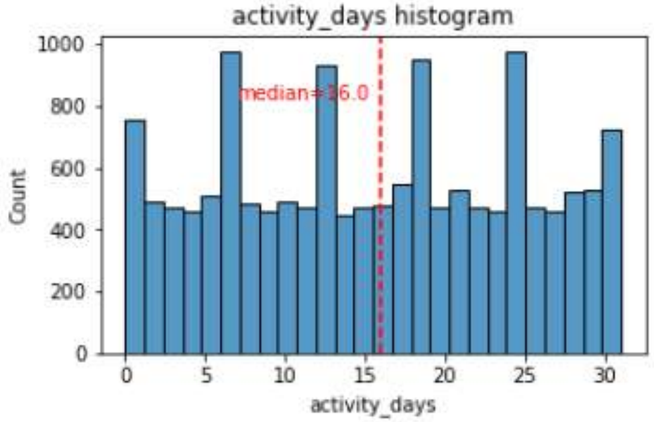
- Total duration driven in minutes during the month





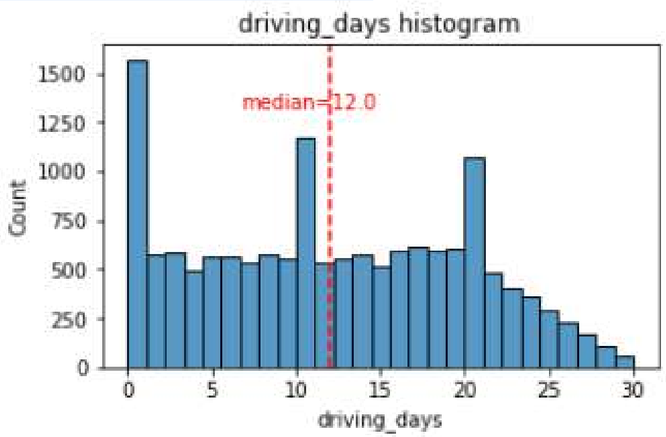
- The `duration_minutes_drives` variable has a heavily skewed right tail. **Half of the users drove less than 1,478 minutes (~25 hours)**, but some users clocked over 14,000 minutes (~250 hours) over the month.

- Number of days the user opens the app during the month

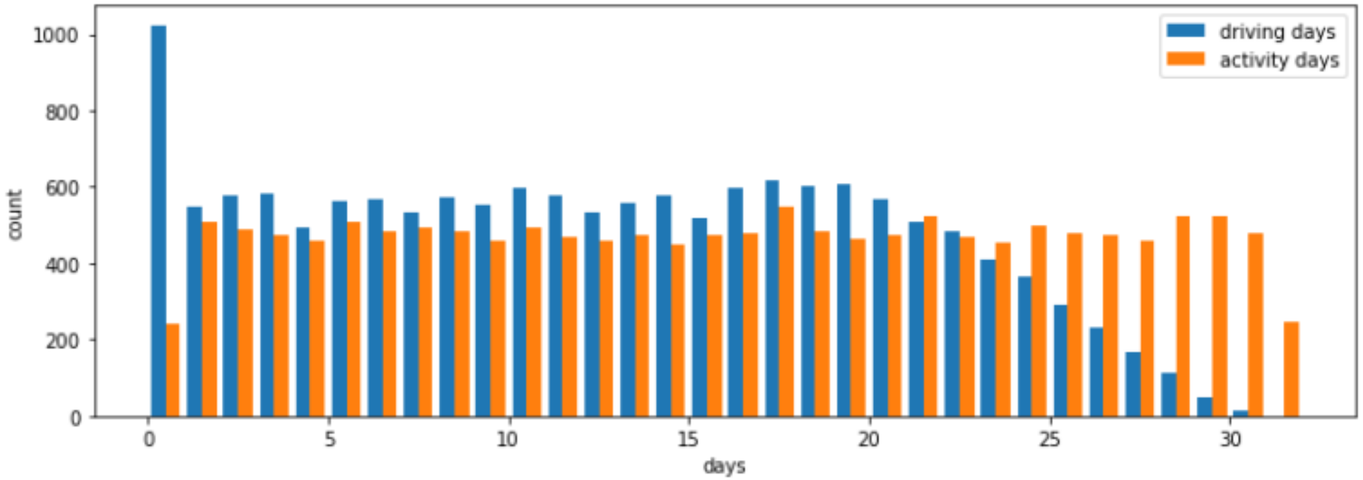


- **Users opened the app a median of 16.** The box plot shows a centered distribution, while the histogram reveals a nearly uniform spread, with about 500 users opening the app on most counts of days.

- Number of days the user drives (at least 1 km) during the month



- The number of days users drove in a month is almost evenly spread and closely matches the number of days they opened the app, except **the driving_days distribution drops off at the higher end.**

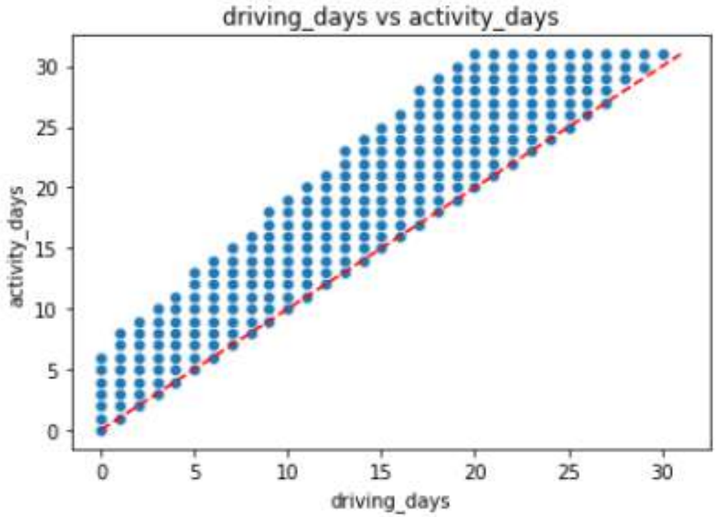


- Since driving_days and activity_days both count days in a month and are closely related, it can be plotted on a single histogram for comparison.
- **About 1,000 users did not drive at all in the month vs. roughly 250 users who did not even open the app.**
- It may look weird that the number of people who never used the app at all (activity_days) is lower than the number who never drove (driving_days). It can also mean the two variables, driving_days and activity_days, are correlated but not identical. **Users probably open the app more than they drive, perhaps just to check routes or change some settings—or maybe accidentally.**
- It's a good idea to ask the Waze data team for clarification, especially since the number of days in the month seems inconsistent between these variables.

- Check the maximum number of days recorded for both driving_days and activity_days.

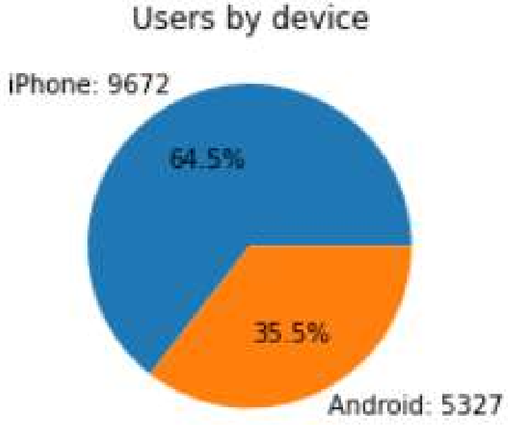
```
print(df["driving_days"].max())
print(df["activity_days"].max())
```

```
30
31
```

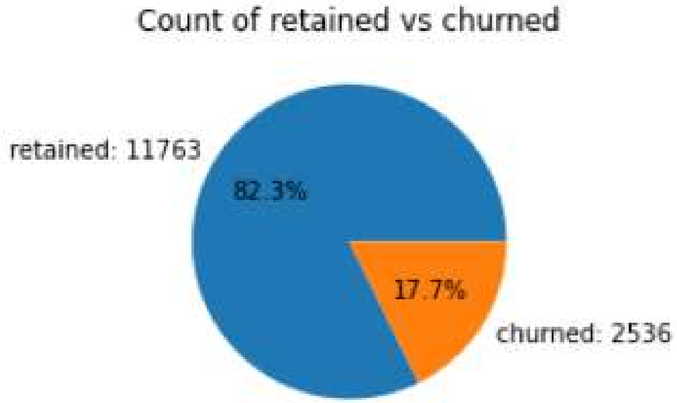


- There is a theoretical limit: if you drive using the app, it counts as an activity day too. So, **drive-days can't exceed activity-days.** None of the data samples break this rule, which is a good sign.

- The type of device a user starts a session with

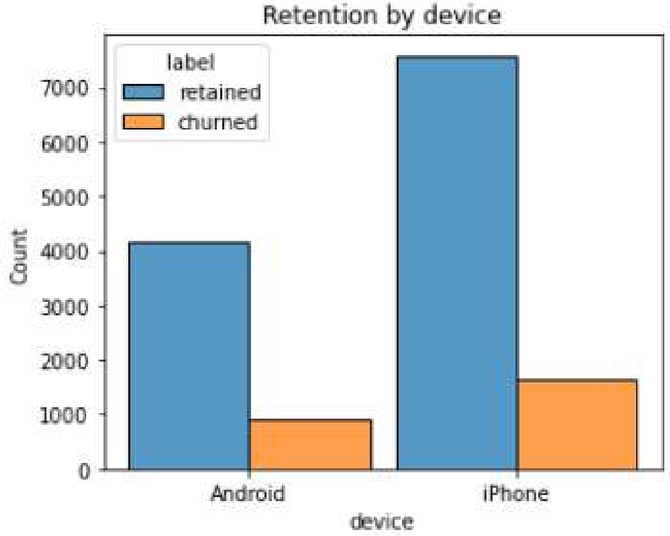


- There are nearly twice as many iPhone users as Android users represented in this data.



- Less than 18% of the users churned.

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



- To create and clean the km_per_driving_day column:

1. Add a new column km_per_driving_day that calculates the average distance driven per driving day for each user.
2. Replace infinite values in the new column—caused by division by zero—with zero using np.inf:.
3. Use `.describe()` on the column to see the results.
4. This will ensure that rows with zero driving days are handled properly, and the column statistics are accurate.

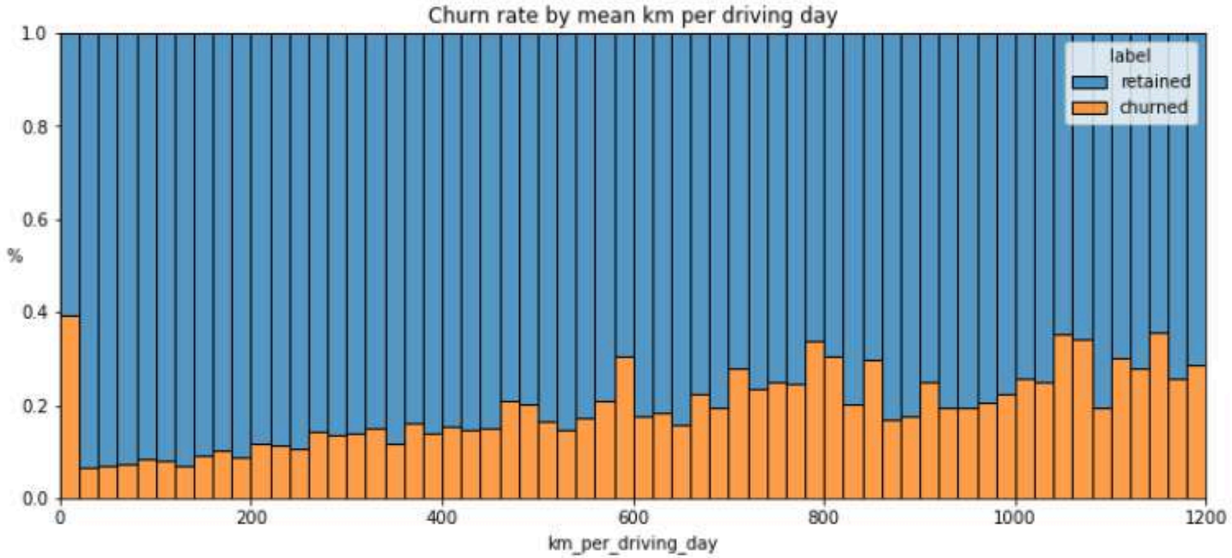
```
df["km_per_driving_day"] = df["driven_km_drives"] / df["driving_days"]
df["km_per_driving_day"].describe()
```

```
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
```

```
df.loc[df["km_per_driving_day"] == np.inf, "km_per_driving_day"] = 0
df["km_per_driving_day"].describe()
```

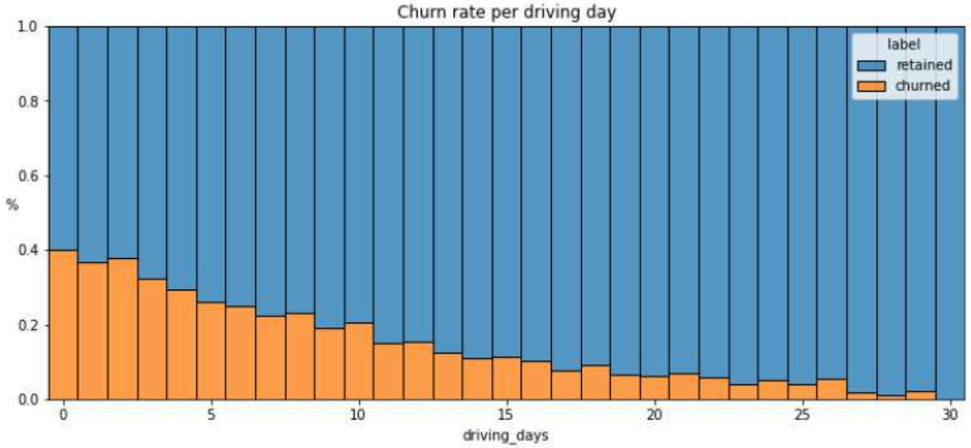
```
count    14999.000000
mean       578.963113
std       1030.094384
min          0.000000
25%        136.238895
50%        272.889272
75%        558.686918
max      15420.234110
Name: km_per_driving_day, dtype: float64
```

- Some users have values in the **km_per_driving_day** column that are above 15,420 km, which does not seem realistic. Driving 1,200 km in one day is quite a stretch, considering (driving 100 km/h for 12 hours). **Remove rows where this column's value is greater than 1,200 km, to address this.**



- Next, plot a histogram of the km_per_driving_day column, excluding those unrealistic values. Use bars with two colors: one showing the percentage of users who churned and the other showing those retained. You can do this using Seaborn's histplot() function with the multiple='fill' parameter. This plot will show the distribution of km_per_driving_day while visualizing churn and retention rates for each group.

- The churn rate tends to increase as the mean daily distance driven increase.
- It’s worth exploring why long-distance drivers are more likely to stop using the app.



- Churn is highest among users who barely used Waze last month. The more they used the app, the less likely they were to churn. For example, 40% of users who didn't use the app at all churned, while no one who used it every day did.

- That makes sense—if heavy app users were churning, it could signal dissatisfaction. On the other hand, people who never use the app might churn because they were unhappy in the past or simply no longer need it—for instance, they might have moved to a city with good public transportation and stopped driving.

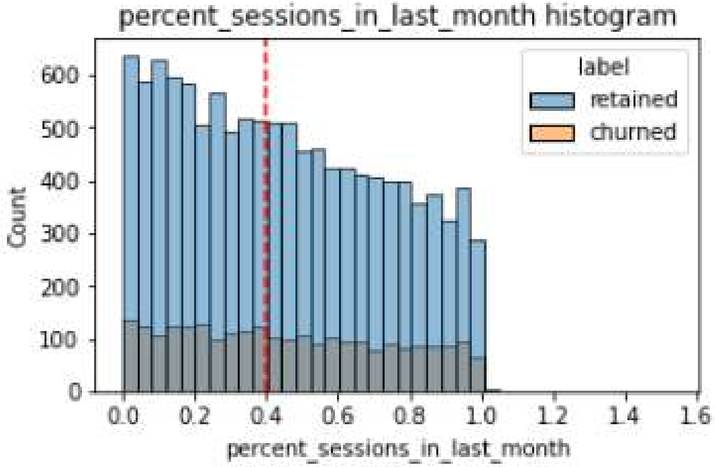
- Create a new column `percent_sessions_in_last_month` that represents the percentage of each user's total sessions that were logged in their last month of use.

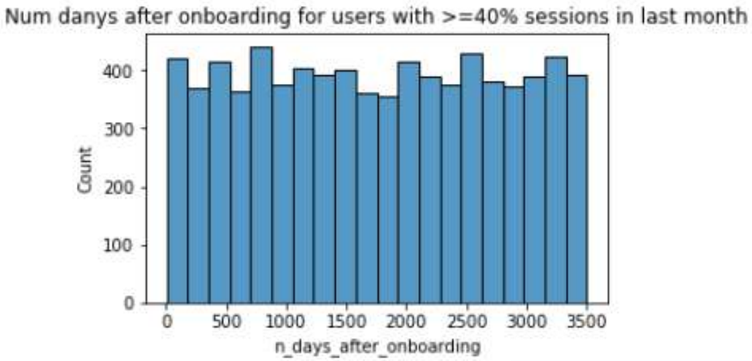
```
df["percent_sessions_in_last_month"] = df["sessions"] / df["total_sessions"]
df["percent_sessions_in_last_month"].median()
```

```
0.42309702992763176
```

```
df["n_days_after_onboarding"].median()
```

```
1741.0
```





- Half of the users had 40% or more of their sessions in just the last month, yet the overall median time since onboarding is almost five years.
- The distribution of n_days_after_onboarding for these users is uniform, which is very unusual. It raises the question of why so many long-time users suddenly increased their app usage in the last month. This might have any possible reasons behind this pattern.

A. Feature Engineering

```
df1["professional_driver"] = np.where((df["drives"] >= 60) & (df["driving_days"] >=15), 1, 0)
print(df1["professional_driver"].value_counts())
df1.groupby(["professional_driver"])["label"].value_counts(normalize=True)
```

```
0    12405
1     2594
Name: professional_driver, dtype: int64

professional_driver  label
0                    retained    0.801202
                     churned     0.198798
1                    retained    0.924437
                     churned     0.075563
Name: label, dtype: float64
```

- The goal is to identify professional drivers. **Create a new feature** called **professional_driver**, where 1 = users with 60+ drives and 15+ driving days in the last month and 0 = all other users.
- Professional drivers have a churn rate of 7.6%, compared to 19.9% for non-professionals. This feature might improve the model's predictions.

```
for col in ["sessions", "drives", "total_sessions", "total_navigations_fav1",
            "total_navigations_fav2", "driven_km_drives", "duration_minutes_drives"]:
    threshold = df1[col].quantile(0.95)
    df1.loc[df1[col] > threshold, col] = threshold
```

- **Impute the outlier values** for these columns. Calculate the **95th percentile** of each column and change to this value any value in the column that exceeds it.

```
RangeIndex: 14999 entries, 0 to 14998
Data columns (total 14 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   label                    14299 non-null  object
 1   sessions                 14999 non-null  int64
 2   drives                   14999 non-null  int64
 3   total_sessions           14999 non-null  float64
 4   n_days_after_onboarding  14999 non-null  int64
 5   total_navigations_fav1   14999 non-null  int64
 6   total_navigations_fav2   14999 non-null  int64
 7   driven_km_drives         14999 non-null  float64
 8   duration_minutes_drives  14999 non-null  float64
 9   activity_days            14999 non-null  int64
 10  driving_days             14999 non-null  int64
 11  device                   14999 non-null  object
 12  km_per_driving_day       14999 non-null  float64
 13  professional_driver      14999 non-null  int64
dtypes: float64(4), int64(8), object(2)
```

```
df1.dropna(subset=["label"])
```

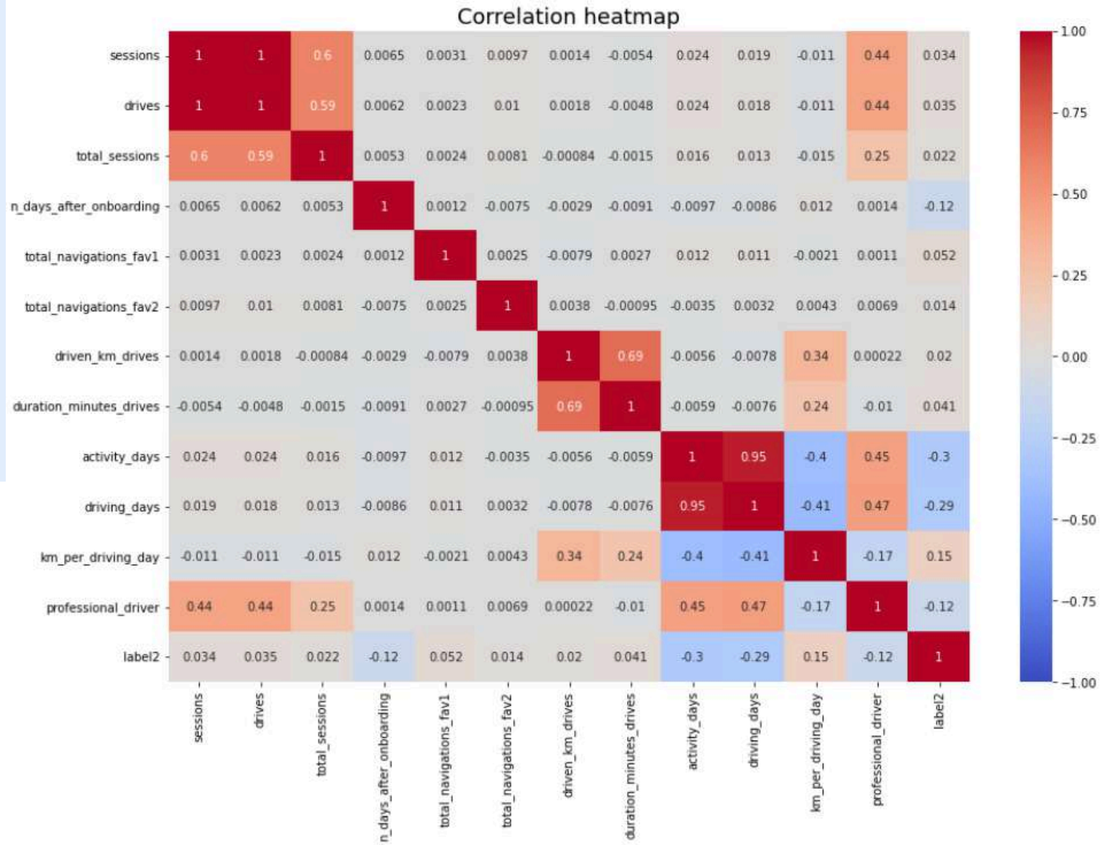
- There are **700 missing values** in column label, making up less than 5% of the data. Use dropna() to **remove** these rows.

```
df1["device2"] = np.where(df1["device"] == 'Android', 0, 1)
```

```
df1["label2"] = np.where(df1["label"] == "churned", 1, 0)
```

- **Encode categorical variables by converting the device column** to binary. Assign 0 to Android and 1 to iPhone. Save it as device2 to keep the original label unchanged.
- Encode categorical variables by converting **the label column** to binary. Assign 0 to retained users and 1 to churned users. Save it as label2 to keep the original label unchanged.

B. Construct Model



- Collinearity: Check the correlation among predictor variables.
- Based on the Pearson correlation coefficient, the variables **`sessions` and `drives` are perfectly multicollinear with a value of 1.0**. Additionally, **`driving_days` and `activity_days` have a high correlation of 0.95.**

```
X = df1.drop(columns = ["label", "label2", "device", "sessions", "driving_days"])
y= df1["label2"]
```

- Assign predictor variables (X) and target variable (y).
- Notice : **`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.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)
```

- **Split the data** with a train/test split using the earlier assigned X and y variables. **The target class here is highly imbalanced with 82% retained and 18% churned.** So an unequal train/test split will underrepresent the minority class. **Therefore, stratify is used on y** so that the minority class has proportional representation in the train and test sets.

```
model = LogisticRegression(penalty='none', max_iter=400)
model.fit(X_train, y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=400,
                   multi_class='auto', n_jobs=None, penalty='none',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

- **Instantiate a logistic regression model**. Since your predictors are unscaled, add the argument `penalty = None`.
- **Fit the model** on `X_train` and `y_train`.

```
pd.Series(model.coef_[0], index=X.columns)
```

```
drives                        0.002347
total_sessions               -0.000113
n_days_after_onboarding      -0.000393
total_navigations_fav1        0.001127
total_navigations_fav2        0.000989
driven_km_drives             -0.000035
duration_minutes_drives       0.000127
activity_days                -0.102266
km_per_driving_day            0.000007
professional_driver          -0.001404
device2                      -0.001164
dtype: float64
```

```
model.intercept_
```

```
array([-0.00173325])
```

- **The coefficients** are in order of how the variables are listed in the dataset. The coefficients represent the change in the log odds of the target variable for every one unit increase in X.

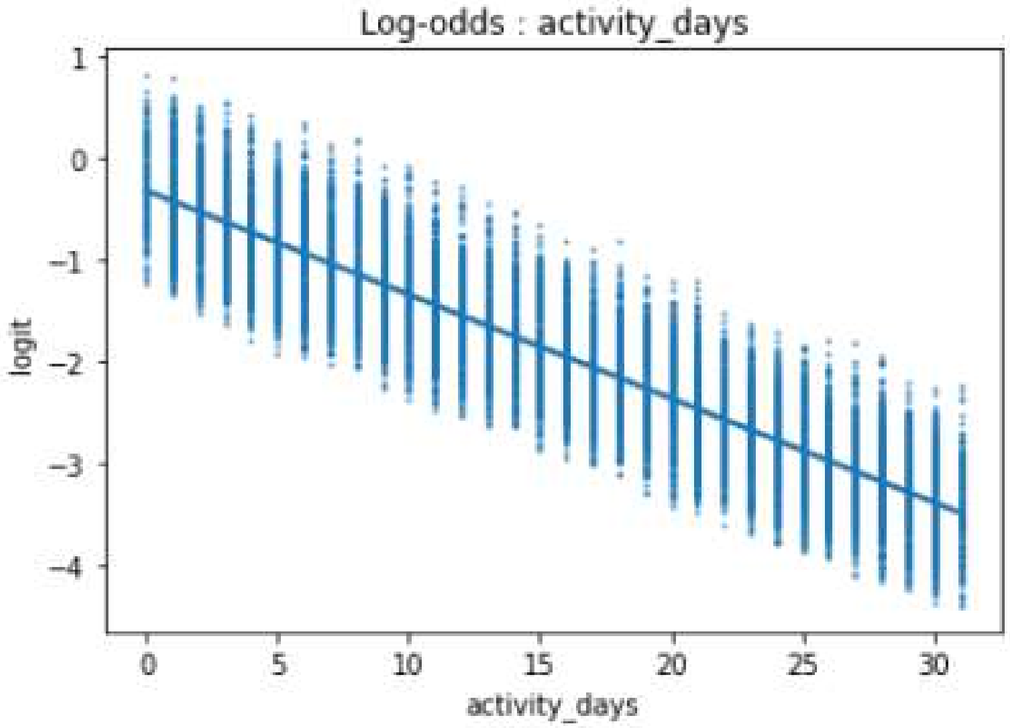
```
training_probabilities = model.predict_proba(X_train)
```

- **Check final assumption: Generate the probability of response** for each sample in the training data. The first column is the probability of the user not churning, and the second column is the probability of the user churning.

```
logit_data = X_train.copy()
logit_data["logit"] = [np.log(prob[1] / prob[0]) for prob in training_probabilities]
```

- In logistic regression, the predictor and dependent variables don't need to have a linear relationship, but **the log-odds (logit) of the dependent variable should be linear with respect to the predictor.**
- To verify this, create a new dataframe called logit_data as a copy of df. Add a new column, logit, to logit_data that contains the logit values for each user.

```
sns.regplot(x='activity_days', y='logit', data=logit_data, scatter_kws={'s': 2, 'alpha': 0.5})
plt.title('Log-odds : activity_days');
```



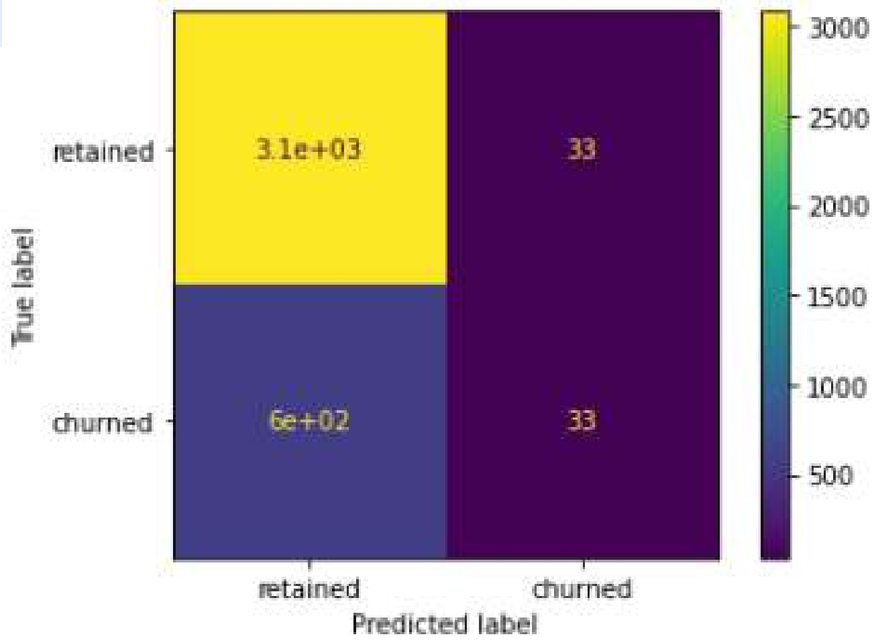
- **Verify the linear relationship between X and the estimated log-odds (logit) using a regplot.**

C. Results and Evaluation

```
target_labels = ["retained", "churned"]
print(classification_report(y_test, y_preds, target_names=target_labels))
```

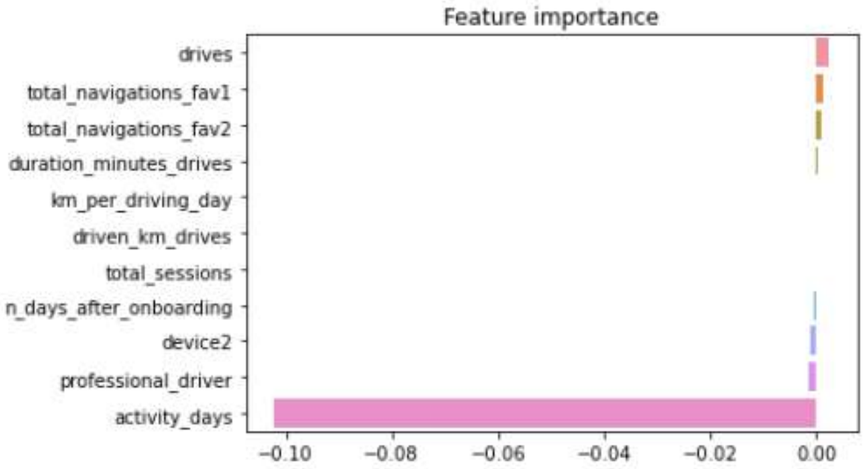
	precision	recall	f1-score	support
retained	0.84	0.99	0.91	3116
churned	0.50	0.05	0.09	634
accuracy			0.83	3750
macro avg	0.67	0.52	0.50	3750
weighted avg	0.78	0.83	0.77	3750

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




```
feature_importance = list(zip(X_train.columns, model.coef_[0]))
feature_importance = sorted(feature_importance, key=lambda x: x[1], reverse=True)
feature_importance
```

```
[('drives', 0.0023471356247837O5),
 ('total_navigations_fav1', 0.0011265968643180656),
 ('total_navigations_fav2', 0.000898487427392623),
 ('duration_minutes_drives', 0.00012710408444657686),
 ('km_per_driving_day', 6.5871171433344866e-06),
 ('driven_km_drives', -3.498232803413078e-05),
 ('total_sessions', -0.00011259285882929962),
 ('n_days_after_onboarding', -0.0003934678337O459775),
 ('device2', -0.0011643707746116048),
 ('professional_driver', -0.0014041246839772588),
 ('activity_days', -0.10226628701296434)]
```



- **The most important feature in the model, activity_days, was that of negative correlation with user churn.**
- As part of logistic regression, features can interact counter-intuitively. These interactions make such models far more predictive but harder to understand. The model did badly as evidenced by the very poor recall score. However it may still be used as a guide for further investigation.
- New feature creation would possibly improve predictions. As such, one new variable, professional_driver ranked number three in the list of the most predictive variables.

A. Feature Engineering

```
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)
```

```
Index: 14299 entries, 0 to 14998
Data columns (total 21 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   label                           14299 non-null  object
 1   sessions                        14299 non-null  int64
 2   drives                          14299 non-null  int64
 3   total_sessions                  14299 non-null  float64
 4   n_days_after_onboarding         14299 non-null  int64
 5   total_navigations_fav1          14299 non-null  int64
 6   total_navigations_fav2          14299 non-null  int64
 7   driven_km_drives                14299 non-null  float64
 8   duration_minutes_drives         14299 non-null  float64
 9   activity_days                   14299 non-null  int64
 10  driving_days                    14299 non-null  int64
 11  device                          14299 non-null  object
 12  km_per_driving                  14299 non-null  float64
 13  percent_sessions_in_last_month  14299 non-null  float64
 14  professional_driver             14299 non-null  int64
 15  total_sessions_per_day          14299 non-null  float64
 16  km_per_hour                     14299 non-null  float64
 17  km_per_drive                    14299 non-null  float64
 18  percent_of_sessions_to_favorite 14299 non-null  float64
 19  device2                         14299 non-null  int64
 20  label2                          14299 non-null  int64
dtypes: float64(9), int64(10), object(2)
```

- **Remove `ID`**, since it doesn't contain any information relevant to churn.

```
df = df.drop("ID", axis=1)
```

- **Add `km_per_driving_day` column**, the mean number of kilometers driven on each driving day in the last month for each user.

```
df["km_per_driving"] = df["driven_km_drives"] / df["driving_days"]
df.loc[df["km_per_driving"] == np.inf, "km_per_driving"] = 0
```

- **Add `percent_sessions_in_last_month` column**, the percentage of each user's total sessions that were logged in their last month of use.

```
df["percent_sessions_in_last_month"] = df["sessions"] / df["total_sessions"]
```

- **Add `professional_driver` column**, where 1 = users with 60+ drives and 15+ driving days in the last month and 0 = all other users.

```
df1["professional_driver"] = np.where((df["drives"] >= 60) & (df["driving_days"] >=15), 1, 0)
```

- **Add `total_sessions_per_day` column**, the mean number of sessions per day _since onboarding_.

```
df["total_sessions_per_day"] = df["total_sessions"] / df["n_days_after_onboarding"]
```

- **Add `km_per_hour` column**, the mean kilometers per hour driven in the last month.

```
df["km_per_hour"] = df["driven_km_drives"] / (df["duration_minutes_drives"] / 60)
```

- **Add `km_per_drive`column**, the mean number of kilometers per drive made in the last month for each user.

```
df["km_per_drive"] = df["driven_km_drives"] / df["drives"]
df.loc[df["km_per_drive"] == np.inf, "km_per_drive"] = 0
```

- **Add `percent_of_sessions_to_favorite`**, the percentage of total sessions that were used to navigate to one of the users' favorite places.

```
df["percent_of_sessions_to_favorite"] = (df["total_navigations_fav1"] + df["total_navigations_fav2"]) / df["total_sessions"]
```

- People whose drives to non-favorite places make up a higher percentage of their total drives might be less likely to churn, since they're making more drives to less familiar places.

- **Encode categorical variables by converting the `device` column** to binary. Assign 0 to Android and 1 to iPhone. **Save it as `device2`** to keep the original label unchanged.

```
df1["device2"] = np.where(df1["device"] == 'Android', 0, 1)
```

- **Encode categorical variables by converting the `label` column** to binary. Assign 0 to retained users and 1 to churned users. **Save it as `label2`** to keep the original label unchanged.

```
df1["label2"] = np.where(df1["label"] == "churned", 1, 0)
```

- There are **700 missing values** in column label, making up less than 5% of the data. Use dropna() to **remove** these rows.

```
df = df.dropna(subset=["label"])
```

- From earlier analysis, we know that many columns have outliers. However, tree-based models handle outliers well, so there's **no need to make any imputations.**

```
df["label"].value_counts(normalize=True)
```

```
label
retained    0.822645
churned     0.177355
Name: proportion, dtype: float64
```

- About 18% of users in this **dataset** churned, which is **unbalanced** but still easily handled without rebalancing the classes.
- When choosing a valuation metric, accuracy might not be ideal. On an imbalanced dataset, a model can have a high accuracy but poor ability in predicting the minority class. Given the risks of false positives—predicting a user will churn when they won't—are very minimal, meaning no significant harm, loss, or consequence, **focus on the recall score in order to choose the best model.**

B. Construct Model

- Split the data **80/20 into an interim training set and a test set**. Stratify the splits, and set the random state to 42.
- Split the interim training set **75/25 into a training set and a validation set**, yielding a final ratio of 60/20/20 for training/validation/test sets. Stratify the splits and set the random state to 42.

```
X = df.drop(columns=["label", "label2", "device"])
y = df["label2"]
X_tr, X_test, y_tr, y_test = train_test_split(X, y, stratify=y, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_tr, y_tr, stratify=y_tr, test_size=0.25, random_state=42)
```

- The dataset is divided into training, validation, and test sets in the 8,579 : 2,860 : 2,860.

```
for x in [X_train, X_val, X_test]
    print(len(x))
```

```
8579
2860
2860
```

```
rf = RandomForestClassifier(random_state=42)
cv_params = {'max_depth': [None],
             'max_features': [1.0],
             'max_samples': [1.0],
             'min_samples_leaf': [2],
             'min_samples_split': [2],
             'n_estimators': [300]}
scoring = ['accuracy', 'precision', 'recall', 'f1']
rf_cv = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='recall')
rf_cv.fit(X_train, y_train)
```

	model	precision	recall	F1	accuracy
0	RF cv	0.457163	0.126782	0.198445	0.81851

- **Asside from the accuracy, the scores aren't that good.** However,the recall from logistic regression model was ~0.05, which means that this model has 150% better recall and about the same accuracy, and it was trained on less data.

```
xgb = XGBClassifier(objective= 'binary:logistic', random_state=42)
cv_params = {'max_depth': [8],
             'min_child_weight': [3],
             'learning_rate': [0.01, 0.1],
             'n_estimators': [30]}cv_params = {'max_depth': [8],
             'min_child_weight': [3],
             'learning_rate': [0.01, 0.1],
             'n_estimators': [30]}
scoring = ['accuracy', 'precision', 'recall', 'f1']
xgb_cv = GridSearchCV(xgb, cv_params, scoring=scoring, cv=4, refit='recall')
xgb_cv.fit(X_train, y_train)
```

	model	precision	recall	F1	accuracy
0	RF cv	0.457163	0.126782	0.198445	0.818510
0	XGB cv	0.447306	0.139273	0.212365	0.816761

- This model fits the data better than the random forest model. Its recall score is higher than the logistic regression and random forest model's, while keeping similar accuracy and precision.

	model	precision	recall	F1	accuracy
0	RF cv	0.457163	0.126782	0.198445	0.818510
0	XGB cv	0.447306	0.139273	0.212365	0.816761
0	RF val	0.445255	0.120316	0.189441	0.817483
0	XGB val	0.461538	0.153846	0.230769	0.818182

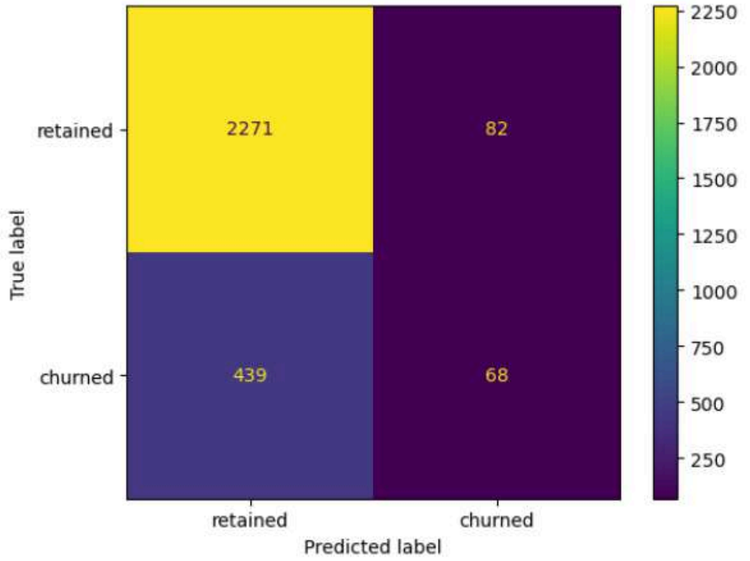
- The scores went down from the training scores across all metrics, but only by very little. This means that the model did not overfit the training data.
- The XGBoost model's validation scores were higher, but only very slightly.

C. Results and Evaluation

```
xgb_test_preds = xgb_cv.best_estimator_.predict(X_test)
xgb_test_scores = get_test_scores('XGB test', xgb_test_preds, y_test)
results = pd.concat([results, xgb_test_scores], axis=0)
results
```

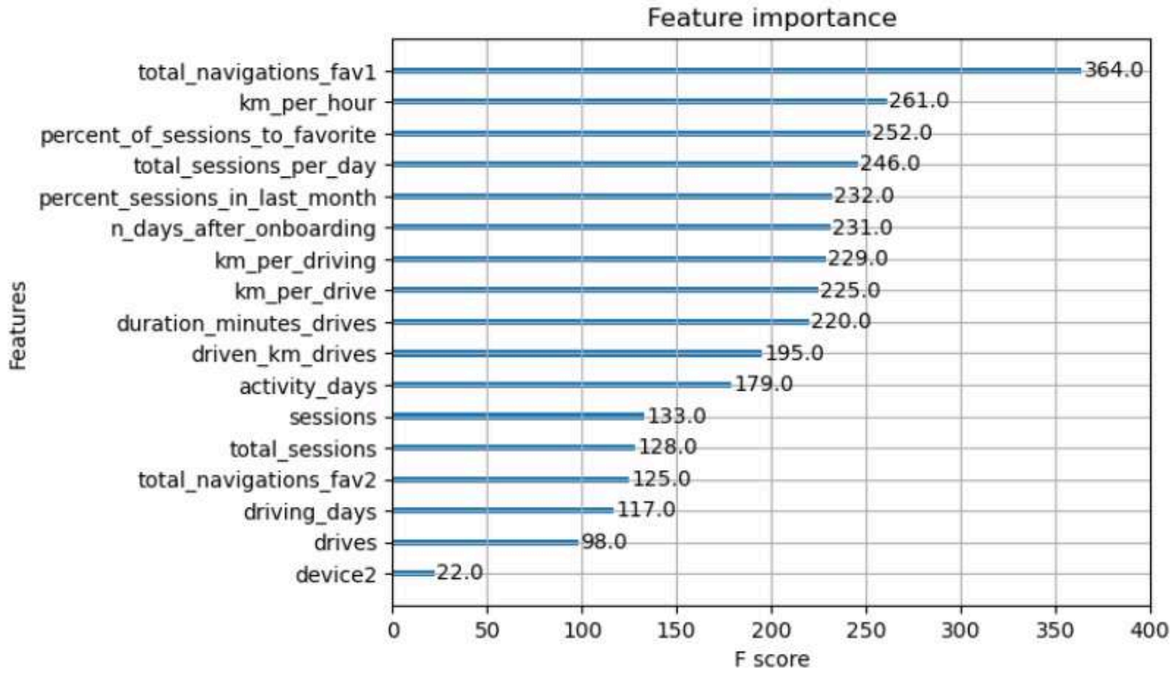
	model	precision	recall	F1	accuracy
0	RF cv	0.457163	0.126782	0.198445	0.818510
0	XGB cv	0.447306	0.139273	0.212365	0.816761
0	RF val	0.445255	0.120316	0.189441	0.817483
0	XGB val	0.461538	0.153846	0.230769	0.818182
0	XGB test	0.453333	0.134122	0.207002	0.817832

```
cm = confusion_matrix(y_test, xgb_test_preds, labels=xgb_cv.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['retained', 'churned'])
disp.plot();
```



- The recall and 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.
- The model predicted four times as many false negatives than it did false positives.

```
plot_importance(xgb_cv.best_estimator_);
```



- The XGBoost model utilized more features than the logistic regression model, which heavily relied on a single feature (activity_days) for its predictions.
- Feature engineering is often a simple and effective way to improve model performance. The important features may vary between different models. Never discard some features as unimportant without going deep into their relationship with the target variable. Sometimes, the reason for different feature importance comes from complicated interactions among features.

D. Summary and Recommendations

- It has a low recall and hence is not strong enough to make reliable predictions. However, it could be useful for exploratory analysis. Not suitable to drive important business decisions.
- The splitting of the data into three sets leaves less data for training compared to a two-way split. However, using a validation set for model selection and testing the final model on a separate test set provides a better estimate of future performance than selecting a model based on test data.
- Logistic regression is easier to interpret since it assigns coefficients to features, which show the importance and whether they have a positive or negative effect on the predictions.
- Tree-based ensemble models usually outperform logistic regression in terms of accuracy with less data cleaning and assumptions about the distribution of data. They are good to go when the priority is predictive power.
- Feature engineering based on domain knowledge often improves the predictions. Further model building with different combinations of features may also be helpful since it reduces noise from non-informative features.
- Additional data that could help in improving the model may include: drive level information, for example, drive time and location, fine-grained user interaction data - for example, how users report or confirm a hazard.or counts of unique starting and ending locations entered by drivers each month.

THANK YOU!

by : Ana Farida