Waze Project

Milestone 6 / 6A - Build a machine learning model. Communicate final insights

Build a machine learning model

The purpose of this model is to find factors that drive user churn.

The goal of this model is to predict whether or not a Waze user is retained or churned.

This notebook has four parts:

Part 1: Imports and Data Loading

Part 2: Feature engineering

Part 3: Modeling

Part 4: Insights and Conclusion

Part 1: Imports and data loading

In [1]:
```
# Import packages for data manipulation
import numpy as np
import pandas as pd

# Import packages for data visualization
import matplotlib.pyplot as plt

# This lets us see all of the columns, preventing Juptyer from redacting them.
pd.set_option('display.max_columns', None)

# Import packages for data modeling
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import roc_auc_score, roc_curve, auc
from sklearn.metrics import accuracy_score, precision_score, recall_score,\
f1_score, confusion_matrix, ConfusionMatrixDisplay, RocCurveDisplay, PrecisionRecallDisp

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

# from google.colab import drive
# drive.mount('/content/drive', force_remount=True)
```

In [2]:
```
# Import dataset
df0 = pd.read_csv('waze_dataset.csv')
```

In [3]:
```
# Inspect the first five rows
df0.head()
```

Out[3]:

	ID	label	sessions	drives	total_sessions	n_days_after_onboarding	total_navigations_fav1	total_navigati
0	0	retained	283	226	296.748273	2276	208	
1	1	retained	133	107	326.896596	1225	19	
2	2	retained	114	95	135.522926	2651	0	
3	3	retained	49	40	67.589221	15	322	
4	4	retained	84	68	168.247020	1562	166	

Part 2: Feature engineering

In [4]:
```
# Copy the df0 dataframe
df = df0.copy()
```

In [5]:
```
df.info()
```

```
<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
```

`km_per_driving_day`

Creates a feature representing the mean number of kilometers driven on each driving day in the last month for each user.

In [6]:
```
# 1. Create `km_per_driving_day` feature
df['km_per_driving_day'] = df['driven_km_drives'] / df['driving_days']

# 2. Get descriptive stats
df['km_per_driving_day'].describe()
```

Out[6]:
```
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
```

In [7]:
```
# 1. Convert infinite values to zero
df.loc[df['km_per_driving_day']==np.inf, 'km_per_driving_day'] = 0
```

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

Out[7]:
```
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
```

`percent_sessions_in_last_month`

Creates 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.

In [8]:
```
# 1. Create `percent_sessions_in_last_month` feature
df['percent_sessions_in_last_month'] = df['sessions'] / df['total_sessions']

# 2. Get descriptive stats
df['percent_sessions_in_last_month'].describe()
```

Out[8]:
```
count    14999.000000
mean         0.449255
std          0.286919
min          0.000000
25%          0.196221
50%          0.423097
75%          0.687216
max          1.530637
Name: percent_sessions_in_last_month, dtype: float64
```

`professional_driver`

Creates a new, binary feature called `professional_driver` that is a 1 for users who had 100 or more drives **and** drove on 20+ days in the last month.

In [9]:
```
# Create `professional_driver` feature
df['professional_driver'] = np.where((df['drives'] >= 60) & (df['driving_days'] >= 15),
```

`total_sessions_per_day`

Creates a new column that represents the mean number of sessions per day *since onboarding*.

In [10]:
```
# Create `total_sessions_per_day` feature
df['total_sessions_per_day'] = df['total_sessions'] / df['n_days_after_onboarding']
```

In [11]:
```
# Get descriptive stats
df['total_sessions_per_day'].describe()
```

Out[11]:
```
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
```

`km_per_hour`

Creates a column representing the mean kilometers per hour driven in the last month.

In [12]:
```
# Create `km_per_hour` feature
df['km_per_hour'] = df['driven_km_drives'] / df['duration_minutes_drives'] / 60
df['km_per_hour'].describe()
```

Out[12]:
```
count    14999.000000
mean         0.052887
std          0.092965
min          0.020004
25%          0.025196
50%          0.033995
75%          0.053647
max          6.567478
Name: km_per_hour, dtype: float64
```

`km_per_drive`

Creates a column representing the mean number of kilometers per drive made in the last month for each user.

In [13]:
```
# Create `km_per_drive` feature
df['km_per_drive'] = df['driven_km_drives'] / df['drives']
df['km_per_drive'].describe()
```

Out[13]:
```
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 [14]:
```
# 1. Convert infinite values to zero
df.loc[df['km_per_drive']==np.inf, 'km_per_drive'] = 0

# 2. Confirm that it worked
df['km_per_drive'].describe()
```

Out[14]:
```
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
```

`percent_of_sessions_to_favorite`

Creates a new column that represents the percentage of total sessions that were used to navigate to one of the users' favorite places.

This serves as a substitute indicator for the percentage of all drives that are made to a preferred location.

As the dataset lacks information on the total number of drives since the initial use, the total number of sessions can be considered a reasonable estimate.

Individuals who have a higher proportion of drives to non-preferred destinations in relation to their total trips may exhibit a lower likelihood of churn, as they are driving to unfamiliar places more frequently.

In [15]:
```
# Create `percent_of_sessions_to_favorite` feature
df['percent_of_drives_to_favorite'] = (
    df['total_navigations_fav1'] + df['total_navigations_fav2']) / df['total_sessions']

# Get descriptive stats
df['percent_of_drives_to_favorite'].describe()
```

Out[15]:
```
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
```

Drop missing values

In [16]:
```
# Drop rows with missing values
df = df.dropna(subset=['label'])
```

Outliers

Tree-based models are resilient to outliers, so there is no need to make any imputations.

Variable encoding

Dummying features

Creates a new, binary column called `device2` that encodes user devices as follows:

- `Android` -> `0`
- `iPhone` -> `1`

In [17]:
```
# Create new `device2` variable
df['device2'] = np.where(df['device']=='Android', 0, 1)
df[['device', 'device2']].tail()
```

Out[17]:

	device	device2
14994	iPhone	1
14995	Android	0
14996	iPhone	1
14997	iPhone	1
14998	iPhone	1

Target encoding

Changes the data type of the `label` column to be binary. This change is needed to train the models.

Assigns a `0` for all `retained` users.

Assigns a `1` for all `churned` users.

Variables saved as `label2` so as not to overwrite the original `label` variable.

In [18]:
```
# Create binary `label2` column
df['label2'] = np.where(df['label']=='churned', 1, 0)
df[['label', 'label2']].tail()
```

Out[18]:

	label	label2
14994	retained	0
14995	retained	0
14996	retained	0
14997	churned	1
14998	retained	0

Feature selection

The only feature that can be cut is `ID`, since it doesn't contain any information relevant to churn.

`device` won't be used simply because it's a copy of `device2`.

Drops `ID` from the `df` dataframe.

In [19]:
```
# Drop `ID` column
df = df.drop(['ID'], axis=1)
```

Evaluation metric

Examines the class balance of the target variable.

In [20]:
```
# Get class balance of 'label' col
df['label'].value_counts(normalize=True)
```

Out[20]:
```
label
retained    0.822645
churned     0.177355
Name: proportion, dtype: float64
```

Around 18% of the users included in this dataset experienced churn. Although the dataset is imbalanced, it can be still modeled without requiring any class rebalancing.

We will select the model based on recall.

Modeling workflow and model selection process

The final modeling dataset contains 14,299 samples. This is towards the lower end of what might be considered sufficient to conduct a robust model selection process, but still doable.

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

Split the data

1. Defines a variable `X` that isolates the features.

2. Defines a variable `y` that isolates the target variable (`label2`).

3. Splits the data 80/20 into an interim training set and a test set.

4. Splits 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.

In [21]:
```
# 1. Isolate X variables
X = df.drop(columns=['label', 'label2', 'device'])

# 2. Isolate y variable
y = df['label2']

# 3. Split into train and test sets
X_tr, X_test, y_tr, y_test = train_test_split(X, y, stratify=y,
                                              test_size=0.2, random_state=42)

# 4. Split into train and validate sets
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)
```

In [22]:
```
for x in [X_train, X_val, X_test]:
    print(len(x))
```

```
8579
2860
2860
```

This is consistent with what was expected.

Part 3: Modeling

Random forest

Begin with using `GridSearchCV` to tune a random forest model.

1. Instantiates the random forest classifier `rf` and sets the random state.

2. Creates a dictionary `cv_params` of any of the following hyperparameters and their corresponding values to tune.

   - `max_depth`
   - `max_features`
   - `max_samples`
   - `min_samples_leaf`
   - `min_samples_split`
   - `n_estimators`

3. Defines a dictionary `scoring` of scoring metrics for GridSearch to capture (precision, recall, F1 score, and accuracy).

4. Instantiates the `GridSearchCV` object `rf_cv`. Passes to it as arguments:

   - estimator= `rf`
   - param_grid= `cv_params`
   - scoring= `scoring`
   - cv: define the number of cross-validation folds you want (`cv=_`)
   - refit: indicate which evaluation metric you want to use to select the model (`refit=_`)

   `refit` should be set to `'recall'`.

In [23]:
```python
# 1. Instantiate the random forest classifier
rf = RandomForestClassifier(random_state=42)

# 2. Create a dictionary of hyperparameters to tune
cv_params = {'max_depth': [None],
             'max_features': [1.0],
             'max_samples': [1.0],
             'min_samples_leaf': [2],
             'min_samples_split': [2],
             'n_estimators': [300],
             }

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

# 4. Instantiate the GridSearchCV object
rf_cv = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='recall')
```

In [24]:
```python
%%time
rf_cv.fit(X_train, y_train)
```

```
CPU times: user 1min 56s, sys: 27.3 ms, total: 1min 56s
Wall time: 1min 56s
```

Out[24]:
```
            GridSearchCV
estimator: RandomForestClassifier
      RandomForestClassifier
```

The best average score across all the validation folds.

In [25]:
```python
# Examine best score
rf_cv.best_score_
```

Out[25]:
```
0.12678201409034398
```

The best combination of hyperparameters.

In [26]:
```python
# Examine best hyperparameter combo
rf_cv.best_params_
```

Out[26]:
```
{'max_depth': None,
 'max_features': 1.0,
 'max_samples': 1.0,
 'min_samples_leaf': 2,
 'min_samples_split': 2,
 'n_estimators': 300}
```

Creates a `make_results()` function to output all of the scores of the model.

In [27]:
```python
def make_results(model_name:str, model_object, metric:str):
    '''
    Arguments:
        model_name (string): what you want the model to be called in the output table
        model_object: a fit GridSearchCV object
        metric (string): precision, recall, f1, or accuracy

    Returns a pandas df with the F1, recall, precision, and accuracy scores
    for the model with the best mean 'metric' score across all validation folds.
    '''

    # Create dictionary that maps input metric to actual metric name in GridSearchCV
    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[metric]].idxmax(), :

    # 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
```

Passes the `GridSearch` object to the `make_results()` function.

In [28]:
```python
results = make_results('RF cv', rf_cv, 'recall')
results
```

Out[28]:

	model	precision	recall	F1	accuracy
0	RF cv	0.458198	0.126782	0.198534	0.818626

Apart from the accuracy, the scores are not particularly impressive. It is worth noting that with the previously constructed logistic regression model, the recall was approximately 0.09. This indicates that the current model exhibits a 33% improvement in recall while maintaining a similar level of accuracy, despite being trained on a smaller dataset.

We could fine-tune the hyperparameters in an attempt to achieve a higher score. There is a possibility of making slight improvements to the model.

XGBoost

1. Instantiates the XGBoost classifier `xgb` and set `objective='binary:logistic'`. Also sets the random state.

2. Creates a dictionary `cv_params` of the following hyperparameters and their corresponding values to tune:

    - `max_depth`
    - `min_child_weight`
    - `learning_rate`
    - `n_estimators`
3. Defines a dictionary `scoring` of scoring metrics for grid search to capture (precision, recall, F1 score, and accuracy).

4. Instantiates the `GridSearchCV` object `xgb_cv`. Passes to it as arguments:

    - estimator= `xgb`
    - param_grid= `cv_params`
    - scoring= `scoring`
    - cv: define the number of cross-validation folds you want (`cv=_`)
    - refit: indicate which evaluation metric you want to use to select the model (`refit='recall'`)

In [29]:
```python
# 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='recall')
```

Fits the model to the `X_train` and `y_train` data.

In [30]:
```python
%%time
xgb_cv.fit(X_train, y_train)
```

```
CPU times: user 4min 14s, sys: 1.9 s, total: 4min 16s
Wall time: 2min 10s
```

Out[30]:
```
□        GridSearchCV
□ estimator: XGBClassifier
      □ XGBClassifier
```

The best score from this model.

In [31]:
```python
# Examine best score
```

```
xgb_cv.best_score_
```

Out[31]:
```
0.1734683657963807
```

The best parameters.

In [32]:
```
# Examine best parameters
xgb_cv.best_params_
```

Out[32]:
```
{'learning_rate': 0.1,
 'max_depth': 12,
 'min_child_weight': 3,
 'n_estimators': 300}
```

Uses the `make_results()` function to output all of the scores of the model.

In [33]:
```
# 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[33]:

	model	precision	recall	F1	accuracy
0	RF cv	0.458198	0.126782	0.198534	0.818626
0	XGB cv	0.442586	0.173468	0.248972	0.814780

This model not only outperformed the random forest model in terms of data fitting, but it also achieved a recall score that is nearly twice as high as the recall score obtained by the logistic regression model. It also demonstrates an improvement of almost 50% in recall compared to the random forest model, while maintaining similar levels of accuracy and precision.

Model selection

Random forest

In [34]:
```
# Use random forest model to predict on validation data
rf_val_preds = rf_cv.best_estimator_.predict(X_val)
```

Uses the `get_test_scores()` function to generate a table of scores from the predictions on the validation data.

In [35]:
```
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 in the output tabl
        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 scores for your model
    '''
    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 [36]:
```
# 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[36]:

	model	precision	recall	F1	accuracy
0	RF cv	0.458198	0.126782	0.198534	0.818626
0	XGB cv	0.442586	0.173468	0.248972	0.814780
0	RF val	0.445255	0.120316	0.189441	0.817483

The scores experienced a slight decrease compared to the training scores across all metrics, though with minimal deviation. This suggests that the model did not exhibit overfitting to the training data.

XGBoost

In [37]:
```
# 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[37]:

	model	precision	recall	F1	accuracy
0	RF cv	0.458198	0.126782	0.198534	0.818626
0	XGB cv	0.442586	0.173468	0.248972	0.814780
0	RF val	0.445255	0.120316	0.189441	0.817483
0	XGB val	0.430769	0.165680	0.239316	0.813287

Just like the random forest model, the XGBoost model exhibited slightly lower validation scores. However, it still emerges as the clear champion.

Using the champion model(XGBoost) to predict on test data

In [38]:
```
# 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[38]:

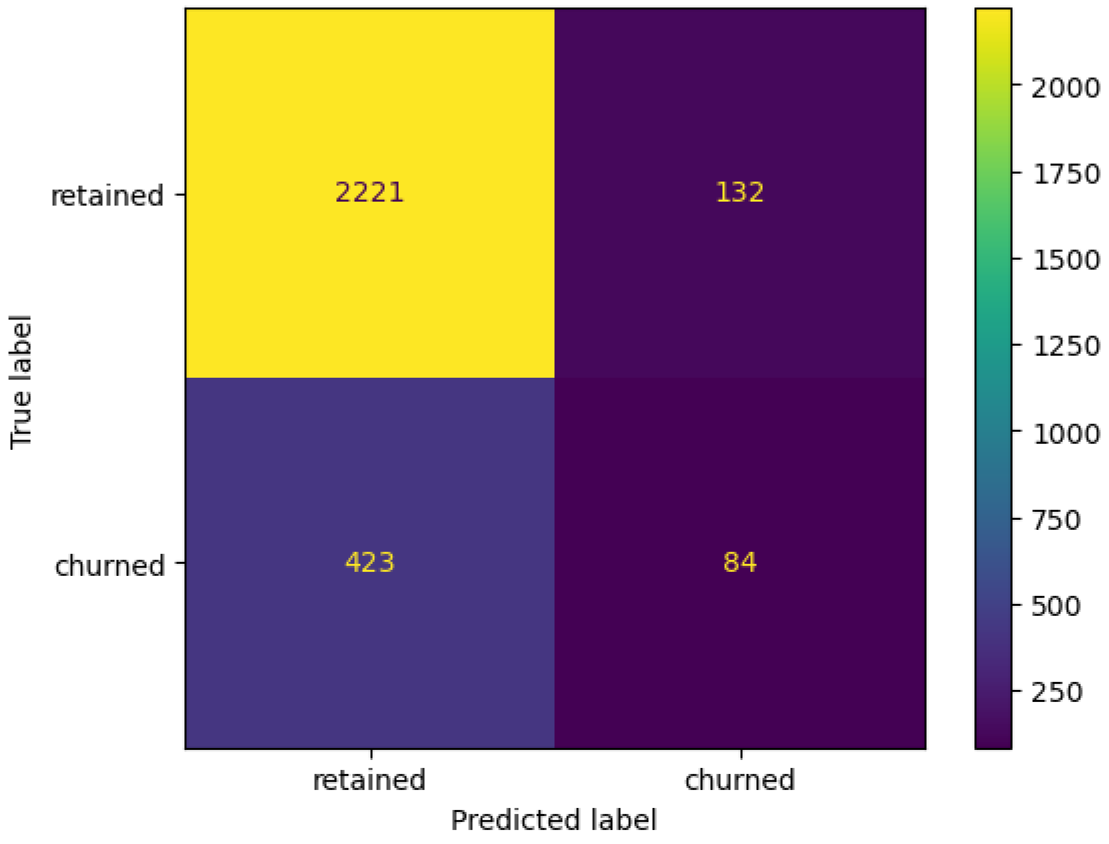
	model	precision	recall	F1	accuracy
0	RF cv	0.458198	0.126782	0.198534	0.818626
0	XGB cv	0.442586	0.173468	0.248972	0.814780
0	RF val	0.445255	0.120316	0.189441	0.817483
0	XGB val	0.430769	0.165680	0.239316	0.813287
0	XGB test	0.388889	0.165680	0.232365	0.805944

The recall remained unchanged from the validation data, while the precision experienced a significant decline, resulting in a slight drop in all other scores. Nevertheless, these variations fall within an acceptable range for performance disparities between validation and test scores.

Task 13. Confusion matrix

In [39]:
```
# Generate array of values for confusion matrix
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();
```

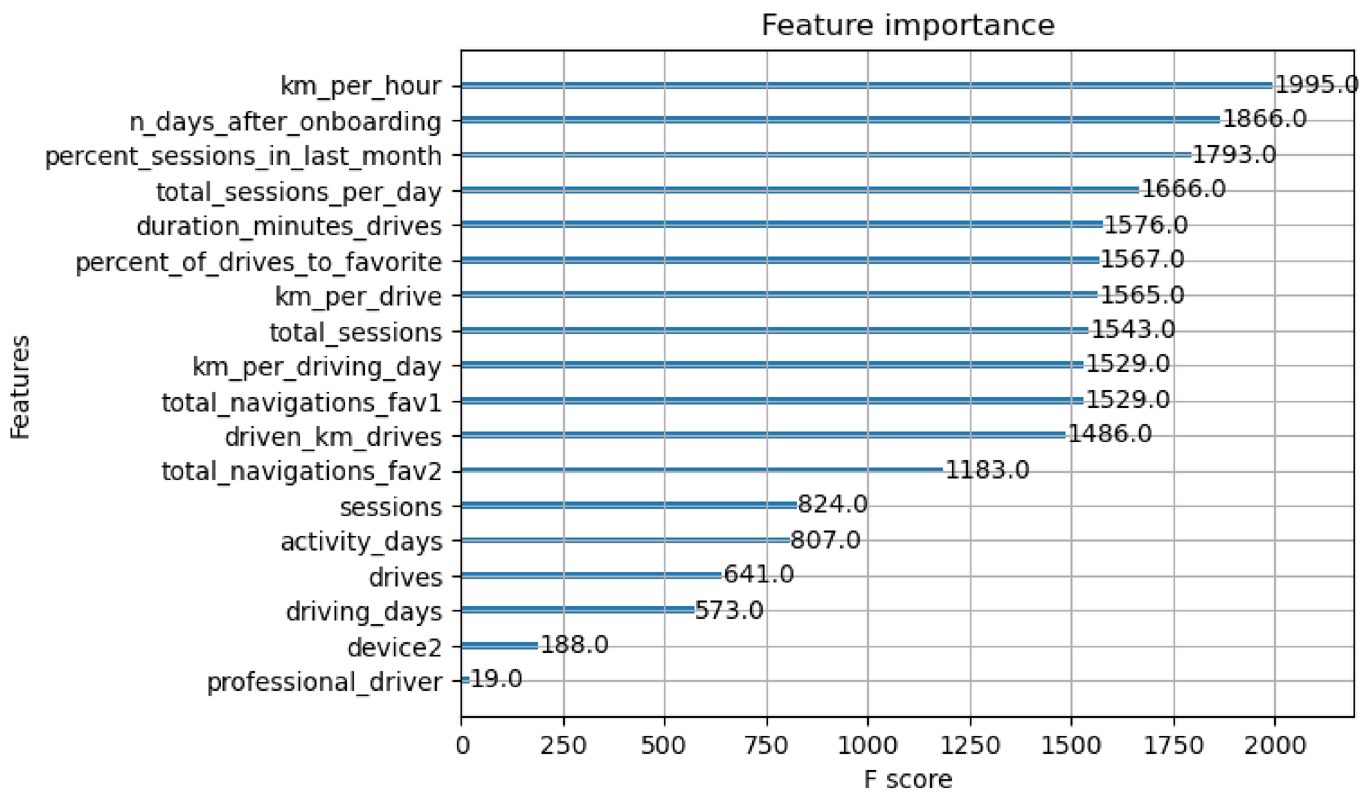


The model's false negatives outnumbered false positives by a factor of three, and it accurately identified only 16.6% of the users who churned.

Feature importance

Uses the `plot_importance` function to inspect the most important features of the final model.

In [40]:
```
plot_importance(xgb_cv.best_estimator_);
```



The XGBoost model utilized a greater number of features compared to the logistic regression model. In particular, the logistic regression model heavily relied on a single feature, namely "activity_days," for its final prediction.

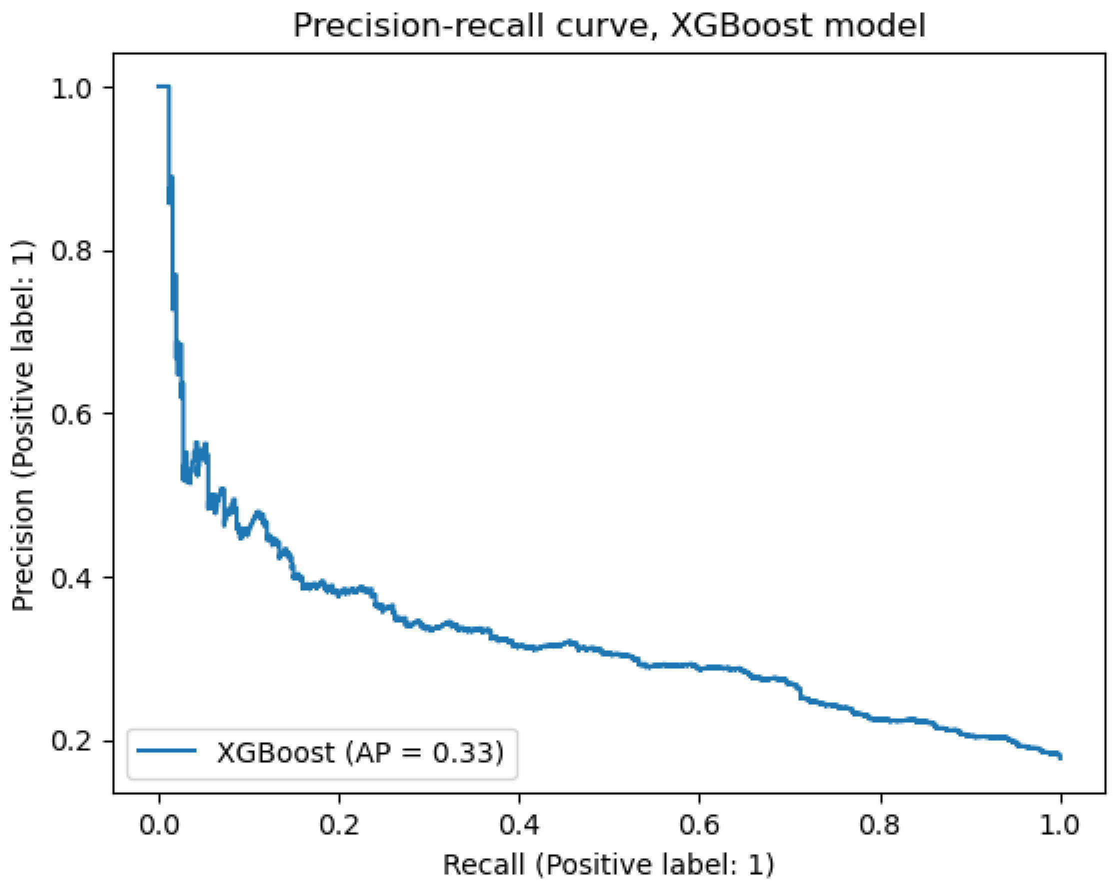
This further emphasizes the significance of feature engineering, as the engineered features played a significant role. They comprised six out of the top 10 features, including three out of the top five.

Additionally, it is worth noting that the selection of important features can vary between different models. Such disparities in selected features are often a result of intricate interactions among features, highlighting the complexity involved in feature selection.

Finding threshold to increase recall

Identify an optimal decision threshold

In [41]:
```
# Plot precision-recall curve
display = PrecisionRecallDisplay.from_estimator(
    xgb_cv.best_estimator_, X_test, y_test, name='XGBoost'
    )
plt.title('Precision-recall curve, XGBoost model');
```



In [42]:
```
# Get predicted probabilities on the test data
predicted_probabilities = xgb_cv.best_estimator_.predict_proba(X_test)
predicted_probabilities
```

Out[42]:
```
array([[0.9765248 , 0.0234752 ],
       [0.5623678 , 0.43763223],
       [0.9964199 , 0.00358006],
       ...,
       [0.80931014, 0.19068986],
       [0.9623124 , 0.03768761],
       [0.64760244, 0.35239756]], dtype=float32)
```

The `predict_proba()` method returns a 2-D array of probabilities where each row represents a user. The first number in the row is the probability of belonging to the negative class, the second number in the row is the probability of belonging to the positive class. (Notice that the two numbers in each row are complimentary to each other and sum to one.)

You can generate new predictions based on this array of probabilities by changing the decision threshold for what is considered a positive response. For example, the following code converts the predicted probabilities to {0, 1} predictions with a threshold of 0.4. In other words, any users who have a value ≥ 0.4 in the second column will get assigned a prediction of `1`, indicating that they churned.

In [43]:
```
# Create a list of just the second column values (probability of target)
probs = [x[1] for x in predicted_probabilities]

# Create an array of new predictions that assigns a 1 to any value >= 0.4
new_preds = np.array([1 if x >= 0.4 else 0 for x in probs])
new_preds
```

Out[43]:
```
array([0, 1, 0, ..., 0, 0, 0])
```

Evaluation metrics when threshold is 0.4

In [44]:
```
# Get evaluation metrics for when the threshold is 0.4
get_test_scores('XGB, threshold = 0.4', new_preds, y_test)
```

Out[44]:

	model	precision	recall	F1	accuracy
0	XGB, threshold = 0.4	0.383333	0.226824	0.285006	0.798252

Previous models for comparison.

In [45]:
```
results
```

Out[45]:

	model	precision	recall	F1	accuracy
0	RF cv	0.458198	0.126782	0.198534	0.818626
0	XGB cv	0.442586	0.173468	0.248972	0.814780
0	RF val	0.445255	0.120316	0.189441	0.817483
0	XGB val	0.430769	0.165680	0.239316	0.813287
0	XGB test	0.388889	0.165680	0.232365	0.805944

Recall and F1 score increased significantly, while precision and accuracy decreased.

In [46]:
```
def threshold_finder(y_test_data, probabilities, desired_recall):
    '''
    Find the threshold that most closely yields a desired recall score.

    Inputs:
        y_test_data: Array of true y values
        probabilities: The results of the `predict_proba()` model method
        desired_recall: The recall that you want the model to have

    Outputs:
        threshold: The threshold that most closely yields the desired recall
        recall: The exact recall score associated with `threshold`
    '''
    probs = [x[1] for x in probabilities]  # Isolate second column of `probabilities`
    thresholds = np.arange(0, 1, 0.001)    # Set a grid of 1,000 thresholds to test

    scores = []
    for threshold in thresholds:
        # Create a new array of {0, 1} predictions based on new threshold
        preds = np.array([1 if x >= threshold else 0 for x in probs])
        # Calculate recall score for that threshold
        recall = recall_score(y_test_data, preds)
        # Append the threshold and its corresponding recall score as a tuple to `scores`
        scores.append((threshold, recall))

    distances = []
    for idx, score in enumerate(scores):
        # Calculate how close each actual score is to the desired score
        distance = abs(score[1] - desired_recall)
        # Append the (index#, distance) tuple to `distances`
        distances.append((idx, distance))

    # Sort `distances` by the second value in each of its tuples (least to greatest)
    sorted_distances = sorted(distances, key=lambda x: x[1], reverse=False)
    # Identify the tuple with the actual recall closest to desired recall
    best = sorted_distances[0]
    # Isolate the index of the threshold with the closest recall score
    best_idx = best[0]
    # Retrieve the threshold and actual recall score closest to desired recall
    threshold, recall = scores[best_idx]
    return threshold, recall
```

Tests the function to find the threshold that results in a recall score closest to 0.5.

In [47]:
```
# Get the predicted probabilities from the champion model
probabilities = xgb_cv.best_estimator_.predict_proba(X_test)

# Call the function
threshold_finder(y_test, probabilities, 0.5)
```

Out[47]:
```
(0.124, 0.5029585798816568)
```

By establishing a threshold of 0.124, the recall comes in at 0.503.

According to the precision-recall curve, a recall score of 0.5 should correspond to a precision value of approximately 0.3.

In [48]:
```
# Create an array of new predictions that assigns a 1 to any value >= 0.124
new_preds = np.array([1 if x >= 0.124 else 0 for x in probs])

# Get evaluation metrics for when the threshold is 0.124
get_test_scores('XGB, threshold = 0.124', new_preds, y_test)
```

Out[48]:

	model	precision	recall	F1	accuracy
0	XGB, threshold = 0.124	0.304296	0.502959	0.379182	0.708042

Part 4: Insights and Conclusion

Questions:

Recommendation to use or not use this model for churn prediction:

- If the model is utilized for significant business decisions, then it falls short in being a robust predictor, as evidenced by its low recall score. However, if the model is solely employed to guide exploratory efforts, it can provide value.

Tradeoffs made by splitting the data into training, validation, and test sets as opposed to just training and test sets:

- Although dividing the data into three sets results in less data available for model training compared to a two-way split, conducting model selection on a separate validation set allows for testing the champion model exclusively on the test set. This approach provides a better estimation of future performance compared to a two-way split where the champion model is selected based on performance on the test data.

Benefits of using a logistic regression model over an ensemble of tree-based models for classification tasks:

- Logistic regression models offer easier interpretability due to the assignment of coefficients to predictor variables. This reveals not only the most influential features in the final predictions but also the directionality of their impact. It indicates whether each feature is positively or negatively correlated with the target in the model's final prediction.

Benefits of using an ensemble of tree-based models over a logistic regression model for classification tasks:

- Tree-based model ensembles generally excel in predictive power. If the primary concern is the model's predictive performance, tree-based modeling tends to outperform logistic regression. Tree-based models also require less data cleaning and make fewer assumptions about the underlying distributions of predictor variables, making them more convenient to work with.

Improvements that could be made to this model:

- Introducing new features could enhance the model's predictive capabilities, particularly when domain knowledge is leveraged. In the case of this model, engineered features accounted for over half of the top 10 most-predictive features employed by the model. Reconstructing the model using different combinations of predictor variables can help reduce noise originating from non-predictive features.

Additional features that could help improve the model:

- Having drive-level information for each user, such as drive times and geographic locations, would be beneficial. More detailed data providing insights into user interactions with the app, such as the frequency of reporting or confirming road hazard alerts, would be valuable. Also, knowing the monthly count of unique starting and ending locations provided by each driver could offer further assistance.