Waze Project

Course 6 - The nuts and bolts of machine learning

Your team is close to completing their user churn project. Previously, you completed a project proposal, and used Python to explore and analyze Waze's user data, create data visualizations, and conduct a hypothesis test. Most recently, you built a binomial logistic regression model based on multiple variables.

Leadership appreciates all your hard work. Now, they want your team to build a machine learning model to predict user churn. To get the best results, your team decides to build and test two tree-based models: random forest and XGBoost.

Your work will help leadership make informed business decisions to prevent user churn, improve user retention, and grow Waze's business.

Course 6 End-of-Course Project: Build a machine learning model

In this activity, you will practice using tree-based modeling techniques to predict on a binary target class.

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 activity has three parts:

Part 1: Ethical considerations

- Consider the ethical implications of the request
- Should the objective of the model be adjusted?

Part 2: Feature engineering

- Perform feature selection, extraction, and transformation to prepare the data for modeling

Part 3: Modeling

- Build the models, evaluate them, and advise on next steps

Follow the instructions and answer the questions below to complete the activity. Then, you will complete an Executive Summary using the questions listed on the PACE Strategy Document.

Be sure to complete this activity before moving on. The next course item will provide you with a completed exemplar to compare to your own work.

Build a machine learning model

PACE stages

Throughout these project notebooks, you'll see references to the problem-solving framework PACE. The following notebook components are labeled with the respective PACE stage: Plan, Analyze, Construct, and Execute.

PACE: Plan

Consider the questions in your PACE Strategy Document to reflect on the Plan stage.

In this stage, consider the following questions:

1. What are you being asked to do?
2. What are the ethical implications of the model? What are the consequences of your model making errors?

- What is the likely effect of the model when it predicts a false negative (i.e., when the model says a Waze user won't churn, but they actually will)?
- What is the likely effect of the model when it predicts a false positive (i.e., when the model says a Waze user will churn, but they actually won't)?

3. Do the benefits of such a model outweigh the potential problems?
4. Would you proceed with the request to build this model? Why or why not?

==> ENTER YOUR RESPONSES TO QUESTIONS 1-4 HERE

Task 1. Imports and data loading

Import packages and libraries needed to build and evaluate random forest and XGBoost classification models.

In [2]:
```
# Import packages for data manipulation
### YOUR CODE HERE ###
import numpy as np
import pandas as pd
# Import packages for data visualization
### YOUR CODE HERE ###
import matplotlib.pyplot as plt
# This lets us see all of the columns, preventing Juptyer from redacting them.
### YOUR CODE HERE ###
pd.set_option('display.max_columns', None)
# Import packages for data modeling
### YOUR CODE HERE ###
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, PrecisionRec
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
# This is the function that helps plot feature importance
### YOUR CODE HERE ###
from xgboost import plot_importance

# This module lets us save our models once we fit them.
### YOUR CODE HERE ###
import pickle
```

Now read in the dataset as `df0` and inspect the first five rows.

Note: As shown in this cell, the dataset has been automatically loaded in for you. You do not need to download the .csv file, or provide more code, in order to access the dataset and proceed with this lab. Please continue with this activity by completing the following instructions.

In [3]:
```
# Import dataset
df0 = pd.read_csv('waze_dataset.csv')
```

In [4]:
```
# Inspect the first five rows
### YOUR CODE HERE ###
df0.head()
```

Out[4]:

	ID	label	sessions	drives	total_sessions	n_days_after_onboarding	total_navigations_fav1	tc
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	

PACE: Analyze

Consider the questions in your PACE Strategy Document to reflect on the Analyze stage.

Task 2. Feature engineering

You have already prepared much of this data and performed exploratory data analysis (EDA) in previous courses. You know that some features had stronger correlations with churn than others, and you also created some features that may be useful.

In this part of the project, you'll engineer these features and some new features to use for modeling.

To begin, create a copy of `df0` to preserve the original dataframe. Call the copy `df`.

In [5]:
```
# Copy the df0 dataframe
### YOUR CODE HERE ###
df = df0.copy()
```

Call `info()` on the new dataframe so the existing columns can be easily referenced.

In [6]:
```
### YOUR CODE HERE ###
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

1. Create a feature representing the mean number of kilometers driven on each driving day in the last month for each user. Add this feature as a column to `df`.

In [7]:
```
# 1. Create `km_per_driving_day` feature
### YOUR CODE HERE ###
df['km_per_driving_day'] = df['driven_km_drives'] / df['driving_days']
# 2. Get descriptive stats
### YOUR CODE HERE ###
df['km_per_driving_day'].describe()
```

Out[7]:
```
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
```

Notice that some values are infinite. This is the result of there being values of zero in the `driving_days` column. Pandas imputes a value of infinity in the corresponding rows of the new column because division by zero is undefined.

1. Convert these values from infinity to zero. You can use `np.inf` to refer to a value of infinity.
2. Call `describe()` on the `km_per_driving_day` column to verify that it worked.

In [8]:
```
# 1. Convert infinite values to zero
### YOUR CODE HERE ###
df.loc[df['km_per_driving_day']==np.inf, 'km_per_driving_day'] = 0
# 2. Confirm that it worked
### YOUR CODE HERE ###
df['km_per_driving_day'].describe()
```

Out[8]:
```
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`

1. 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.
2. Get descriptive statistics for this new feature

In [9]:
```
# 1. Create `percent_sessions_in_last_month` feature
### YOUR CODE HERE ###
df['percent_sessions_in_last_month'] = df['sessions'] / df['total_sessions']
# 1. Get descriptive stats
### YOUR CODE HERE ###
df['percent_sessions_in_last_month'].describe()
```

Out[9]:
```
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

Create a new, binary feature called `professional_driver` that is a 1 for users who had 60 or more drives **and** drove on 15+ days in the last month.

Note: The objective is to create a new feature that separates professional drivers from other drivers. In this scenario, domain knowledge and intuition are used to determine these deciding thresholds, but ultimately they are arbitrary.

To create this column, use the `np.where()` (https://numpy.org/doc/stable/reference/generated/numpy.where.html) function. This function accepts as arguments:

1. A condition
2. What to return when the condition is true
3. What to return when the condition is false

```
Example:
x = [1, 2, 3]
x = np.where(x > 2, 100, 0)
x
array([  0,   0, 100])
```

In [10]:
```
# Create `professional_driver` feature
### YOUR CODE HERE ###
df['professional_driver'] = np.where((df['drives'] >= 60) & (df['driving_days'] >
```

total_sessions_per_day

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

In [11]:
```
# Create `total_sessions_per_day` feature
### YOUR CODE HERE ###
df['total_sessions_per_day'] = df['total_sessions'] / df['n_days_after_onboarding
```

As with other features, get descriptive statistics for this new feature.

In [12]:
```
# Get descriptive stats
### YOUR CODE HERE ###
df['total_sessions_per_day'].describe()
```

Out[12]:
```
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

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

In [13]:
```
# Create `km_per_hour` feature
### YOUR CODE HERE ###
df['km_per_hour'] = df['driven_km_drives'] / (df['duration_minutes_drives'] / 60)
df['km_per_hour'].describe()
```

Out[13]:
```
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
```

km_per_drive

Create a column representing the mean number of kilometers per drive made in the last month for each user. Then, print descriptive statistics for the feature.

In [14]:
```
# Create `km_per_drive` feature
### YOUR CODE HERE ###
df['km_per_drive'] = df['driven_km_drives'] / df['drives']
df['km_per_drive'].describe()
```

Out[14]:
```
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
```

This feature has infinite values too. Convert the infinite values to zero, then confirm that it worked.

In [15]:
```
# 1. Convert infinite values to zero
### YOUR CODE HERE ###
df.loc[df['km_per_drive']==np.inf, 'km_per_drive'] = 0
# 2. Confirm that it worked
### YOUR CODE HERE ###
df['km_per_drive'].describe()
```

Out[15]:
```
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

Finally, create a new column that represents the percentage of total sessions that were used to navigate to one of the users' favorite places. Then, print descriptive statistics for the new column.

This is a proxy representation for the percent of overall drives that are to a favorite place. Since total drives since onboarding are not contained in this dataset, total sessions must serve as a reasonable approximation.

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.

In [16]:
```
# Create `percent_of_sessions_to_favorite` feature
### YOUR CODE HERE ###
df['percent_of_drives_to_favorite'] = (
    df['total_navigations_fav1'] + df['total_navigations_fav2']) / df['total_sess
# Get descriptive stats
### YOUR CODE HERE ###
df['percent_of_drives_to_favorite'].describe()
```

Out[16]:
```
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
```

Task 3. Drop missing values

Because you know from previous EDA that there is no evidence of a non-random cause of the 700 missing values in the `label` column, and because these observations comprise less than 5% of the data, use the `dropna()` method to drop the rows that are missing this data.

In [17]:
```
# Drop rows with missing values
### YOUR CODE HERE ###
df = df.dropna(subset=['label'])
```

Task 4. Outliers

You know from previous EDA that many of these columns have outliers. However, tree-based models are resilient to outliers, so there is no need to make any imputations.

Task 5. Variable encoding

Dummying features

In order to use `device` as an X variable, you will need to convert it to binary, since this variable is categorical.

In cases where the data contains many categorical variables, you can use pandas built-in `pd.get_dummies()` (https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html), or you can use scikit-learn's `OneHotEncoder()` (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) function.

Note: Each possible category of each feature will result in a feature for your model, which could lead to an inadequate ratio of features to observations and/or difficulty understanding your model's predictions.

Because this dataset only has one remaining categorical feature (`device`), it's not necessary to use one of these special functions. You can just implement the transformation directly.

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

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

In [18]:
```
# Create new `device2` variable
### YOUR CODE HERE ###
df['device2'] = np.where(df['device']=='Android', 0, 1)
df[['device', 'device2']].tail()
```

Out[18]:

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

Target encoding

The target variable is also categorical, since a user is labeled as either "churned" or "retained." Change the data type of the `label` column to be binary. This change is needed to train the models.

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

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

Save this variable as `label2` so as not to overwrite the original `label` variable.

Note: There are many ways to do this. Consider using `np.where()` as you did earlier in this notebook.

In [19]:
```
# Create binary `label2` column
### YOUR CODE HERE ###
df['label2'] = np.where(df['label']=='churned', 1, 0)
df[['label', 'label2']].tail()
```

Out[19]:

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

Task 6. Feature selection

Tree-based models can handle multicollinearity, so the only feature that can be cut is `ID`, since it doesn't contain any information relevant to churn.

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

Drop `ID` from the `df` dataframe.

In [20]:
```
# Drop `ID` column
### YOUR CODE HERE ###
df = df.drop(['ID'], axis=1)
```

Task 7. Evaluation metric

Before modeling, you must decide on an evaluation metric. This will depend on the class balance of the target variable and the use case of the model.

First, examine the class balance of your target variable.

In [21]:
```
# Get class balance of 'label' col
### YOUR CODE HERE ###
df['label'].value_counts()
```

Out[21]:
```
label
retained    11763
churned      2536
Name: count, dtype: int64
```

Approximately 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.

Now, consider which evaluation metric is best. Remember, 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.

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. Therefore, select the model based on the recall score.

PACE: Construct

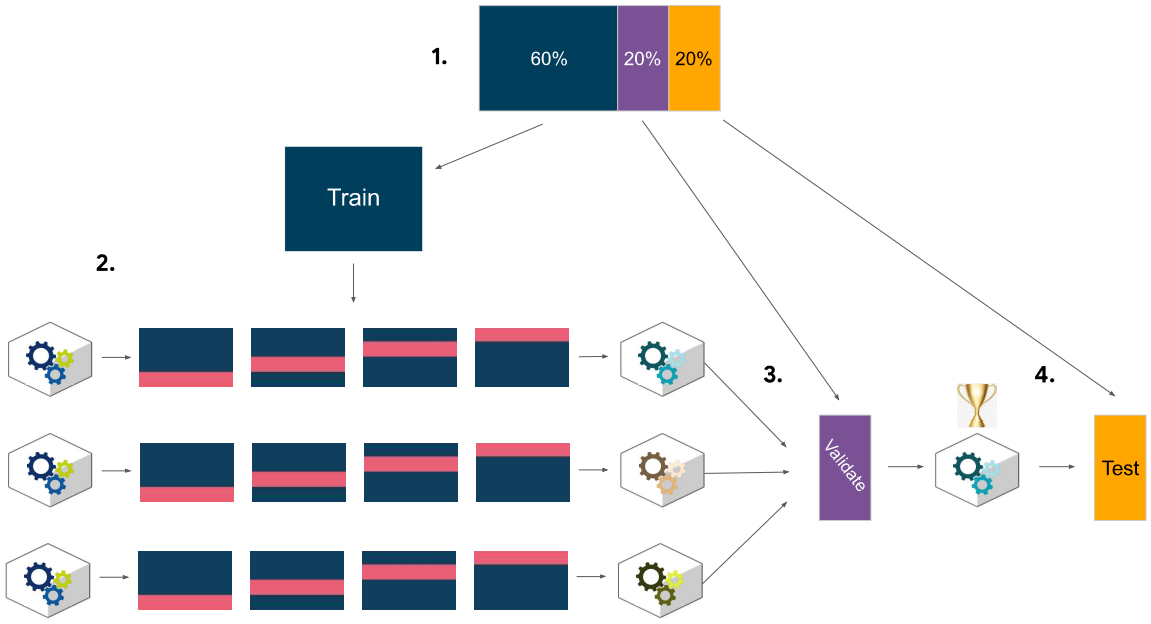
Consider the questions in your PACE Strategy Document to reflect on the Construct stage.

Task 8. 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)

Note that, when deciding the split ratio and whether or not to use a validation set to select a champion model, consider both how many samples will be in each data partition, and how many examples of the minority class each would therefore contain. In this case, a 60/20/20 split would result in ~2,860 samples in the validation set and the same number in the test set, of which ~18% —or 515 samples—would represent users who churn. 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



Task 9. Split the data

Now you're ready to model. The only remaining step is to split the data into features/target variable and training/validation/test sets.

1. Define a variable `X` that isolates the features. Remember not to use `device`.
2. Define a variable `y` that isolates the target variable (`label2`).
3. Split the data 80/20 into an interim training set and a test set. Don't forget to stratify the splits, and set the random state to 42.
4. 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. Again, don't forget to stratify the splits and set the random state.

In [22]:
```
# 1. Isolate X variables
### YOUR CODE HERE ###
X = df.drop(columns=['label', 'label2', 'device'])
# 2. Isolate y variable
### YOUR CODE HERE ###
y = df['label2']
# 3. Split into train and test sets
### YOUR CODE HERE ###
X_tr, X_test, y_tr, y_test = train_test_split(X, y, stratify=y,test_size=0.2, ran
# 4. Split into train and validate sets
### YOUR CODE HERE ###
X_train, X_val, y_train, y_val = train_test_split(X_tr, y_tr, stratify=y_tr,test_
```

Verify the number of samples in the partitioned data.

In [23]:
```
### YOUR CODE HERE ###
for x in [X_train, X_val, X_test]:
    print(len(x))
```

```
8579
2860
2860
```

This aligns with expectations.

Task 10. Modeling

Random forest

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

1. Instantiate the random forest classifier `rf` and set the random state.
2. Create a dictionary `cv_params` of any of the following hyperparameters and their corresponding values to tune. The more you tune, the better your model will fit the data, but the longer it will take.

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

3. Define a dictionary `scoring` of scoring metrics for GridSearch to capture (precision, recall, F1 score, and accuracy).
4. Instantiate the `GridSearchCV` object `rf_cv` . Pass 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'` .

Note: To save time, this exemplar doesn't use multiple values for each parameter in the grid search, but you should include a range of values in your search to home in on the best set of parameters.

In [24]:
```
# 1. Instantiate the random forest classifier
### YOUR CODE HERE ###
rf = RandomForestClassifier(random_state=42)
# 2. Create a dictionary of hyperparameters to tune
### YOUR CODE HERE ###
cv_params = {'max_depth': [None],'max_features': [1.0],'max_samples': [1.0],'min_
             'n_estimators': [300]}
# 3. Define a dictionary of scoring metrics to capture
### YOUR CODE HERE ###
scoring = {'accuracy': 'accuracy', 'precision': 'precision', 'recall': 'recall',
# 4. Instantiate the GridSearchCV object
### YOUR CODE HERE ###
rf_cv = GridSearchCV(rf, cv_params, scoring=scoring, cv=4, refit='recall')
```

Now fit the model to the training data.

In [27]:
```
### YOUR CODE HERE ###
%%time
rf_cv.fit(X_train, y_train)
```

```
UsageError: Line magic function `%%time` not found.
```

Examine the best average score across all the validation folds.

In [28]:
```
# Examine best score
### YOUR CODE HERE ###
rf_cv.best_score_
```

Out[28]:
```
0.12678201409034398
```

Examine the best combination of hyperparameters.

In [29]:
```
# Examine best hyperparameter combo
### YOUR CODE HERE ###
rf_cv.best_params_
```

Out[29]:
```
{'max_depth': None,
 'max_features': 1.0,
 'max_samples': 1.0,
 'min_samples_leaf': 2,
 'min_samples_split': 2,
 'n_estimators': 300}
```

Use the `make_results()` function to output all of the scores of your model. Note that the function accepts three arguments.

HINT

In [36]:
```
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 t
        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 GridSearchC
  ### YOUR CODE HERE ###
    metric_dict = {'precision': 'mean_test_precision','recall': 'mean_test_recall
                   'accuracy': 'mean_test_accuracy'}
  # Get all the results from the CV and put them in a df
  ### YOUR CODE HERE ###
    cv_results = pd.DataFrame(model_object.cv_results_)
  # Isolate the row of the df with the max(metric) score
  ### YOUR CODE HERE ###
    best_estimator_results = cv_results.iloc[cv_results[metric_dict[metric]].idxm
  # Extract Accuracy, precision, recall, and f1 score from that row
  ### YOUR CODE HERE ###
    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
  ### YOUR CODE HERE ###
    table = pd.DataFrame({'model': [model_name],
                          'precision': [precision],
                          'recall': [recall],
                          'F1': [f1],
                          'accuracy': [accuracy],
                          },
                         )

    return table
```

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

In [ ]:
```
### YOUR CODE HERE ###
results = make_results('RF cv', rf_cv)
results
```

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

If you want, feel free to try retuning your hyperparameters to try to get a better score. You might be able to marginally improve the model.

XGBoost

Try to improve your scores using an XGBoost model.

1. Instantiate the XGBoost classifier `xgb` and set `objective='binary:logistic'`. Also set the random state.
2. Create a dictionary `cv_params` of the following hyperparameters and their corresponding values to tune:

- `max_depth`
- `min_child_weight`
- `learning_rate`
- `n_estimators`

3. Define a dictionary `scoring` of scoring metrics for grid search to capture (precision, recall, F1 score, and accuracy).
4. Instantiate the `GridSearchCV` object `xgb_cv`. Pass 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 [39]:
```
# 1. Instantiate the XGBoost classifier
### YOUR CODE HERE ###
xgb = XGBClassifier(objective='binary:logistic', random_state=42)
# 2. Create a dictionary of hyperparameters to tune
### YOUR CODE HERE ###
cv_params = {'max_depth': [6, 12],'min_child_weight': [3, 5],'learning_rate': [0.
# 3. Define a dictionary of scoring metrics to capture
### YOUR CODE HERE ###
scoring = {'accuracy': 'accuracy', 'precision': 'precision', 'recall': 'recall',
# 4. Instantiate the GridSearchCV object
### YOUR CODE HERE ###
xgb_cv = GridSearchCV(xgb, cv_params, scoring=scoring, cv=4, refit='recall')
```

Now fit the model to the `X_train` and `y_train` data.

Note this cell might take several minutes to run.

In [ ]:
```
### YOUR CODE HERE ###
xgb_cv.fit(X_train, y_train)
```

Get the best score from this model.

In [ ]:
```
# Examine best score
### YOUR CODE HERE ###
xgb_cv.best_score_
```

And the best parameters.

In [ ]:
```
# Examine best parameters
### YOUR CODE HERE ###
xgb_cv.best_params_
```

Use the `make_results()` function to output all of the scores of your model. Note that the function accepts three arguments.

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

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, while maintaining a similar accuracy and precision score.

Task 11. Model selection

Now, 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.

Random forest

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

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

In [ ]:
```
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 outp
        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
    '''
    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 [ ]:
```
# Get validation scores for RF model
### YOUR CODE HERE ###
rf_val_scores = get_test_scores('RF val', rf_val_preds, y_val)
# Append to the results table
### YOUR CODE HERE ###
results = pd.concat([results, rf_val_scores], axis=0)
results
```

Notice that 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.

XGBoost

Now, do the same thing to get the performance scores of the XGBoost model on the validation data.

In [ ]:
```
# Use XGBoost model to predict on validation data
### YOUR CODE HERE ###
xgb_val_preds = xgb_cv.best_estimator_.predict(X_val)
# Get validation scores for XGBoost model
### YOUR CODE HERE ###
xgb_val_scores = get_test_scores('XGB val', xgb_val_preds, y_val)
# Append to the results table
### YOUR CODE HERE ###
results = pd.concat([results, xgb_val_scores], axis=0)
results
```

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

PACE: Execute

Consider the questions in your PACE Strategy Document to reflect on the Execute stage.

Task 12. Use champion model to predict on test data

Now, use the champion model to predict on the test dataset. This is to give a final indication of how you should expect the model to perform on new future data, should you decide to use the model.

In [ ]:
```
# Use XGBoost model to predict on test data
### YOUR CODE HERE ###
xgb_test_preds = xgb_cv.best_estimator_.predict(X_test)
# Get test scores for XGBoost model
### YOUR CODE HERE ###
xgb_test_scores = get_test_scores('XGB test', xgb_test_preds, y_test)
# Append to the results table
### YOUR CODE HERE ###
results = pd.concat([results, xgb_test_scores], axis=0)
results
```

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.

Task 13. Confusion matrix

Plot a confusion matrix of the champion model's predictions on the test data.

In [ ]:
```
# Generate array of values for confusion matrix
### YOUR CODE HERE ###
cm = confusion_matrix(y_test, xgb_test_preds, labels=xgb_cv.classes_)
# Plot confusion matrix
### YOUR CODE HERE ###
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=['retained', 'churned'])
disp.plot();
```

The model predicted three times as many false negatives than it did false positives, and it correctly identified only 16.6% of the users who actually churned.

Task 14. Feature importance

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

In [ ]:
```
### YOUR CODE HERE ###
plot_importance(xgb_cv.best_estimator_);
```

The XGBoost model made more use of many of the features than did the logistic regression model from the previous course, which weighted a single feature (`activity_days`) very heavily in its final prediction.

If anything, this underscores the importance of feature engineering. Notice that engineered features accounted for six of the top 10 features (and three of the top five). Feature engineering is often one of the best and easiest ways to boost model performance.

Also, note that the important features in one model might not be the same as the important features in another model. That's why you shouldn't discount features as unimportant without thoroughly examining them and understanding their relationship with the dependent variable, if possible. These discrepancies between features selected by models are typically caused by complex feature interactions.

Remember, sometimes your data simply will not be predictive of your chosen target. This is common. Machine learning is a powerful tool, but it is not magic. If your data does not contain predictive signal, even the most complex algorithm will not be able to deliver consistent and accurate predictions. Do not be afraid to draw this conclusion.

Even if you cannot use the model to make strong predictions, was the work done in vain? What insights can you report back to stakeholders?

Task 15. Conclusion

Now that you've built and tested your machine learning models, the next step is to share your findings with the Waze leadership team. Consider the following questions as you prepare to write your executive summary. Think about key points you may want to share with the team, and what information is most relevant to the user churn project.

Questions:

1. Would you recommend using this model for churn prediction? Why or why not?
2. What tradeoff was made by splitting the data into training, validation, and test sets as opposed to just training and test sets?
3. What is the benefit of using a logistic regression model over an ensemble of tree-based models (like random forest or XGBoost) for classification tasks?
4. What is the benefit of using an ensemble of tree-based models like random forest or XGBoost over a logistic regression model for classification tasks?
5. What could you do to improve this model?
6. What additional features would you like to have to help improve the model?

==> ENTER YOUR RESPONSES TO QUESTIONS 1-6 HERE

1. No, I suppose as the model has not done that well.
2. Splitting meant that there was data availabe to train the model.
3. Logistic Regression models are easier to understand and interpret.
4. Tree based model or Xboosting are better predictors though.
5. Probably new features can be used and engineered.
6. More indepth data could be made availabe what users do on the app.

Congratulations! You've completed this lab. However, you may not notice a green check mark next to this item on Coursera's platform. Please continue your progress regardless of the check mark. Just click on the "save" icon at the top of this notebook to ensure your work has been logged.