Decision tree

In [3]:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# This function displays the splits of the tree
from sklearn.tree import plot_tree

from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
from sklearn.metrics import recall_score, precision_score, f1_score, accuracy_scor
```

Read in the data

In [7]:

```python
# Read in data
file = 'archive (4).zip'
df_original = pd.read_csv(file)
df_original.head()
```

Out[7]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfPr
0	1	15634602	Hargrave	619	France	Female	42	2	0.00	
1	2	15647311	Hill	608	Spain	Female	41	1	83807.86	
2	3	15619304	Onio	502	France	Female	42	8	159660.80	
3	4	15701354	Boni	699	France	Female	39	1	0.00	
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	

Examine the data

At this point in a typical data science project, you'd do a thorough exploratory data analysis (EDA) to better understand your data and what it's telling you. For the purposes of this lab, we will skip this.

In the steps that follow, we'll perform many of the same steps that we took to build our Naive Bayes model. We'll redo them here to review them for good measure.

We'll begin by checking the balance of the classes in our target variable (Exited), as this will help us determine how to prepare our data. It will also inform our decision of what evaluation metric to use to select our final model.

Check class balance

In [8]:

```
df_original['Exited'].value_counts()
```

Out[8]:

```
0    7963
1    2037
Name: Exited, dtype: int64
```

The class of our target variable is split roughly 80/20. In other words, ~20% of the people in this dataset churned. This is an unbalanced dataset, but it's not extreme. We will preserve this ratio when we model.

Select an evaluation metric

The data contains 10,000 observations, and the class distribution is approximately 80/20.

Since we have some imbalance in our target classes, we know that if we measure model performance by accuracy alone, the model could predict 0 (no churn) 100% of the time and have an accuracy of ~80%. An accuracy of 80% might seem pretty good, but we know in this case it would be meaningless, because our model would fail to identify anybody who churned. Therefore, accuracy is not the best metric to use to evaluate our model's performance.

To determine which evaluation metric might be best, consider how our model might be wrong. There are two possibilities for bad predictions:

False positives: When the model predicts a customer will churn when in fact they won't False negatives: When the model predicts a customer will not churn when in fact they will As you know, there are a number of performance metrics aside from accuracy to choose from. Some of these include precision, recall, and F1 score. Let's examine these more closely, beginning with *precision*:

$precision$=TPFP+TP And *recall*:

$recall$=TPFN+TP

Refer to the confusion matrix for a reminder on what the terms represent.

Precision represents the percentage of all our model's predicted positives that are true positives. This might not be the best metric for us to use, because it disincentivizes predicting someone will churn unless there is a high degree of certainty that they will. This could translate to a high rate of false negatives.

On the other hand, recall represents the percentage of all actual positives that the model identifies as such. This also might not be the best metric to use, because it rewards predicting someone will churn even if the likelihood of their doing so is very small. This could translate to a high rate of false positives.

So which is worse, false positives or false negatives? Well, we'd first have to define what worse means. This is dependent on the details of the project that you're working on. For the sake of this exercise, let us suppose that we're defining it as the error that would cost the bank more money.

We can quickly get an idea of how much money each customer who churns costs the bank by calculating the average balance of all customers who churned.

In [10]:

```
# Calculate average balance of customers who churned
avg_churned_bal = df_original[df_original['Exited']==1]['Balance'].mean()
avg_churned_bal
```

Out[10]:

```
91108.53933726063
```

This shows that the customers who churned each took with them €91,108.54, on average. That's a lot of money! This represents the average cost of the model predicting a false negative.

What's the cost of predicting a false positive? Well, it's the cost of whatever the incentive might be to convince someone to stay with the bank when they were going to stay regardless of whether or not they were incentived. We don't have a number for this, and even if it's probably less than €91,108.54, it still could be thousands of Euros per customer in lost revenue, depending on the details of the incentive.

Since correctly identifying customers who will churn is potentially very valuable, we could select recall as our most important metric. This might be a perfectly valid approach, depending on the specifics of the campaign. But this could also be problematic. After all, if we select a model based solely on recall, we could select a very biased model that predicts everyone to churn, but then 8,000 people would be given incentives needlessly.

Since we don't know the exact cost of predicting a false negative, we'll make an assumption for this exercise. We'll assume that a metric that balances precision and recall is best. The metric that helps us achieve this balance is *F1 score*, which is defined as the harmonic mean of precision and recall.

$F1=2\cdot precision\cdot recallprecision+recall$

Again, there are many metrics to choose from. The important thing is that you make an informed decision that is based on your use case.

Now that we've decided on an evaluation metric, let's prepare the data for modeling.

Feature engineering

Feature selection

In this step, we'll prepare the data for modeling. These are the same steps as what we did for the Naive Bayes model. For more thorough explanation of this process, refer to the Feature Engineering notebook from Module 3. Note that for time considerations, we won't create any new features.

We begin by dropping the columns that we wouldn't expect to offer any predictive signal to the model. These columns include RowNumber, CustomerID, and Surname. We'll drop these columns so they don't introduce noise to our model.

We'll also drop the Gender column, because we don't want our model to make predictions based on gender.

In [12]:

```
# Create a new df that drops RowNumber, CustomerId, Surname, and Gender cols
churn_df = df_original.drop(['RowNumber', 'CustomerId', 'Surname', 'Gender'],
                            axis=1)
```

In [13]:
```
churn_df.head()
```

Out[13]:

	CreditScore	Geography	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	Estimate
0	619	France	42	2	0.00	1	1	1	10
1	608	Spain	41	1	83807.86	1	0	1	11
2	502	France	42	8	159660.80	3	1	0	11
3	699	France	39	1	0.00	2	0	0	9
4	850	Spain	43	2	125510.82	1	1	1	7

Feature transformation

Next, we'll dummy encode the Geography variable, which is categorical. There are three possible categories captured here: France, Spain, and Germany. When we call pd.get_dummies() on this feature, it will replace the Geography column with three new Boolean columns--one for each possible category contained in the column being dummied.

When we specify drop_first='True' in the function call, it means that instead of replacing Geography with three new columns, it will instead replace it with two columns. We can do this because no information is lost from this, but the dataset is shorter and simpler.

In this case, we end up with two new columns called Geography_Germany and Geography_Spain. We don't need a Geography_France column. Why not? Because if a customer's values in Geography_Germany and Geography_Spain are both 0, we'll know they're from France!

In [14]:
```
# Dummy encode categorical variables
churn_df = pd.get_dummies(churn_df, drop_first=True)
```

In [15]:
```
churn_df.head()
```

Out[15]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exi
0	619	42	2	0.00	1	1	1	101348.88	
1	608	41	1	83807.86	1	0	1	112542.58	
2	502	42	8	159660.80	3	1	0	113931.57	
3	699	39	1	0.00	2	0	0	93826.63	
4	850	43	2	125510.82	1	1	1	79084.10	

#Split the data It's time to split the data into features and target variable, and into training data and test data. We do this using the train_test_split() function. We'll put 25% of the data into our test set, and use the remaining 75% to train the model.

Don't forget to include the stratify=y parameter, as this is what ensures that the 80/20 class ratio of the target variable is maintained in both the training and test datasets after splitting.

Lastly, we set a random seed so we and others can reproduce our work.

In [16]:

```
# Define the y (target) variable
y = churn_df['Exited']

# Define the X (predictor) variables
X = churn_df.copy()
X = X.drop('Exited', axis=1)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.25, stratify=y,
                                                    random_state=42)
```

Baseline model We'll first train a baseline model, just to get a sense of how predictive the data is and to give us scores that we can reference later. This will also show the process of instantiating and fitting the model, and then using it to make predictions. We'll predict on the test data. Note: The following operation may take over 30 minutes to complete

In [17]:

```
# Instantiate the model
decision_tree = DecisionTreeClassifier(random_state=0)

# Fit the model to training data
decision_tree.fit(X_train, y_train)

# Make predictions on test data
dt_pred = decision_tree.predict(X_test)
```

In [18]:

```
# Generate performance metrics
print("Accuracy:", "%.3f" % accuracy_score(y_test, dt_pred))
print("Precision:", "%.3f" % precision_score(y_test, dt_pred))
print("Recall:", "%.3f" % recall_score(y_test, dt_pred))
print("F1 Score:", "%.3f" % f1_score(y_test, dt_pred))
```

```
Accuracy: 0.790
Precision: 0.486
Recall: 0.503
F1 Score: 0.494
```

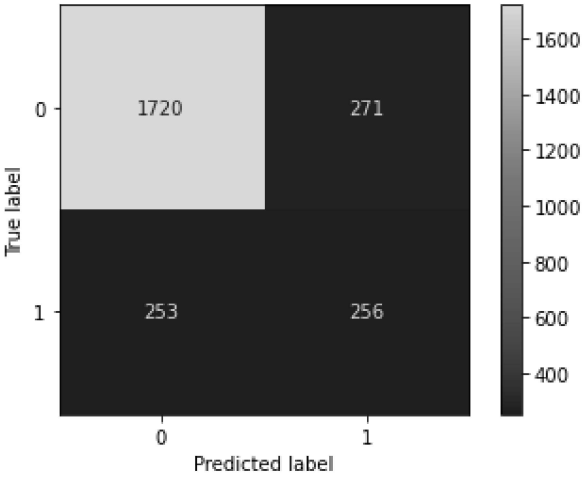
Analysis of baseline model Confusion matrix Let's inspect the confusion matrix of our decision tree's predictions. First, we'll write a short helper function to help us display the matrix.

In [19]:

```
#Confusion matrix
#Let's inspect the confusion matrix of our decision tree's predictions. First, we'

def conf_matrix_plot(model, x_data, y_data):
    '''
    Accepts as argument model object, X data (test or validate), and y data (test
    Returns a plot of confusion matrix for predictions on y data.
    '''

    model_pred = model.predict(x_data)
    cm = confusion_matrix(y_data, model_pred, labels=model.classes_)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                                  display_labels=model.classes_)

    disp.plot(values_format='')
    plt.show()
```

In [20]:

```
# Generate confusion matrix
conf_matrix_plot(decision_tree, X_test, y_test)
```



Notice from this confusion matrix that the model correctly predicts many true negatives. Of course, this is to be expected, given that the dataset is imbalanced in favor of negatives. When the model makes an error, it appears slightly more likely to predict a false positive than a false negative, but it's generally balanced. This is reflected in the precision and recall scores both being very close to each other.

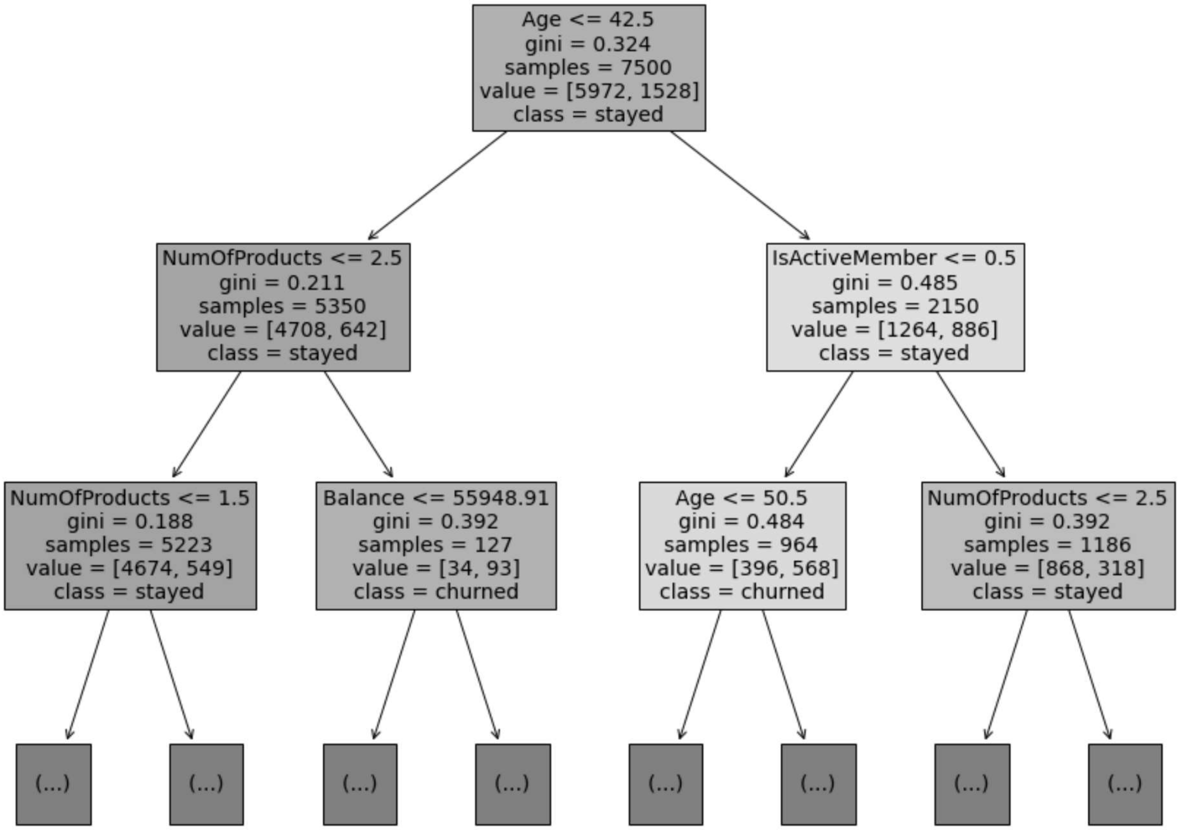
Plotting the tree Next, let's examine the splits of the tree. We'll do this by using the plot_tree() function that we imported. We pass to it our fit model as well as some additional parameters. Note that if we did not set max_depth=2, the function would return a plot of the entire tree, all the way down to the leaf nodes. This is intractable and unnecessary. We're most interested in the splits nearest the root, because these tell us the most predictive features.

class_names displays what the majority class of each node is, and filled colors the nodes according to their majority class.

Note that this plot represents how the tree grew from the training data. To make its predictions on the test data, the tree would simply pass each customer in the test data through its splits, from the root node all the way down to a leaf node.

In [21]:

```
# Plot the tree
plt.figure(figsize=(15,12))
plot_tree(decision_tree, max_depth=2, fontsize=14, feature_names=X.columns,
          class_names={0:'stayed', 1:'churned'}, filled=True);
plt.show()
```



In [22]:

```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV
```

Cross-validated hyperparameter tuning

Cross-validating a model using GridSearchCV can be done in a number of different ways. If you find notebooks online that other people have written, you'll likely soon discover this for yourself. But all variations must fulfill the same general requirements. (Refer to the GridSearchCV documentation for further reading.)

The format presented below is step-wise, making it easier to follow.

Create a dictionary of hyperparameters to search over:

key = name of hyperparameter (string) value = values to search over (list)

In [24]:

```
# Assign a dictionary of hyperparameters to search over
tree_para = {'max_depth':[4,5,6,7,8,9,10,11,12,15,20,30,40,50],
             'min_samples_leaf': [2, 5, 10, 20, 50]}
```

Create a dictionary of scoring metrics to capture. These metrics can be selected from scikit-learn's built-in options or custom-defined. For this exercise, we'll capture accuracy, precision, recall, and F1 score so we can examine all of them. The metrics are entered as strings.

In [25]:

```
# Assign a dictionary of scoring metrics to capture
scoring = {'accuracy', 'precision', 'recall', 'f1'}
```

In [26]:

```
# Instantiate the classifier
tuned_decision_tree = DecisionTreeClassifier(random_state = 42)
```

Instantiate the GridSearchCV object. Pass as arguments:

The classifier (tuned_decision_tree)

The dictionary of hyperparameters to search over (tree_para)

The dictionary of scoring metrics (scoring)

The number of cross-validation folds you want (cv=5)

The scoring metric that you want GridSearch to use when it selects the "best" model (i.e., the model that performs best on average over all validation folds) (refit='f1'*)

- The reason it's called refit is because once the algorithm finds the combination of hyperparameters that results in the best average score across all validation folds, it will then refit this model to all of the training data. Remember, up until now, with a 5-fold cross-validation, the model has only ever been fit on 80% (4/5) of the training data, because the remaining 20% was held out as a validation fold.

Fit the data (X_train, y_train) to the GridSearchCV object (clf)

Depending on the number of different hyperparameters you choose, the number of combinations you search over, the size of your data, and your available computing resources, this could take a long time.

In [27]:

```
# Instantiate the GridSearch
clf = GridSearchCV(tuned_decision_tree,
                   tree_para,
                   scoring = scoring,
                   cv=5,
                   refit="f1")

# Fit the model
clf.fit(X_train, y_train)
```

Out[27]:

```
GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=42),
             param_grid={'max_depth': [4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 20, 30,
                                       40, 50],
                         'min_samples_leaf': [2, 5, 10, 20, 50]},
             refit='f1', scoring={'accuracy', 'f1', 'precision', 'recall'})
```

Now that the model is fit and cross-validated, we can use the best_estimator_ attribute to inspect the hyperparameter values that yielded the highest F1 score during cross-validation.

In [28]:

```
# Examine the best model from GridSearch
clf.best_estimator_
```

Out[28]:

```
DecisionTreeClassifier(max_depth=8, min_samples_leaf=20, random_state=42)
```

The best_score_ attribute returns the best average F1 score across the different folds among all the combinations of hyperparameters. Note that if we had set refit='recall' when we instantiated our GridSearchCV object earlier, then calling best_score_ would return the best recall score, and the best parameters might not be the same as what they are in the above cell, because the model would be selected based on a different metric.

In [29]:

```
print("Best Avg. Validation Score: ", "%.4f" % clf.best_score_)
```

```
Best Avg. Validation Score:  0.5607
```

Although the F1 score of 0.561 is significantly better than the baseline model's F1 score of 0.494, it's not a fair comparison, because the baseline model was scored on the test data and the tuned model was scored against validation folds that came from the training data.

Recall that when we ran our grid search, we specified that we also wanted to capture precision, recall, and accuracy. The reason for doing this is that it's difficult to interpret an F1 score. These other metrics are much more directly interpretable, so they're worth knowing.

The following cell defines a helper function that extracts these scores from the fit GridSearchCV object and returns a pandas dataframe with all four scores from the model with the best average F1 score during validation. This function will help us later when we want to add the results of other models to the table.

In [30]:

```python
def make_results(model_name, model_object):
    '''
    Accepts as arguments a model name (your choice - string) and
    a fit GridSearchCV model object.

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

    # 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(mean f1 score)
    best_estimator_results = cv_results.iloc[cv_results['mean_test_f1'].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()
    table = table.append({'Model': model_name,
                          'F1': f1,
                          'Recall': recall,
                          'Precision': precision,
                          'Accuracy': accuracy
                          },
                          ignore_index=True
                         )

    return table
```

In [33]:

```python
# Call the function on our model
result_table = make_results("Tuned Decision Tree", clf)
```

```
C:\Users\Blessing\AppData\Local\Temp\ipykernel_8508\2409802599.py:24: FutureWarning:
The frame.append method is deprecated and will be removed from pandas in a future ver
sion. Use pandas.concat instead.
  table = table.append({'Model': model_name,
```

We can save these results and open them in another notebook if we want to add to them. We'll save as a .csv file using to_csv().

In [34]:

```python
# Save results table as csv
result_table.to_csv("Results.csv")
```

In [35]:

```python
# View the results
result_table
```

Out[35]:

	Model	F1	Recall	Precision	Accuracy
0	Tuned Decision Tree	0.560655	0.469255	0.701608	0.8504

These results show that our model's performance isn't great, but it's not terrible either. Maybe another kind of model will do better...

In [ ]: