Predicting_Engagement

In [1]:

```
%matplotlib inline
```

In [2]:

```
import matplotlib.pyplot as plt
import pandas as pd
```

1. Load Data

In [3]:

```
df = pd.read_csv('Data/WA_Fn-UseC_-Marketing-Customer-Value-Analysis.csv')
```

In [4]:

```
df.shape
```

Out[4]:

```
(9134, 24)
```

In [5]:

```
df.head()
```

Out[5]:

	Customer	State	Customer Lifetime Value	Response	Coverage	Education	Effective To Date	Employm
0	BU79786	Washington	2763.519279	No	Basic	Bachelor	2/24/11	
1	QZ44356	Arizona	6979.535903	No	Extended	Bachelor	1/31/11	Ur
2	AI49188	Nevada	12887.431650	No	Premium	Bachelor	2/19/11	
3	WW63253	California	7645.861827	No	Basic	Bachelor	1/20/11	Ur
4	HB64268	Washington	2813.692575	No	Basic	Bachelor	2/3/11	

5 rows × 24 columns

In [6]:

```
df.select_dtypes(include=['category', object]).columns
```

Out[6]:

```
Index(['Customer', 'State', 'Response', 'Coverage', 'Education',
       'Effective To Date', 'EmploymentStatus', 'Gender', 'Location Code',
       'Marital Status', 'Policy Type', 'Policy', 'Renew Offer Type',
       'Sales Channel', 'Vehicle Class', 'Vehicle Size'],
      dtype='object')
```

As we can see from the previous output, we have numerous columns that contain categorical variables.
In order to build machine learning models using the *scikit-learn* package, all features in the dataset need to have numerical values.
So we need to encode the categorical variables into numbers, starting from the Response target variable and then passing across all the others we want to consider in our predictive model.

2. Variable Encoding

2.1. Response Variable: Response

In [7]:

```
df['Engaged'] = df['Response'].apply(lambda x: 1 if x == 'Yes' else 0)
```

In [8]:

```
# Get the overall response or engagement rate

df['Engaged'].mean()
```

Out[8]:

```
0.14320122618786948
```

2.2. Features: other categorical variables to be encoded

In [9]:

```
df.describe()
```

Out[9]:

	Customer Lifetime Value	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Nu
count	9134.000000	9134.000000	9134.000000	9134.000000	9134.000000	9134.000000	913
mean	8004.940475	37657.380009	93.219291	15.097000	48.064594	0.384388	
std	6870.967608	30379.904734	34.407967	10.073257	27.905991	0.910384	
min	1898.007675	0.000000	61.000000	0.000000	0.000000	0.000000	
25%	3994.251794	0.000000	68.000000	6.000000	24.000000	0.000000	
50%	5780.182197	33889.500000	83.000000	14.000000	48.000000	0.000000	
75%	8962.167041	62320.000000	109.000000	23.000000	71.000000	0.000000	
max	83325.381190	99981.000000	298.000000	35.000000	99.000000	5.000000	

In [10]:

```
continuous_features = [
    'Customer Lifetime Value', 'Income', 'Monthly Premium Auto',
    'Months Since Last Claim', 'Months Since Policy Inception',
    'Number of Open Complaints', 'Number of Policies', 'Total Claim Amount'
]
```

- Creating Dummy Variables

In [11]:

```
columns_to_encode = [
    'Sales Channel', 'Vehicle Size', 'Vehicle Class', 'Policy', 'Policy Type',
    'EmploymentStatus', 'Marital Status', 'Education', 'Coverage'
]

# To encode these variables we are going to create dummy variables for each category
# of the individual categorical variables

categorical_features = []
for col in columns_to_encode:
    encoded_df = pd.get_dummies(df[col])
    encoded_df.columns = [col.replace(' ', '.') + '.' + x for x in encoded_df.columns]

    categorical_features += list(encoded_df.columns)

    df = pd.concat([df, encoded_df], axis=1)
```

In [12]:

```
df.shape
```

Out[12]:

```
(9134, 66)
```

In [13]:

```
df.columns
```

Out[13]:

```
Index(['Customer', 'State', 'Customer Lifetime Value', 'Response', 'Covera
ge',
       'Education', 'Effective To Date', 'EmploymentStatus', 'Gender',
       'Income', 'Location Code', 'Marital Status', 'Monthly Premium Aut
o',
       'Months Since Last Claim', 'Months Since Policy Inception',
       'Number of Open Complaints', 'Number of Policies', 'Policy Type',
       'Policy', 'Renew Offer Type', 'Sales Channel', 'Total Claim Amoun
t',
       'Vehicle Class', 'Vehicle Size', 'Engaged', 'Sales.Channel.Agent',
       'Sales.Channel.Branch', 'Sales.Channel.Call Center',
       'Sales.Channel.Web', 'Vehicle.Size.Large', 'Vehicle.Size.Medsize',
       'Vehicle.Size.Small', 'Vehicle.Class.Four-Door Car',
       'Vehicle.Class.Luxury Car', 'Vehicle.Class.Luxury SUV',
       'Vehicle.Class.SUV', 'Vehicle.Class.Sports Car',
       'Vehicle.Class.Two-Door Car', 'Policy.Corporate L1',
       'Policy.Corporate L2', 'Policy.Corporate L3', 'Policy.Personal L1',
       'Policy.Personal L2', 'Policy.Personal L3', 'Policy.Special L1',
       'Policy.Special L2', 'Policy.Special L3', 'Policy.Type.Corporate Au
to',
       'Policy.Type.Personal Auto', 'Policy.Type.Special Auto',
       'EmploymentStatus.Disabled', 'EmploymentStatus.Employed',
       'EmploymentStatus.Medical Leave', 'EmploymentStatus.Retired',
       'EmploymentStatus.Unemployed', 'Marital.Status.Divorced',
       'Marital.Status.Married', 'Marital.Status.Single', 'Education.Bache
lor',
       'Education.College', 'Education.Doctor',
       'Education.High School or Below', 'Education.Master', 'Coverage.Bas
ic',
       'Coverage.Extended', 'Coverage.Premium'],
      dtype='object')
```

In [14]:

```
# See as an example what has happened to the column "Sales Channel"

df.filter(items=['Sales.Channel.Agent', 'Sales.Channel.Branch',
                 'Sales.Channel.Call Center', 'Sales.Channel.Web']).head(10)
```

Out[14]:

	Sales.Channel.Agent	Sales.Channel.Branch	Sales.Channel.Call Center	Sales.Channel.Web
0	1	0	0	0
1	1	0	0	0
2	1	0	0	0
3	0	0	1	0
4	1	0	0	0
5	0	0	0	1
6	1	0	0	0
7	1	0	0	0
8	1	0	0	0
9	0	1	0	0

As we can see, now we have four columns for each category contained in the original column and the values are one-hot encoded, meaning that value 1 is attributed to a given record which belong to the given category, and 0 otherwise.

- Encoding Gender

In [15]:

```
# Create a new column called Is.Female and encode its values into 1 for F
# and 0 for M

df['Is.Female'] = df['Gender'].apply(lambda x: 1 if x == 'F' else 0)
categorical_features.append('Is.Female')
```

In [16]:

```
df.head()
```

Out[16]:

	Customer	State	Customer Lifetime Value	Response	Coverage	Education	Effective To Date	Employm
0	BU79786	Washington	2763.519279	No	Basic	Bachelor	2/24/11	
1	QZ44356	Arizona	6979.535903	No	Extended	Bachelor	1/31/11	Ur
2	AI49188	Nevada	12887.431650	No	Premium	Bachelor	2/19/11	
3	WW63253	California	7645.861827	No	Basic	Bachelor	1/20/11	Ur
4	HB64268	Washington	2813.692575	No	Basic	Bachelor	2/3/11	

5 rows × 67 columns

- all features & response

In [17]:

```
all_features = continuous_features + categorical_features
response = 'Engaged'
```

In [18]:

```
# Create a new DataFrame sample_df including all the features and the target variable,
 Response
# Replace all the spaces in the column and feature names with dots

sample_df = df[all_features + [response]]
sample_df.columns = [x.replace(' ', '.') for x in sample_df.columns]
all_features = [x.replace(' ', '.') for x in all_features]
```

In [19]:

```
sample_df.head()
```

Out[19]:

	Customer.Lifetime.Value	Income	Monthly.Premium.Auto	Months.Since.Last.Claim	Months.S
0	2763.519279	56274	69	32	
1	6979.535903	0	94	13	
2	12887.431650	48767	108	18	
3	7645.861827	0	106	18	
4	2813.692575	43836	73	12	

5 rows × 51 columns

3. Training & Testing

We are going to train and test machine learning models to predict the customer responses or engagements.

In [20]:

```
from sklearn.model_selection import train_test_split
```

In [21]:

```
x_train, x_test, y_train, y_test = train_test_split(sample_df[all_features], sample_df[
response], test_size=0.3)
```

In [22]:

```
sample_df.shape
```

Out[22]:

```
(9134, 51)
```

In [23]:

```
x_train.shape
```

Out[23]:

```
(6393, 50)
```

In [24]:

```
x_test.shape
```

Out[24]:

```
(2741, 50)
```

In [25]:

```
6393/9134*100.0
```

Out[25]:

```
69.99124151521787
```

In [26]:

```
2741/9134*100.0
```

Out[26]:

```
30.008758484782135
```

As we can see, roughly 70% of the sample set went into the train set and the remaining 30% of the sample set went into the test set.
We'll be using these train and test sets for building and evaluating models.

3.1. Building RandomForest Model

In [27]:

```
from sklearn.ensemble import RandomForestClassifier
```

In [28]:

```
# Ask the model to build 200 trees, and that each tree can only grow up
# to the depth of 5

rf_model = RandomForestClassifier(
    n_estimators=200,
    max_depth=5
)
```

In [29]:

```
rf_model.fit(X=x_train, y=y_train)
```

Out[29]:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gin
i',
            max_depth=5, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [30]:

```
# Retrieve information about individual trees

rf_model.estimators_[0:5]
```

Out[30]:

```
[DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
             max_features='auto', max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, presort=False,
             random_state=1674768650, splitter='best'),
 DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
             max_features='auto', max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, presort=False,
             random_state=2038793267, splitter='best'),
 DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
             max_features='auto', max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, presort=False,
             random_state=990384462, splitter='best'),
 DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
             max_features='auto', max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, presort=False,
             random_state=949305700, splitter='best'),
 DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
             max_features='auto', max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, presort=False,
             random_state=399325449, splitter='best')]
```

This output shows a list of 5 sub-estimators, which are decision trees.
We can use this information to simulate what each of these estimators predicts for each input.

In [31]:

```
# Get the predictions from the first sub-estimator in the forest

rf_model.estimators_[0].predict(x_test)[:10]
```

Out[31]:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [32]:

```
# Get the predictions from the second sub-estimator in the forest

rf_model.estimators_[1].predict(x_test)[:10]
```

Out[32]:

```
array([0., 0., 1., 0., 0., 0., 0., 0., 0., 0.])
```

In [33]:

```
# Get the predictions from the third sub-estimator in the forest

rf_model.estimators_[2].predict(x_test)[:10]
```

Out[33]:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [34]:

```
# Get the predictions from the fourth sub-estimator in the forest

rf_model.estimators_[3].predict(x_test)[:10]
```

Out[34]:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [35]:

```
# Get the predictions from the fifth sub-estimator in the forest

rf_model.estimators_[4].predict(x_test)[:10]
```

Out[35]:

```
array([1., 0., 0., 0., 0., 0., 1., 0., 0., 1.])
```

By seeing at the output of the first five estimators, we notice that different trees predict differently for each record of the test set because the subsets of features are randomly selected for each trained tree.
As an example of interpretation, we examine the first tree and we can see that it predicts the 6th record to be a class of 1 and the rest to be a class of 0.
Using this information, we can know how the final predictions from the random forest model are formed from the individual sub-estimators or trees.

- Feature Importances

The feature importance represents the importance or the impact of each feature on the final predictions.

In [36]:

```
rf_model.feature_importances_
```

Out[36]:

```
array([6.70527109e-02, 9.05619512e-02, 5.50538831e-02, 3.17087459e-02,
       5.29058655e-02, 1.13850984e-02, 2.08433524e-02, 8.09341173e-02,
       4.45954007e-02, 6.66396018e-03, 6.42941822e-03, 3.81041246e-03,
       9.83170215e-03, 6.64090806e-03, 8.45553781e-03, 4.51430703e-03,
       2.04028518e-04, 2.23051109e-03, 5.24103680e-03, 3.72679552e-03,
       4.95644972e-03, 5.15951265e-04, 9.24648410e-04, 7.54377935e-04,
       1.62556885e-03, 1.69734563e-03, 2.11656688e-03, 9.70613267e-04,
       5.18034142e-04, 3.80700826e-03, 1.25620637e-03, 9.09872949e-04,
       2.06394593e-03, 4.49653903e-03, 2.86649956e-02, 7.99349320e-03,
       2.65164156e-01, 2.94992372e-02, 4.55038881e-02, 2.04580626e-02,
       1.93087132e-02, 3.56356643e-03, 3.60516219e-03, 3.94842189e-03,
       3.36600763e-03, 7.21292492e-03, 7.05373088e-03, 5.74221820e-03,
       1.79666156e-03, 7.71588895e-03])
```

In [37]:

```
feature_importance_df = pd.DataFrame(list(zip(rf_model.feature_importances_,
                                               all_features)))
feature_importance_df.columns = ['feature.importance', 'feature']

feature_importance_df.sort_values(by='feature.importance', ascending=False)
```

Out[37]:

	feature.importance	feature
36	0.265164	EmploymentStatus.Retired
1	0.090562	Income
7	0.080934	Total.Claim.Amount
0	0.067053	Customer.Lifetime.Value
2	0.055054	Monthly.Premium.Auto
4	0.052906	Months.Since.Policy.Inception
38	0.045504	Marital.Status.Divorced
8	0.044595	Sales.Channel.Agent
3	0.031709	Months.Since.Last.Claim
37	0.029499	EmploymentStatus.Unemployed
34	0.028665	EmploymentStatus.Employed
6	0.020843	Number.of.Policies
39	0.020458	Marital.Status.Married
40	0.019309	Marital.Status.Single
5	0.011385	Number.of.Open.Complaints
12	0.009832	Vehicle.Size.Large
14	0.008456	Vehicle.Size.Small
35	0.007993	EmploymentStatus.Medical.Leave
49	0.007716	Is.Female
45	0.007213	Education.Master
46	0.007054	Coverage.Basic
9	0.006664	Sales.Channel.Branch
13	0.006641	Vehicle.Size.Medsize
10	0.006429	Sales.Channel.Call.Center
47	0.005742	Coverage.Extended
18	0.005241	Vehicle.Class.SUV
20	0.004956	Vehicle.Class.Two-Door.Car
15	0.004514	Vehicle.Class.Four-Door.Car
33	0.004497	EmploymentStatus.Disabled
43	0.003948	Education.Doctor
11	0.003810	Sales.Channel.Web
29	0.003807	Policy.Special.L3
19	0.003727	Vehicle.Class.Sports.Car
42	0.003605	Education.College
41	0.003564	Education.Bachelor
44	0.003366	Education.High.School.or.Below
17	0.002231	Vehicle.Class.Luxury.SUV

	feature.importance	feature
26	0.002117	Policy.Personal.L3
32	0.002064	Policy.Type.Special.Auto
48	0.001797	Coverage.Premium
25	0.001697	Policy.Personal.L2
24	0.001626	Policy.Personal.L1
30	0.001256	Policy.Type.Corporate.Auto
27	0.000971	Policy.Special.L1
22	0.000925	Policy.Corporate.L2
31	0.000910	Policy.Type.Personal.Auto
23	0.000754	Policy.Corporate.L3
28	0.000518	Policy.Special.L2
21	0.000516	Policy.Corporate.L1
16	0.000204	Vehicle.Class.Luxury.Car

As we can notice, the *EmploymentStatus.Retired* feature seems to be the most important factor in making the final prediction.

3.2. Evaluating Models

- accuracy, precision, and recall

In [38]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score
```

In [39]:

```
# Make predictions on the random forest model we have built in the previous section

in_sample_preds = rf_model.predict(x_train)
out_sample_preds = rf_model.predict(x_test)
```

With this prediction results, we are going to eveluate how well our random forest model performs in the train and test sets.

In [40]:

```
# accuracy

print('In-Sample Accuracy: %0.4f' % accuracy_score(y_train, in_sample_preds))
print('Out-of-Sample Accuracy: %0.4f' % accuracy_score(y_test, out_sample_preds))
```

```
In-Sample Accuracy: 0.8742
Out-of-Sample Accuracy: 0.8650
```

In [41]:

```
# precision

print('In-Sample Precision: %0.4f' % precision_score(y_train, in_sample_preds))
print('Out-of-Sample Precision: %0.4f' % precision_score(y_test, out_sample_preds))
```

```
In-Sample Precision: 0.9899
Out-of-Sample Precision: 0.9512
```

In [42]:

```
# recall

print('In-Sample Recall: %0.4f' % recall_score(y_train, in_sample_preds))
print('Out-of-Sample Recall: %0.4f' % recall_score(y_test, out_sample_preds))
```

```
In-Sample Recall: 0.1088
Out-of-Sample Recall: 0.0958
```

Accuracy is the measure of correct predictions out of all prediction:

$$\textbf{Accuracy} = \frac{\textbf{Number of Correct Predictions}}{\textbf{Total Number of Records}};$$

Precision is:

$$\textbf{Precision} = \frac{\textbf{True Positive}}{\textbf{True positive} + \textbf{False Positive}};$$

Recall is:

$$\textbf{Recall} = \frac{\textbf{True Positive}}{\textbf{True positive} + \textbf{False Neagative}}.$$

High recall, low precision: Indicates that most of the positive examples are correctly recognized (low FN) but there are a lot of false positives.
Low recall, high precision: Indicates that we miss a lot of positive examples (high FN) but those we predict as positive are indeed positive (low FP).

- ROC & AUC

In [43]:

```
from sklearn.metrics import roc_curve, auc
```

In [44]:

```
# Get the prediction probabilities of the records for the response classes 0 and 1
# For our purpose, take in consideration only the column 1 corresponding
# to the predicted probability of the class 1

in_sample_preds = rf_model.predict_proba(x_train)[:,1]
out_sample_preds = rf_model.predict_proba(x_test)[:,1]
```

Since in our model the response variable has 2 classes, 0 for no responses and 1 for responses, the output of this function has two columns:
1st column: the predicted probability of a negative class, meaning no response for each record
2nd column: the predicted probability of a positive class, meaning a response for each record.
We have selected the 2nd column because we are only interested in the likelihood or responding to the marketing effort.

In [45]:

```
# Compute the ROC curve (receiver Operating Characteristics curve)

in_sample_fpr, in_sample_tpr, in_sample_thresholds = roc_curve(y_train,
                                                               in_sample_preds)
out_sample_fpr, out_sample_tpr, out_sample_thresholds = roc_curve(y_test,
                                                                  out_sample_preds)
```

fpr: False Positive Rate for each given threshold
tpr: True Positive Rate for each given threshold
thresholds: actual thresholds at which fpr and tpr are measured
2 parameters: observed labels and predicted probabilities

In [46]:

```
# Compute AUC (Area Under the Receiver Operating Characteristics)

in_sample_roc_auc = auc(in_sample_fpr, in_sample_tpr)
out_sample_roc_auc = auc(out_sample_fpr, out_sample_tpr)

print('In-Sample AUC: %0.4f' % in_sample_roc_auc)
print('Out-Sample AUC: %0.4f' % out_sample_roc_auc)
```

```
In-Sample AUC: 0.8756
Out-Sample AUC: 0.8266
```

These two values suggest high performance: in effect, an excellent model has AUC near to 1, that happens when the model is able to identify correct predictions.
We can say that a random classifier has an AUC of 0.5, so any classifier with an AUC higher than 0.5 suggests that the model performs better than random predictions.
Moreover, the gap between these two number is small, and this suggest that the model is less overfitting to the train set and can be used to generalize the findings for unforeseen data.

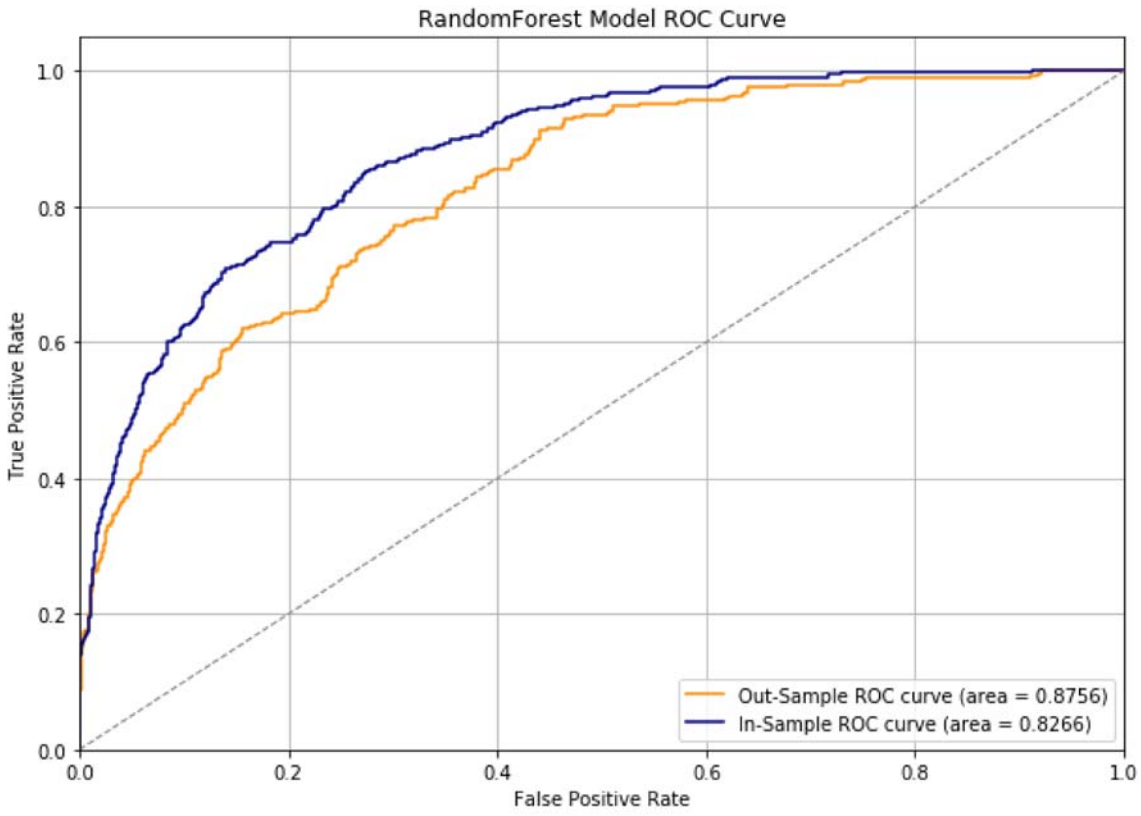
In [47]:

```
# Plot the actual ROC curves

plt.figure(figsize=(10,7))

plt.plot(
    out_sample_fpr, out_sample_tpr, color='darkorange',
    label='Out-Sample ROC curve (area = %0.4f)' % in_sample_roc_auc
)
plt.plot(
    in_sample_fpr, in_sample_tpr, color='navy',
    label='In-Sample ROC curve (area = %0.4f)' % out_sample_roc_auc
)
plt.plot([0, 1], [0, 1], color='gray', lw=1, linestyle='--')
plt.grid()
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('RandomForest Model ROC Curve')
plt.legend(loc="lower right")

plt.show()
```



The plot confirms what examined for the AUC values.