Ensemble Methods

Random Forest and out of Bag error VS Extra Trees (Bootstrap Aggregation/Bagging)

Introduction

We will be using the customer churn data from the telecom industry .

In [2]:
```python
import pandas as pd, numpy as np, matplotlib.pyplot as plt, os, sys, seaborn as sns
```

In [3]:
```python
filepath = 'churndata_processed.csv'
data = pd.read_csv("C:\\Users\\rsnen\\Desktop\\IBM\\churnata_processed for KNN.csv")
```

In [4]:
```python
data.dtypes
```

Out[4]:
```
months                      float64
multiple                      int64
gb_mon                      float64
security                      int64
backup                        int64
protection                    int64
support                       int64
unlimited                     int64
contract                    float64
paperless                     int64
monthly                     float64
satisfaction                float64
churn_value                   int64
payment_Credit Card           int64
payment_Mailed Check          int64
internet_type_DSL             int64
internet_type_Fiber Optic     int64
internet_type_None            int64
offer_Offer A                 int64
offer_Offer B                 int64
offer_Offer C                 int64
offer_Offer D                 int64
offer_Offer E                 int64
dtype: object
```

Examining the Target and Preprocessing

- Examine distribution of the predicted variable (`churn_value`).
- Split the data into train and test sets. Decide if a stratified split should be used or not based on the distribution.
- Examine the distribution of the predictor variable in the train and test data.

In [15]:
```python
target = 'churn_value'
data[target].value_counts()
#data is skewed 85% towords non-churned customers (5174)
```

Out[15]:
```
0    5174
1    1869
Name: churn_value, dtype: int64
```

In [16]:
```python
data[target].value_counts(normalize=True)
#73 percent do not churn, whereas 26.5 do churn.
```

Out[16]:
```
0    0.73463
1    0.26537
Name: churn_value, dtype: float64
```

Given the skew in the predictor variable, let's split the data with the *churned* values being stratified.

In [23]:
```python
from sklearn.model_selection import StratifiedShuffleSplit


feature_cols = [x for x in data.columns if x != target]


# Split the data into two parts with 1500 points in the test data
# going to output our train index and our tests index by calling again, this is a generator objects.
strat_shuff_split = StratifiedShuffleSplit(n_splits=1, test_size=1500, random_state=42)

# Get the index values from the generator
train_idx, test_idx = next(strat_shuff_split.split(data[feature_cols], data[target]))

# Create the data sets
X_train = data.loc[train_idx, feature_cols]
y_train = data.loc[train_idx, target]

X_test = data.loc[test_idx, feature_cols]
y_test = data.loc[test_idx, target]
```

In [24]:
```python
y_train.value_counts(normalize=True)
```

Out[24]:
```
0    0.73462
1    0.26538
Name: churn_value, dtype: float64
```

In [25]:
```python
y_test.value_counts(normalize=True)
```

Out[25]:
```
0    0.734667
1    0.265333
Name: churn_value, dtype: float64
```

Random Forest and Out-of-bag Error.

we will:

- Fit random forest models with a range of tree numbers and evaluate the out-of-bag error for each of these models.
- Plot the resulting oob errors as a function of the number of trees.

Note: since the only thing changing is the number of trees, the `warm_start` flag can be used so that the model just adds more trees to the existing model each time. Use the `set_params` method to update the number of trees.

In [27]:
```python
# Suppress warnings about too few trees from the early models
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=RuntimeWarning)
```

In [32]:
```python
from sklearn.ensemble import RandomForestClassifier

# Initialize the random forest estimator
# Note that the number of trees is not setup here
RF = RandomForestClassifier(oob_score=True,
                            random_state=42,
                            warm_start=True,
                            n_jobs=-1)
#we want to track each one of our out-of-bag errors(oob). So we're going to create an empty list here,
oob_list = list()

# Iterate through all of the possibilities for (i.e)
#we're going to loop through each one of these numbers of trees. So we're going to start with 15 then 20, 30, so on through
#each one of these numbers in our list here up until 400 trees, to see where it plateaus as we increase the number of trees.

for n_trees in [15, 20, 30, 40, 50, 100, 150, 200, 300, 400]:

    # Use this to set the number of trees
    RF.set_params(n_estimators=n_trees)

    # Fit the model
    RF.fit(X_train, y_train)

    # Get the oob error
    oob_error = 1 - RF.oob_score_

    # Store it
    oob_list.append(pd.Series({'n_trees': n_trees, 'oob': oob_error}))

rf_oob_df = pd.concat(oob_list, axis=1).T.set_index('n_trees')

rf_oob_df
```

Out[32]:

n_trees	oob
15.0	0.055566
20.0	0.052138
30.0	0.049973
40.0	0.048890
50.0	0.049071
100.0	0.047447
150.0	0.046726
200.0	0.047447
300.0	0.047988
400.0	0.047808

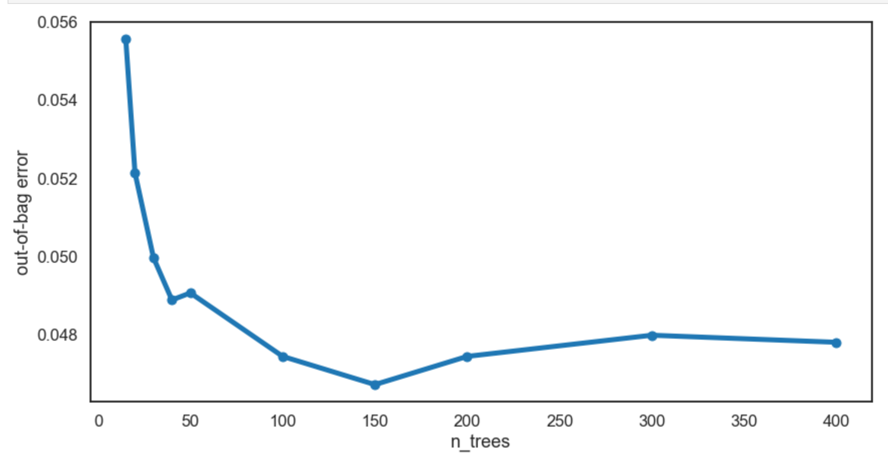
We see the error is gradually decreasing, and then at around 100 to a 150 trees it seems to plateau(stabilized).

In [34]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

In [36]:
```python
sns.set_context('talk')
sns.set_style('white')

ax = rf_oob_df.plot(legend=False, marker='o', figsize=(14, 7), linewidth=5)
ax.set(ylabel='out-of-bag error');
```



ExtraTrees

Here it's going to be all the same steps as Random forest,but Something that we need to do is set that bootstrap argument equal to true. The reason why we do that, is we won't be able to get that out-of-bag error unless we're bootstrapping our model. That bootstrap means that we're taking a sample and that out-of-sample is going to be that out-of-bag.

- (`ExtraTreesClassifier`). Note that the `bootstrap` parameter will have to be set to `True` for this model.
- Compare the out-of-bag errors for the two different types of models.

In general, the default is going to be bootstrap equals false for ExtraTreesClassifier, and then it will fit on the entire dataset. That will be allowed because this ExtraTreesClassifier is more about coming up with random splits than anything else. Then again, we set warm_start = true, oob_score = true. Again, that will only work if the bootstrap is true. Then the number of jobs is just as many jobs as it can run in parallel, given your computer.

In [39]:
```python
from sklearn.ensemble import ExtraTreesClassifier

# Initialize the random forest estimator
# Note that the number of trees is not setup here
EF = ExtraTreesClassifier(oob_score=True,
                          random_state=42,
                          warm_start=True,
                          bootstrap=True,
                          n_jobs=-1)

oob_list = list()

# Iterate through all of the possibilities for
# number of trees
for n_trees in [15, 20, 30, 40, 50, 100, 150, 200, 300, 400]:

    # Use this to set the number of trees
    EF.set_params(n_estimators=n_trees)
    EF.fit(X_train, y_train)

    # oob error
    oob_error = 1 - EF.oob_score_
    oob_list.append(pd.Series({'n_trees': n_trees, 'oob': oob_error}))

et_oob_df = pd.concat(oob_list, axis=1).T.set_index('n_trees')

et_oob_df
```

Out[39]:

n_trees	oob
15.0	0.066570
20.0	0.063864
30.0	0.057550
40.0	0.053942
50.0	0.052318
100.0	0.051236
150.0	0.048890
200.0	0.048530
300.0	0.049612
400.0	0.048530

Combine the two dataframes into a single one for easier plotting.

In [41]:
```python
oob_df = pd.concat([rf_oob_df.rename(columns={'oob':'RandomForest'}),
                    et_oob_df.rename(columns={'oob':'ExtraTrees'})], axis=1)

oob_df
```

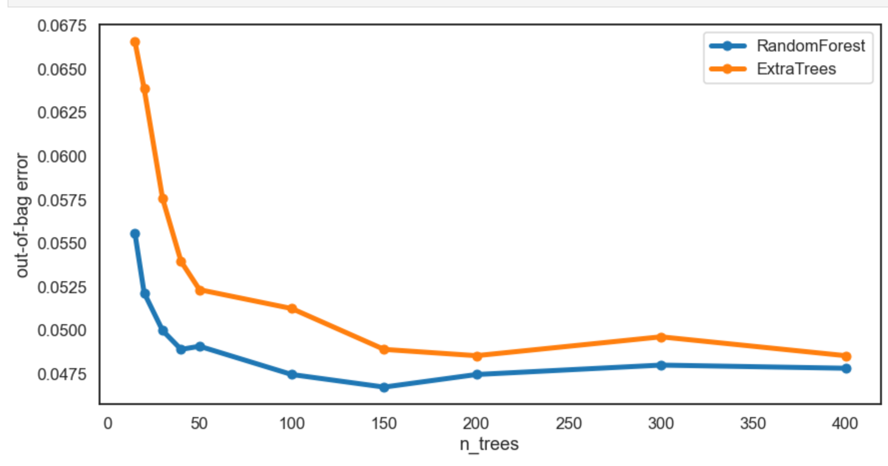
Out[41]:

n_trees	RandomForest	ExtraTrees
15.0	0.055566	0.066570
20.0	0.052138	0.063864
30.0	0.049973	0.057550
40.0	0.048890	0.053942
50.0	0.049071	0.052318
100.0	0.047447	0.051236
150.0	0.046726	0.048890
200.0	0.047447	0.048530
300.0	0.047988	0.049612
400.0	0.047808	0.048530

The random forest model performs consistently better than the extra randomized trees.

In [43]:
```python
sns.set_context('talk')
sns.set_style('white')

ax = oob_df.plot(marker='o', figsize=(14, 7), linewidth=5)
ax.set(ylabel='out-of-bag error');
```



Random forest does perform better across each one of the number of estimators with that line of the error consistently below that of extra trees.

Results

we will:

- Select one of the models that performs well and calculate error metrics and a confusion matrix on the test data set.
- Given the distribution of the predicted class, which metric is most important? Which could be deceiving?

In [49]:
```python
# Random forest with 100 estimators
model = RF.set_params(n_estimators=100)

y_pred = model.predict(X_test)

#get probabilities for each of the two categories
y_prob = model.predict_proba(X_test)
```

In [51]:
```python
y_prob
```

Out[51]:
```
array([[0.035 , 0.965 ],
       [1.    , 0.    ],
       [0.655 , 0.345 ],
       ...,
       [0.0525, 0.9475],
       [0.97  , 0.03  ],
       [0.9825, 0.0175]])
```

Unsurprisingly, recall is rather poor for the customers who churned (True) class since they are quite small. We are doing better than random guessing, though, as the accuracy is 0.96 (vs 0.85 for random guessing).

In [52]:
```python
from sklearn.metrics import classification_report, accuracy_score, precision_score, recall_score
from sklearn.metrics import f1_score, roc_auc_score

cr = classification_report(y_test, y_pred)
print(cr)

score_df = pd.DataFrame({'accuracy': accuracy_score(y_test, y_pred),
                         'precision': precision_score(y_test, y_pred),
                         'recall': recall_score(y_test, y_pred),
                         'f1': f1_score(y_test, y_pred),
                         'auc': roc_auc_score(y_test, y_pred)},
                         index=pd.Index([0]))

print(score_df)
```

```
              precision    recall  f1-score   support

           0       0.94      0.98      0.96      1102
           1       0.94      0.83      0.88       398

    accuracy                           0.94      1500
   macro avg       0.94      0.90      0.92      1500
weighted avg       0.94      0.94      0.94      1500

   accuracy  precision    recall    f1       auc
0      0.94     0.9375  0.829146  0.88  0.904591
```

For negative class, which has a lot more values, we see the higher support, fairly high scores. Then for the positive class, which is that they did churn. We didn't do quite as well, there's a smaller fraction. We did okay, in terms of what we call predicting the actual values better than just a coin flip, but not great. Then we see that each one of these will match up with the scores that we have for our positive class. Then you see that we are also able to get the AUC score, which was fairly high as well. Again, for that positive class.

Examining Results

- Ploting the feature importances.

In [59]:
```python
feature_imp = pd.Series(model.feature_importances_, index=feature_cols).sort_values(ascending=False)

ax = feature_imp.plot(kind='bar', figsize=(16, 6))
ax.set(ylabel='Relative Importance');
ax.set(ylabel='Feature');
```