Table of Contents

Establish Baseline

Baseline Accuracy

We start by answering the question, "how accurate would my model be if everything was a 1?"

```
# get ZeroR accuracy
print y.mean()
```

```
0.62392
```

We would be **62.4%** correct if we guessed every customer in the data set was going to churn. We then create a single training split and train a Random Forest with some default parameters to see how the model performs without any tuning.

```
from sklearn.cross_validation import train_test_split
from sklearn.ensemble import RandomForestClassifier

# get rf accuracy with out of box settings
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
rf = RandomForestClassifier(n_jobs=-1, random_state=0)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)

print 'Training accuracy:', round(rf.score(X_train, y_train), 4)
print 'Test accuracy:', round(rf.score(X_test, y_test), 4)
```

```
Training accuracy: 0.9863
Test accuracy: 0.7533
```

75.3% accurate on that one split. **n_jobs** determines how many processors to run a task on. n_jobs=-1 means use all of them. **random_state** makes it so that the cross validation folds are always the same, as long as random_state is set to 0. Training accuracy is extremely high for random forest because it works by overfitting on training data, and taking a vote using the overfit model, on test data. We can expect our test accuracy to be around 75.3%... But that's not the whole picture.

Accuracy Ranges

We can also get a sense for what a good accuracy range looks like by getting mean and standard deviation test accuracy using K-Folds Cross Validation. **cross_val_score** handles splitting, fitting and predicting all at once.

```
from sklearn.model_selection import cross_val_score

# efficient cv code
scores = cross_val_score(estimator=rf, X=X, y=y, cv=10, n_jobs=-1)
print('CV accuracy scores: %s' % scores)
print('CV accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
```

```
CV accuracy scores: [ 0.74865027  0.75824835  0.74705059  0.74985003  0.7454      0.7498
  0.75195039  0.76075215  0.76195239  0.74874975]
CV accuracy: 0.752 +/- 0.006
```

Now we know our reasonable **1 standard deviation accuracy range** is 75.2% +/- 0.6%.

Other Metrics

Other very metrics include **AUC, Precision, Recall and F1**. We will go into detail with those later. It's most important to note: accuracy is general accuracy—how often you guess the 0 and 1 classes correctly—but **F1 is like accuracy with emphasis on the positive class.** In this exercise since it's very important to identify Churners, we should focus on F1, but for the purpose of this exercise, we will focus on Accuracy.

```
from sklearn import metrics

# Accuracy, AUC, Precision, Recall, F1
print 'Key Metrics \n******************'
print 'Accuracy: %.3f' % metrics.accuracy_score(y_true=y_test, y_pred=y_pred_rf)
print 'ROC AUC: %.3f' % metrics.roc_auc_score(y_true=y_test, y_score=y_pred_rf)
print 'Classification Report: \n\n {0}'.format(metrics.classification_report(y_test, y_pred_rf))
```

```
Key Metrics
******************
Accuracy: 0.753
ROC AUC: 0.741
Classification Report:

             precision    recall  f1-score   support

          0       0.67      0.69      0.68      3822
          1       0.80      0.79      0.80      6178

avg / total       0.75      0.75      0.75     10000
```

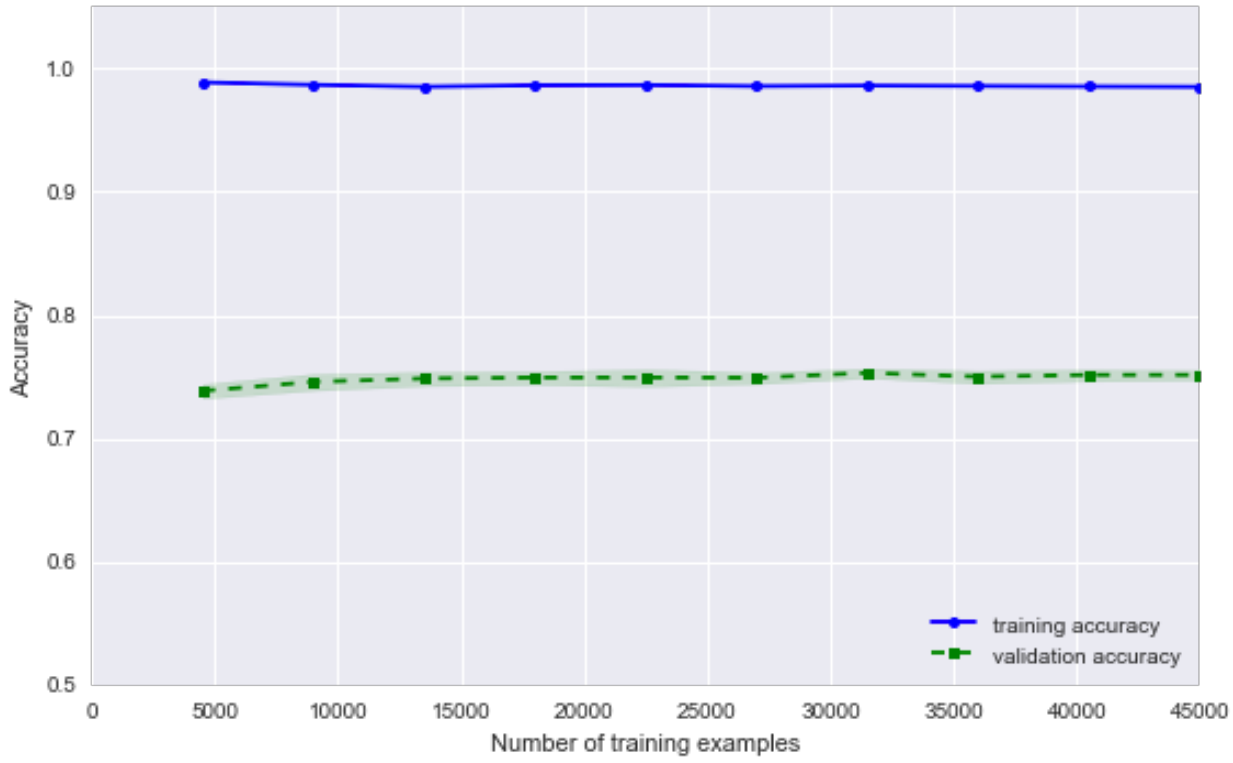
Bias/Variance

Learning Curves

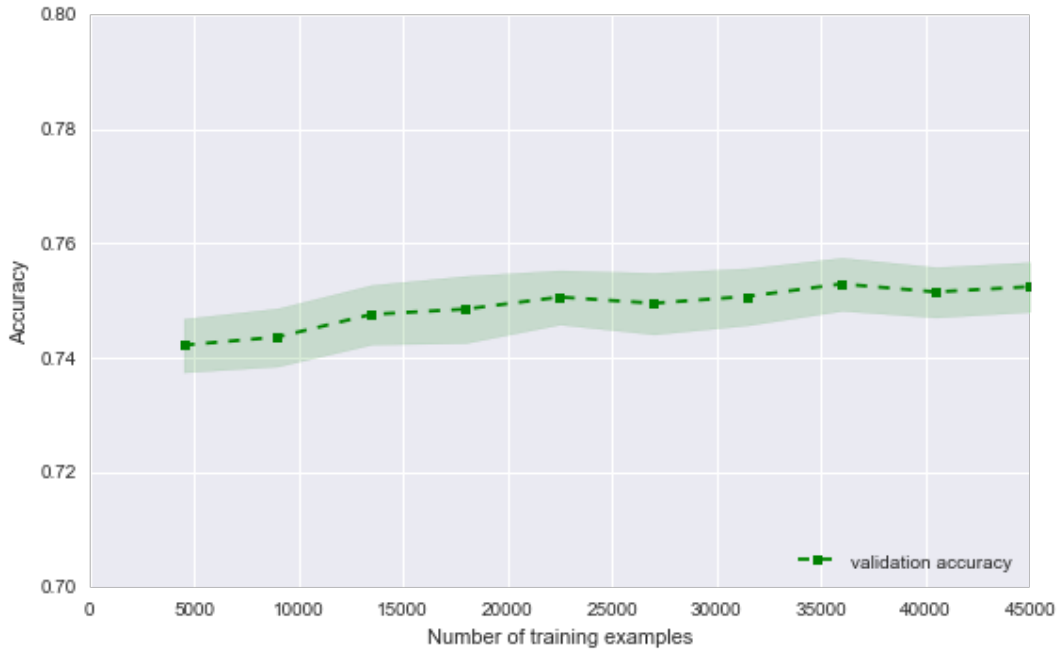
Learning curves tell you how much more accurate your model becomes as it uses more training data.

```
# tells you how accurate model becomes with more data
from sklearn.model_selection import learning_curve

# create cv estimates for range of train_sizes
train_sizes, train_scores, test_scores = \
    learning_curve(
        estimator=rf,
        X=X,
        y=y,
        train_sizes=np.linspace(0.1, 1.0, 10),
        cv=10,
        n_jobs=-1
        )

# get mean + sd, and plot
train_mean_lc = np.mean(train_scores, axis=1)
train_std_lc = np.std(train_scores, axis=1)
test_mean_lc = np.mean(test_scores, axis=1)
test_std_lc = np.std(test_scores, axis=1)
```

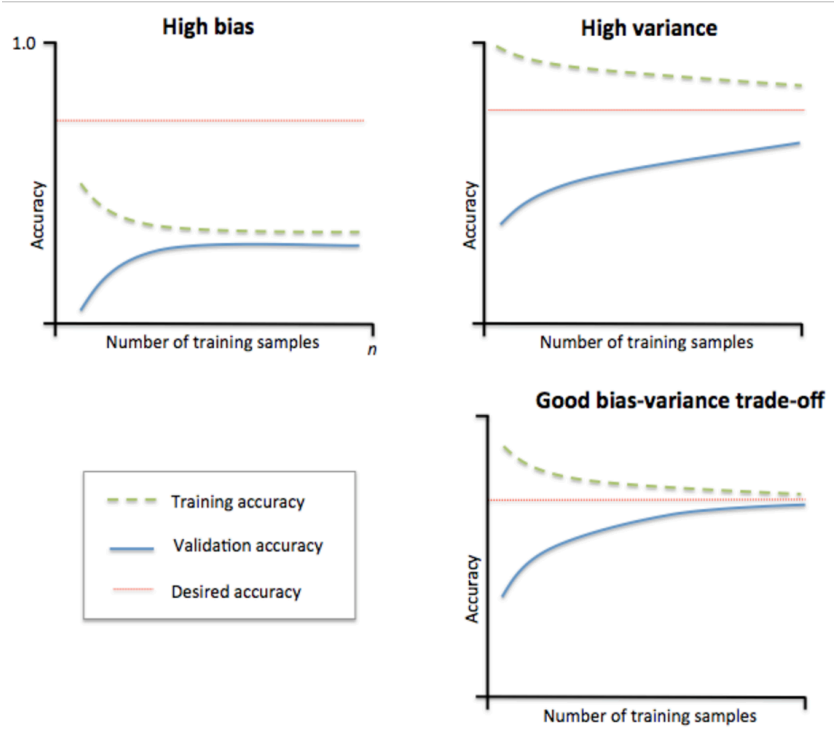
The **learning_curve** function, like cross_val_predict, handles cross validation, fitting and predicting in one go. It also produces an array of values with levels—training sizes—and test/train scores.



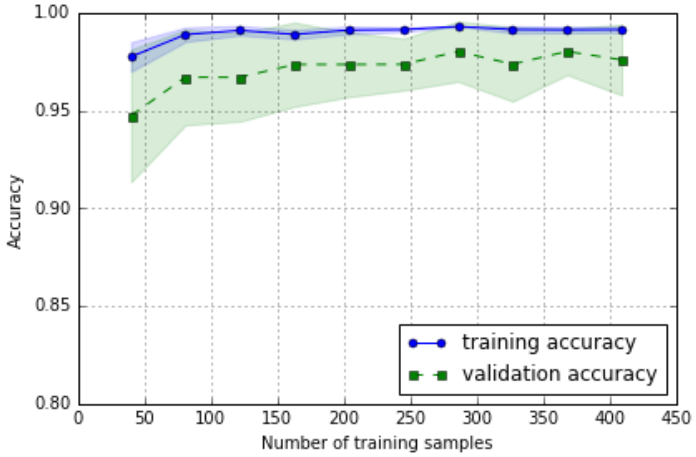
Since Random Forest works by overfitting to training data, it's no surprise that training and test accuracy are so different. By focusing on just test accuracy, we can see how this Random Forest is to getting more training data:



Generally, in addition to telling you how your # samples affects accuracy, learning curves are also supposed to tell you how **biased or variant** your model is. Here are examples of what other curves look like:



High bias means the model doesn't know how to fit this data. Whether testing or training, it's not very accurate. High variance means the model is very sensitive to the data you give it. It can fit very well to training data, but is sensitive to the variation in the test data. A good model that doesn't specifically work by overfitting, has similar test and training accuracy at an acceptable level of accuracy. This is a healthy looking bias/variance curve for a Logistic Regression:

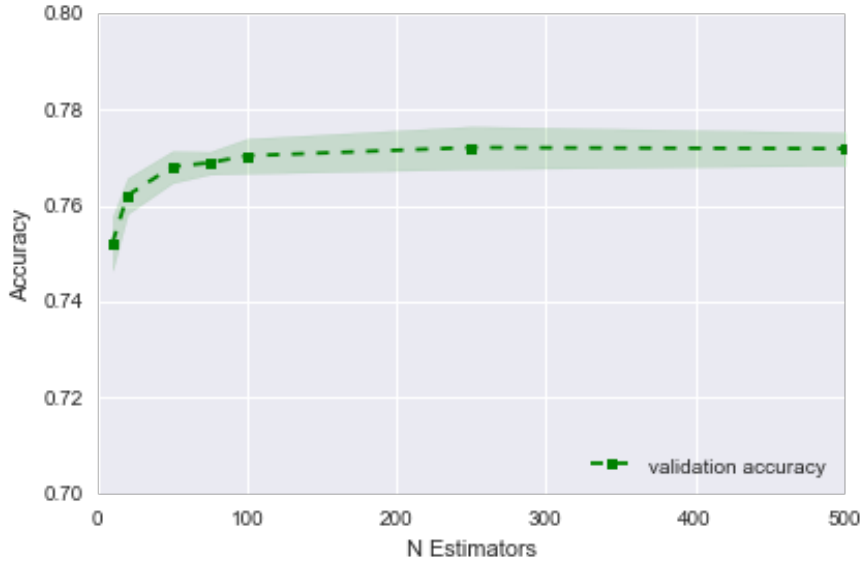


Validation Curve

As with Learning Curves, sci-kit learn has a method for creating validation curves that does cross validation and prediction for you. It's called validation_curve. These are identical to learning curves except the x-axis is hyperparameter. So you're seeing how sensitive the model is in changes to a single hyperparameter.

```
# addresses over/underfitting with different parameter estimates
from sklearn.model_selection import validation_curve

# Validation Curve: iterate through levels of a parameter to see impact on accuracy
#   Uses CV to get ranges of accuracy scores
param_range = [10, 20, 50, 75, 100, 250, 500]

# get cross validation scores
train_scores, test_scores = \
    validation_curve(estimator=rf,
                     X=X,
                     y=y,
                     param_name='n_estimators',
                     param_range=param_range,
                     cv=10)

# get mean + sd, and plot
train_mean_vc = np.mean(train_scores, axis=1)
train_std_vc = np.std(train_scores, axis=1)
test_mean_vc = np.mean(test_scores, axis=1)
test_std_vc = np.std(test_scores, axis=1)
```

In this case we'll play with the **n_estimators** parameter, which is the number of trees used to construct the forest. More is always better, but there's a clear inflection in the cost/benefit to running this with more trees. It appears the cutoff is 100 trees, here:



Grid Search

Grid Search enables scanning more than one parameter at the same time to find the **optimal mix** of hyperparameter values. First we create a "grid" of hyperparameter values to search through, focusing on **max_features** (the maximum number of features a single tree can use to make a prediction) and **min_leaf** (the minimum number of leafs there have to be at the end of every branch).

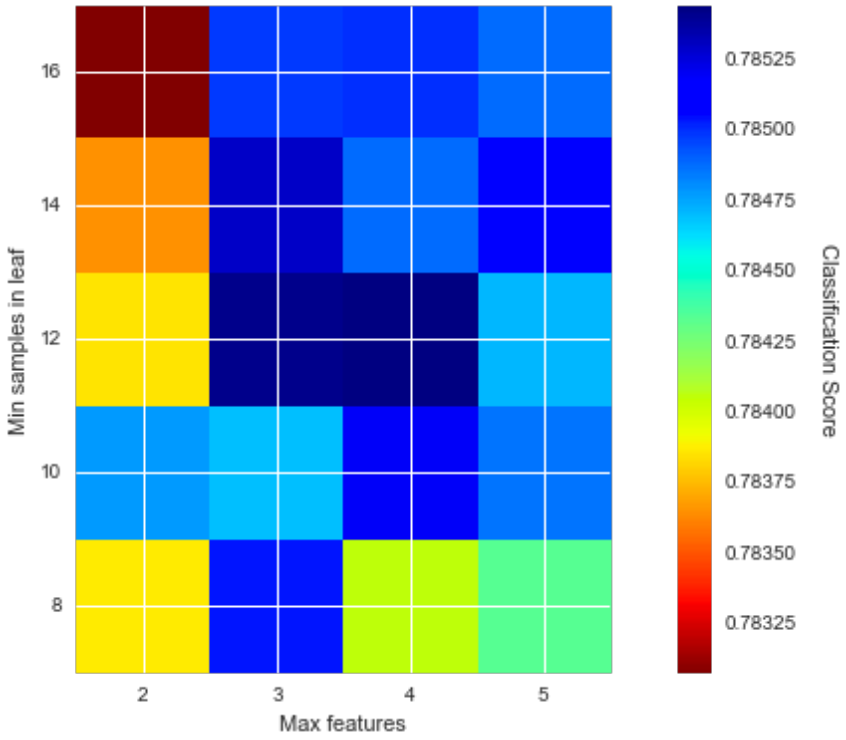
```
# tuning more than 1 hyperparameter
from sklearn.model_selection import GridSearchCV

# create parameter ranges
# rf parameters for accuracy: max_features, n_estimators, min_sample_leaf
# rf parameters for easier training: n_jobs, random_state, oob_score
max_features = [2, 3, 4, 5]  # default is sqrt(n_features)
min_sample_leaf = [8, 10, 12, 14, 16]  # default is 1

# create hyperparameter "grid"
hyperparameters = {
    'max_features': max_features,
    'min_samples_leaf': min_sample_leaf,
    }
```

We then fit Random Forests with every value of those grid parameters to get accuracy estimates. **GridSearchCV** has an attribute which is the best values for each search:

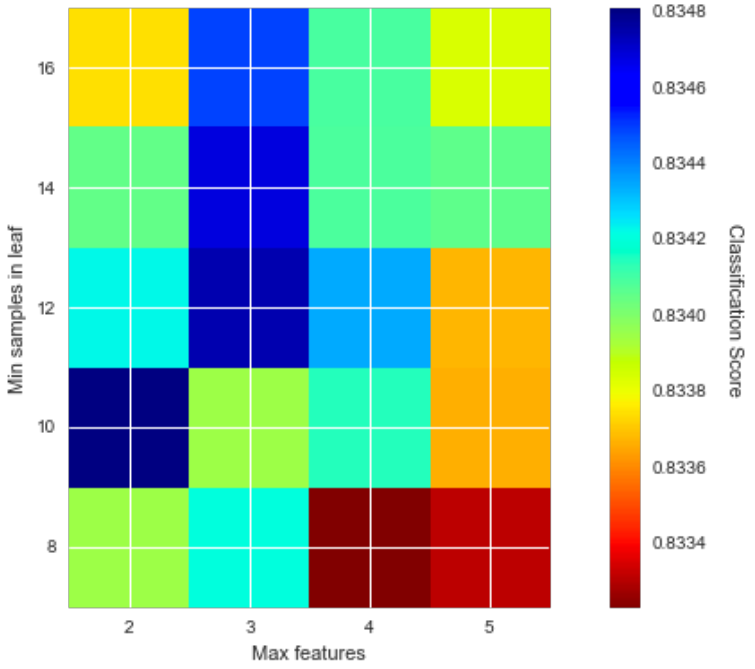
```
# create grid search object with embedded CV method
gridCV = GridSearchCV(
    RandomForestClassifier(n_estimators=100, random_state=1, n_jobs=-1),
    param_grid=hyperparameters,
    scoring='accuracy',  # try 'accuracy' and 'f1'
    cv=10,
    n_jobs=-1
    )

# perform grid search
gridCV.fit(X, y)

# identify optimal values
best_max_features     = gridCV.best_params_['max_features']
best_min_samples_leaf = gridCV.best_params_['min_samples_leaf']
# best_n_estimators     = gridCV.best_params_['n_estimators']

print("The best performing max_features     value is: {}".format(best_max_features))
print("The best performing min_samples_leaf value is: {}".format(best_min_samples_leaf))
```

```
The best performing max_features     value is: 4
The best performing min_samples_leaf value is: 12
```

After printing the grid as a heatmap, we find the optimal values for accuracy are max_features=3 and and min_leaf=12 for an accuracy of 78.5%--a 3% improvement over the previous model:



Running this again for f1 score we get:

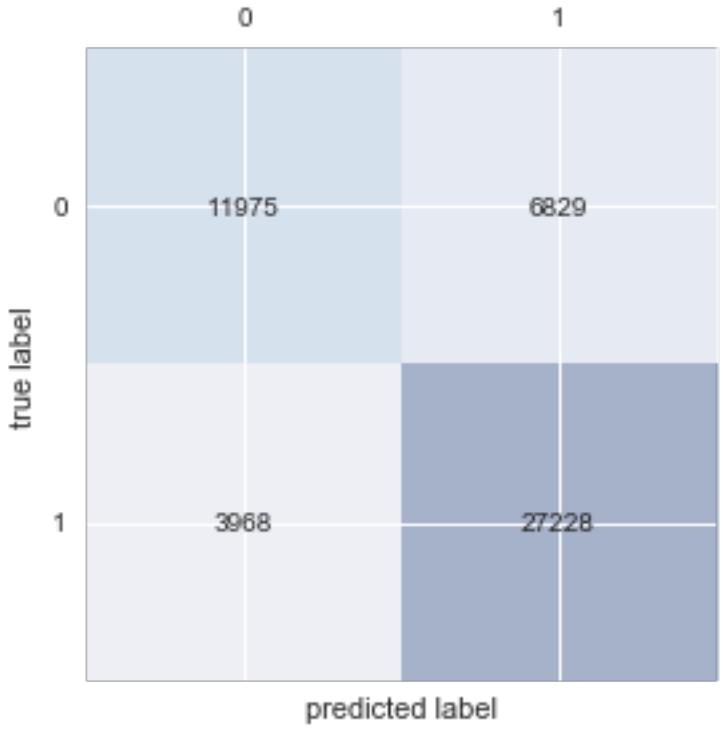


Given this data, what's the optimal model? It depends on whether we care about class separation or correctly guessing the target class. If the former, then accuracy, if the latter, then f1. Since we care about not losing customers, the focus should be f1. Let's build the optimal model:

```
rf = RandomForestClassifier(
    n_estimators=500,
    min_samples_leaf=10,
    max_features=2,
    random_state=0,
    n_jobs=-1)
```

Other Performance Analysis Tools

Confusion Matrix



Precision, Recall, and F1

```
# get precision and recall
print 'Precision: %.3f' % metrics.precision_score(y_true=y_test, y_pred=y_pred)
print 'Recall: %.3f' % metrics.recall_score(y_true=y_test, y_pred=y_pred)
print 'F1: %.3f' % metrics.f1_score(y_true=y_test, y_pred=y_pred)

# exercise: how is f1 calculated?  why is it the harmonic mean and not the geometric mean?
```

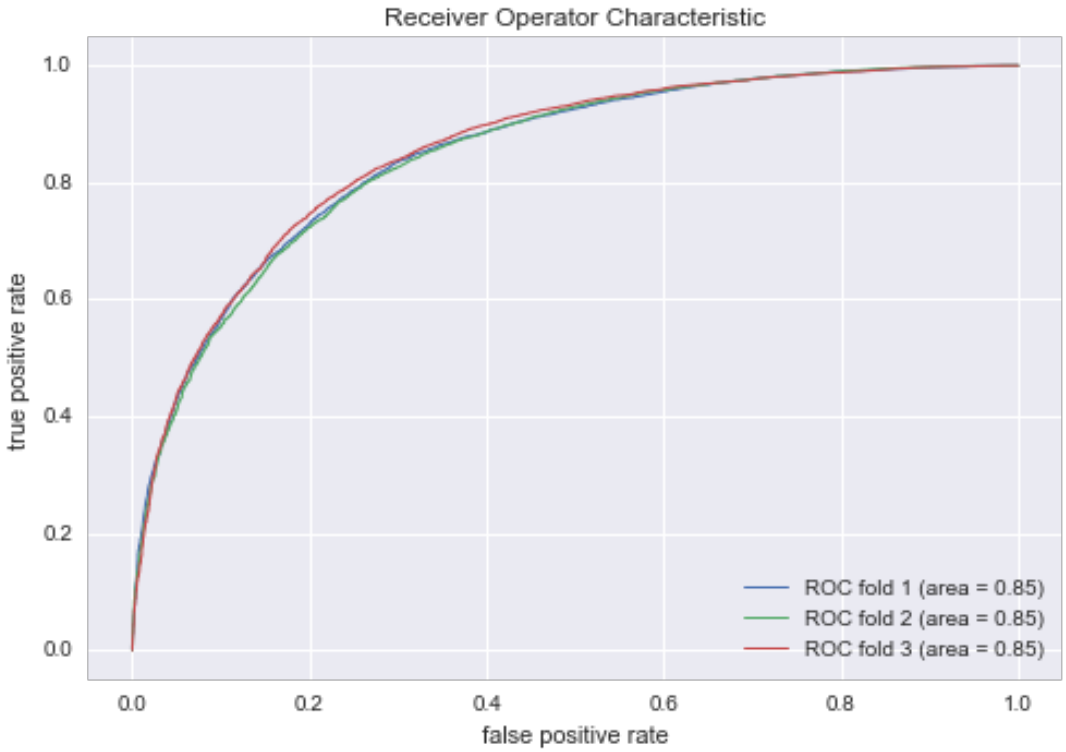
```
Precision: 0.798
Recall: 0.873
F1: 0.834
```

Precision = the % of times we guessed a customer would churn and they did
Recall = the % of churners we were able to identify
F1 = the harmonic mean of Precision and Recall

Harmonic Mean: https://www.youtube.com/watch?v=IaQwWINzjtA&t=2s

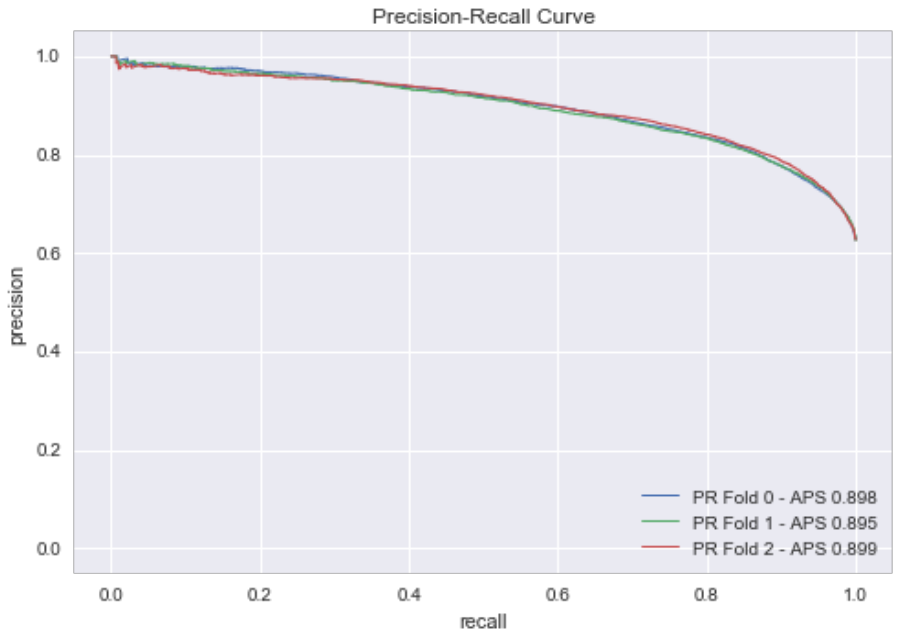
ROC and AUC

ROC curves tell you how well you're ranking your customers by churn probability.



Precision-Recall Curve

Tells you how well you're identifying churners.



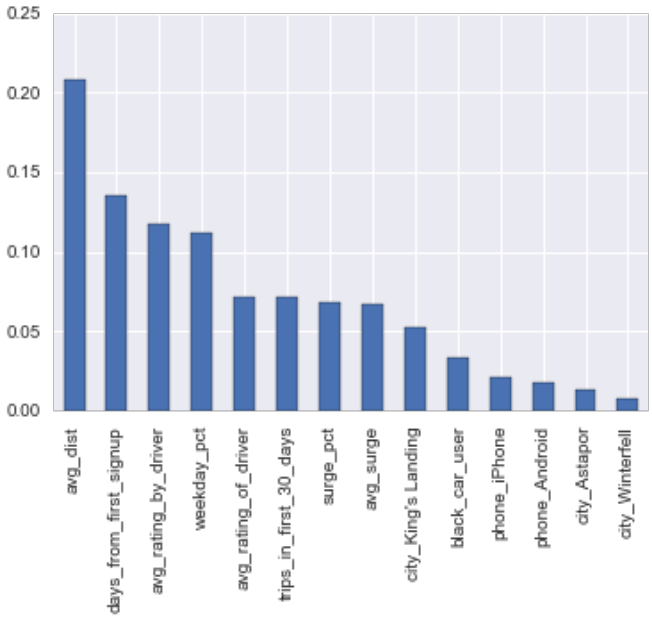
Summary Report

Through hyperparameter tuning we were able to bring Accuracy up 3% and F1 Score up 3%, meaning we both have better class separation and we are identifying churners better. What are some other things we could do to get better predictions?

```
ROC AUC: 0.755
Accuracy: 0.784
Classification Report:

             precision    recall  f1-score   support

          0       0.75      0.64      0.69     18804
          1       0.80      0.87      0.83     31196

avg / total       0.78      0.78      0.78     50000
```

Feature Importance

And finally, what are the predictors that most influence this model?


0.25
0.20
0.15
0.10
0.05
0.00
avg_dist
days_from_first_signup
avg_rating_by_driver
weekday_pct
avg_rating_of_driver
trips_in_first_30_days
surge_pct
avg_surge
city_King's Landing
black_car_user
phone_iPhone
phone_Android
city_Astapor
city_Winterfell