Random Forest - Credit Default Prediction

In this lab, we will build a random forest model to predict whether a given customer defaults or not. Credit default is one of the most important problems in the banking and risk analytics industry. There are various attributes which can be used to predict default, such as demographic data (age, income, employment status, etc.), (credit) behavioural data (past loans, payment, number of times a credit payment has been delayed by the customer etc.).

We'll start the process with data cleaning and preparation and then tune the model to find optimal hyperparameters.

Data Understanding and Cleaning

In [73]:
```
# Importing the required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# To ignore warnings
import warnings
warnings.filterwarnings("ignore")
```

In [74]:
```
# Reading the csv file and putting it into 'df' object.
df = pd.read_csv('credit-card-default.csv')
df.head()
```

Out[74]:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...
0	1	20000	2	2	1	24	2	2	-1	-1	...
1	2	120000	2	2	2	26	-1	2	0	0	...
2	3	90000	2	2	2	34	0	0	0	0	...
3	4	50000	2	2	1	37	0	0	0	0	...
4	5	50000	1	2	1	57	-1	0	-1	0	...

5 rows × 25 columns

In [75]:
```
# Let's understand the type of columns
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
ID           30000 non-null int64
LIMIT_BAL    30000 non-null int64
SEX          30000 non-null int64
EDUCATION    30000 non-null int64
MARRIAGE     30000 non-null int64
AGE          30000 non-null int64
PAY_0        30000 non-null int64
PAY_2        30000 non-null int64
PAY_3        30000 non-null int64
PAY_4        30000 non-null int64
PAY_5        30000 non-null int64
PAY_6        30000 non-null int64
BILL_AMT1    30000 non-null int64
BILL_AMT2    30000 non-null int64
BILL_AMT3    30000 non-null int64
BILL_AMT4    30000 non-null int64
BILL_AMT5    30000 non-null int64
BILL_AMT6    30000 non-null int64
PAY_AMT1     30000 non-null int64
PAY_AMT2     30000 non-null int64
PAY_AMT3     30000 non-null int64
PAY_AMT4     30000 non-null int64
PAY_AMT5     30000 non-null int64
PAY_AMT6     30000 non-null int64
defaulted    30000 non-null int64
dtypes: int64(25)
memory usage: 5.7 MB
```

In this case, we know that there are no major data quality issues, so we'll go ahead and build the model.

Data Preparation and Model Building

In [76]:
```
# Importing test_train_split from sklearn library
from sklearn.model_selection import train_test_split
```

In [77]:
```
# Putting feature variable to X
X = df.drop('defaulted',axis=1)

# Putting response variable to y
y = df['defaulted']

# Splitting the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, rand
om_state=101)
```

Default Hyperparameters

Let's first fit a random forest model with default hyperparameters.

In [78]:
```
# Importing random forest classifier from sklearn library
from sklearn.ensemble import RandomForestClassifier

# Running the random forest with default parameters.
rfc = RandomForestClassifier()
```

In [79]:
```
# fit
rfc.fit(X_train,y_train)
```

Out[79]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, 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=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [80]:
```
# Making predictions
predictions = rfc.predict(X_test)
```

In [108]:
```
# Importing classification report and confusion matrix from sklearn metrics
from sklearn.metrics import classification_report,confusion_matrix, accuracy_s
core
```

In [82]:
```
# Let's check the report of our default model
print(classification_report(y_test,predictions))
```

```
             precision    recall  f1-score   support

          0       0.83      0.94      0.89      7058
          1       0.61      0.32      0.42      1942

avg / total       0.79      0.81      0.78      9000
```

In [83]:
```
# Printing confusion matrix
print(confusion_matrix(y_test,predictions))
```

```
[[6663  395]
 [1325  617]]
```

In [109]:
```
print(accuracy_score(y_test,predictions))
```

```
0.827666666667
```

So far so good, let's now look at the list of hyperparameters which we can tune to improve model performance.

Hyperparameter Tuning

The following hyperparameters are present in a random forest classifier. Note that most of these hypereparameters are actually of the decision trees that are in the forest.

- **n_estimators**: integer, optional (default=10): The number of trees in the forest.
- **criterion**: string, optional (default="gini")The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.
- **max_features** : int, float, string or None, optional (default="auto")The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features).
  - If "sqrt", then max_features=sqrt(n_features) (same as "auto").
  - If "log2", then max_features=log2(n_features).
  - If None, then max_features=n_features.
  - Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
- **max_depth** : integer or None, optional (default=None)The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- **min_samples_split** : int, float, optional (default=2)The minimum number of samples required to split an internal node:**
  - **If int, then consider min_samples_split as the minimum number.
  - **If float, then min_samples_split is a percentage and ceil(min_samples_split, n_samples) are the minimum number of samples for each split.
- **min_samples_leaf** : int, float, optional (default=1)The minimum number of samples required to be at a leaf node:**
  - **If int, then consider min_samples_leaf as the minimum number.**
  - **If float, then min_samples_leaf is a percentage and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.**
- **min_weight_fraction_leaf** : float, optional (default=0.)The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
- **max_leaf_nodes** : int or None, optional (default=None)Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.
- **min_impurity_split** : float,Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Tuning max_depth

Let's try to find the optimum values for max_depth and understand how the value of max_depth impacts the overall accuracy of the ensemble.

In [84]:
```
# GridSearchCV to find optimal n_estimators
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV


# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'max_depth': range(2, 20, 5)}

# instantiate the model
rf = RandomForestClassifier()


# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy")
rf.fit(X_train, y_train)
```

Out[84]:
```
GridSearchCV(cv=5, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, cr
iterion='gini',
            max_depth=None, 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=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'max_depth': range(2, 20, 5)}, pre_dispatch='2*n_jobs',
       refit=True, return_train_score='warn', scoring='accuracy',
       verbose=0)
```

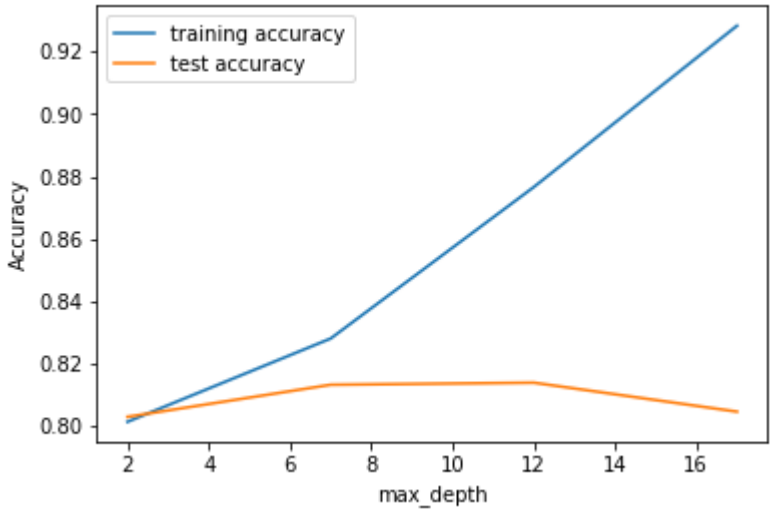
In [85]:
```
# scores of GridSearch CV
scores = rf.cv_results_
pd.DataFrame(scores).head()
```

Out[85]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_max_de
0	0.096119	0.004265	0.802905	0.801310	2
1	0.268716	0.007664	0.813190	0.828012	7
2	0.380998	0.010035	0.813857	0.876583	12
3	0.474810	0.010995	0.804571	0.928286	17

4 rows × 21 columns

In [86]:
```
# plotting accuracies with max_depth
plt.figure()
plt.plot(scores["param_max_depth"],
         scores["mean_train_score"],
         label="training accuracy")
plt.plot(scores["param_max_depth"],
         scores["mean_test_score"],
         label="test accuracy")
plt.xlabel("max_depth")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```



You can see that as we increase the value of max_depth, both train and test scores increase till a point, but after that test score starts to decrease. The ensemble tries to overfit as we increase the max_depth.

Thus, controlling the depth of the constituent trees will help reduce overfitting in the forest.

Tuning n_estimators

Let's try to find the optimum values for n_estimators and understand how the value of n_estimators impacts the overall accuracy. Notice that we'll specify an appropriately low value of max_depth, so that the trees do not overfit.

In [87]:
```
# GridSearchCV to find optimal n_estimators
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV


# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'n_estimators': range(100, 1500, 400)}

# instantiate the model (note we are specifying a max_depth)
rf = RandomForestClassifier(max_depth=4)


# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy")
rf.fit(X_train, y_train)
```

Out[87]:
```
GridSearchCV(cv=5, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, cr
iterion='gini',
            max_depth=4, 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=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'n_estimators': range(100, 1500, 400)},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='accuracy', verbose=0)
```

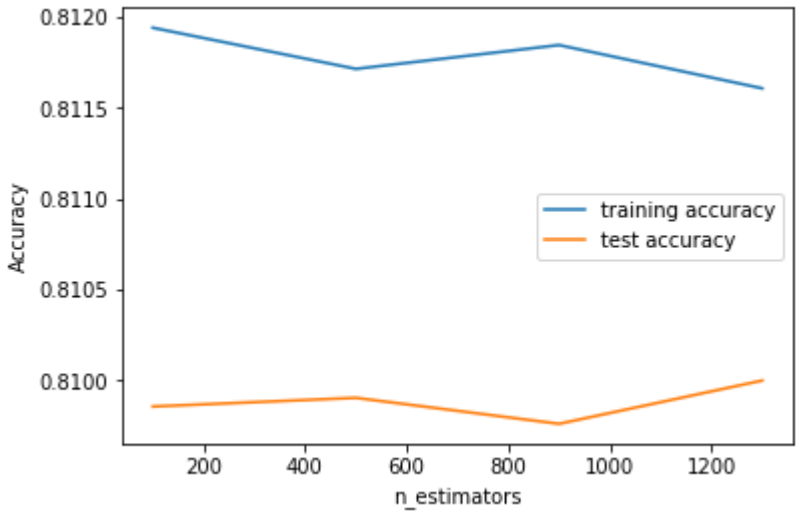
In [88]:
```
# scores of GridSearch CV
scores = rf.cv_results_
pd.DataFrame(scores).head()
```

Out[88]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_n_estin
0	1.529487	0.044068	0.809857	0.811940	100
1	7.696058	0.196299	0.809905	0.811714	500
2	13.122141	0.351332	0.809762	0.811845	900
3	18.787532	0.467269	0.810000	0.811607	1300

4 rows × 21 columns

In [89]:
```
# plotting accuracies with n_estimators
plt.figure()
plt.plot(scores["param_n_estimators"],
         scores["mean_train_score"],
         label="training accuracy")
plt.plot(scores["param_n_estimators"],
         scores["mean_test_score"],
         label="test accuracy")
plt.xlabel("n_estimators")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```



Tuning max_features

Let's see how the model performance varies with `max_features`, which is the maximum numbre of features considered for splitting at a node.

In [90]:
```
# GridSearchCV to find optimal max_features
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV


# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'max_features': [4, 8, 14, 20, 24]}

# instantiate the model
rf = RandomForestClassifier(max_depth=4)


# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy")
rf.fit(X_train, y_train)
```

Out[90]:
```
GridSearchCV(cv=5, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, cr
iterion='gini',
            max_depth=4, 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=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'max_features': [4, 8, 14, 20, 24]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='accuracy', verbose=0)
```

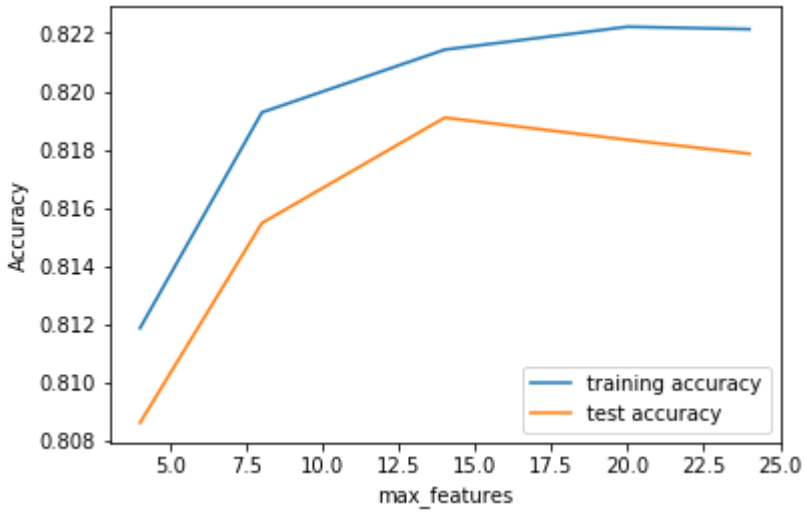
In [91]:
```
# scores of GridSearch CV
scores = rf.cv_results_
pd.DataFrame(scores).head()
```

Out[91]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_max_fe
0	0.145784	0.004746	0.808619	0.811869	4
1	0.237351	0.005474	0.815476	0.819274	8
2	0.376728	0.004831	0.819095	0.821429	14
3	0.520276	0.004587	0.818333	0.822214	20
4	0.610253	0.004516	0.817857	0.822131	24

5 rows × 21 columns

In [92]:
```
# plotting accuracies with max_features
plt.figure()
plt.plot(scores["param_max_features"],
         scores["mean_train_score"],
         label="training accuracy")
plt.plot(scores["param_max_features"],
         scores["mean_test_score"],
         label="test accuracy")
plt.xlabel("max_features")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```



Apparently, the training and test scores *both* seem to increase as we increase max_features, and the model doesn't seem to overfit more with increasing max_features. Think about why that might be the case.

Tuning min_samples_leaf

The hyperparameter **min_samples_leaf** is the minimum number of samples required to be at a leaf node:

- If int, then consider min_samples_leaf as the minimum number.
- If float, then min_samples_leaf is a percentage and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

Let's now check the optimum value for min samples leaf in our case.

In [93]:
```
# GridSearchCV to find optimal min_samples_leaf
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV


# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'min_samples_leaf': range(100, 400, 50)}

# instantiate the model
rf = RandomForestClassifier()


# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy")
rf.fit(X_train, y_train)
```

Out[93]:
```
GridSearchCV(cv=5, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, cr
iterion='gini',
            max_depth=None, 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=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'min_samples_leaf': range(100, 400, 50)},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='accuracy', verbose=0)
```

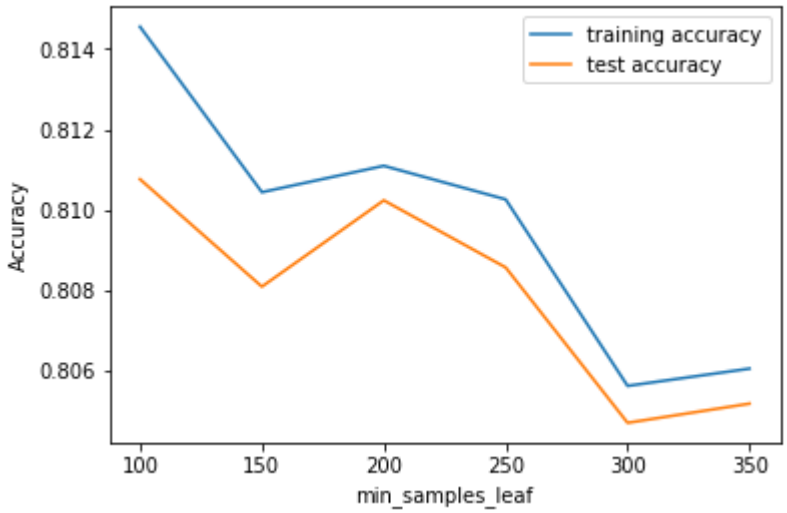
In [94]:
```
# scores of GridSearch CV
scores = rf.cv_results_
pd.DataFrame(scores).head()
```

Out[94]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_min_sa
0	0.246837	0.007659	0.810762	0.814548	100
1	0.219095	0.006198	0.808095	0.810440	150
2	0.209188	0.006197	0.810238	0.811095	200
3	0.197544	0.005801	0.808571	0.810262	250
4	0.185529	0.005814	0.804714	0.805631	300

5 rows × 21 columns

In [95]:
```
# plotting accuracies with min_samples_leaf
plt.figure()
plt.plot(scores["param_min_samples_leaf"],
         scores["mean_train_score"],
         label="training accuracy")
plt.plot(scores["param_min_samples_leaf"],
         scores["mean_test_score"],
         label="test accuracy")
plt.xlabel("min_samples_leaf")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```



You can see that the model starts of overfit as you decrease the value of min_samples_leaf.

Tuning min_samples_split

Let's now look at the performance of the ensemble as we vary min_samples_split.

In [96]:
```
# GridSearchCV to find optimal min_samples_split
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV


# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'min_samples_split': range(200, 500, 50)}

# instantiate the model
rf = RandomForestClassifier()


# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy")
rf.fit(X_train, y_train)
```

Out[96]:
```
GridSearchCV(cv=5, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, cr
iterion='gini',
            max_depth=None, 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=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'min_samples_split': range(200, 500, 50)},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='accuracy', verbose=0)
```

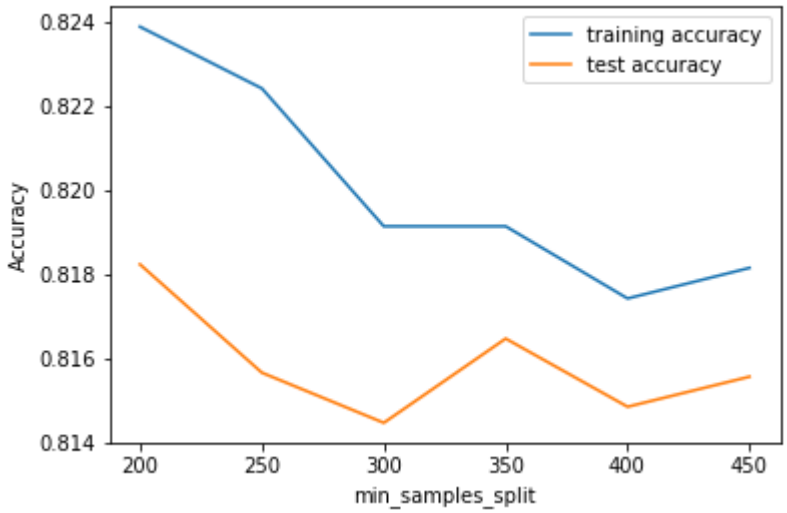
In [97]:
```
# scores of GridSearch CV
scores = rf.cv_results_
pd.DataFrame(scores).head()
```

Out[97]:

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_min_sa
0	0.339940	0.007539	0.818238	0.823881	200
1	0.323215	0.007094	0.815667	0.822417	250
2	0.340311	0.007908	0.814476	0.819143	300
3	0.354950	0.006895	0.816476	0.819143	350
4	0.332439	0.008043	0.814857	0.817428	400

5 rows × 21 columns

In [98]:
```
# plotting accuracies with min_samples_split
plt.figure()
plt.plot(scores["param_min_samples_split"],
         scores["mean_train_score"],
         label="training accuracy")
plt.plot(scores["param_min_samples_split"],
         scores["mean_test_score"],
         label="test accuracy")
plt.xlabel("min_samples_split")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```



Grid Search to Find Optimal Hyperparameters

We can now find the optimal hyperparameters using GridSearchCV.

In [99]:
```
# Create the parameter grid based on the results of random search
param_grid = {
    'max_depth': [4,8,10],
    'min_samples_leaf': range(100, 400, 200),
    'min_samples_split': range(200, 500, 200),
    'n_estimators': [100,200, 300],
    'max_features': [5, 10]
}
# Create a based model
rf = RandomForestClassifier()
# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                           cv = 3, n_jobs = -1,verbose = 1)
```

In [100]:
```
# Fit the grid search to the data
grid_search.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 72 candidates, totalling 216 fits

[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   56.8s
[Parallel(n_jobs=-1)]: Done 192 tasks      | elapsed:  6.8min
[Parallel(n_jobs=-1)]: Done 216 out of 216 | elapsed:  8.1min finished
```

Out[100]:
```
GridSearchCV(cv=3, error_score='raise',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=None, cr
iterion='gini',
            max_depth=None, 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=10, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
       fit_params=None, iid=True, n_jobs=-1,
       param_grid={'max_features': [5, 10], 'n_estimators': [100, 200, 300],
'max_depth': [4, 8, 10], 'min_samples_split': range(200, 500, 200), 'min_samp
les_leaf': range(100, 400, 200)},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=1)
```

In [101]:
```
# printing the optimal accuracy score and hyperparameters
print('We can get accuracy of',grid_search.best_score_,'using',grid_search.bes
t_params_)
```

```
We can get accuracy of 0.818285714286 using {'max_features': 10, 'n_estimator
s': 200, 'max_depth': 8, 'min_samples_split': 200, 'min_samples_leaf': 100}
```

Fitting the final model with the best parameters obtained from grid search.

In [102]:
```
# model with the best hyperparameters
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(bootstrap=True,
                             max_depth=10,
                             min_samples_leaf=100,
                             min_samples_split=200,
                             max_features=10,
                             n_estimators=100)
```

In [103]:
```
# fit
rfc.fit(X_train,y_train)
```

Out[103]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=10, max_features=10, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=100, min_samples_split=200,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [104]:
```
# predict
predictions = rfc.predict(X_test)
```

In [105]:
```
# evaluation metrics
from sklearn.metrics import classification_report,confusion_matrix
```

In [106]:
```
print(classification_report(y_test,predictions))
```

```
             precision    recall  f1-score   support

          0       0.84      0.96      0.90      7058
          1       0.70      0.36      0.47      1942

avg / total       0.81      0.83      0.81      9000
```

In [107]:
```
print(confusion_matrix(y_test,predictions))
```

```
[[6756  302]
 [1249  693]]
```