# Decision Tree: Income Prediction

In this case study, we will build a decision tree to predict the income of a given population, which is labelled as <=$50K and >$50K. The attributes (predictors) are age, working class type, marital status, gender, race etc.

In the following sections, we'll:

- clean and prepare the data,
- build a decision tree with default hyperparameters,
- understand all the hyperparameters that we can tune, and finally
- choose the optimal hyperparameters using grid search cross-validation.

## Understanding and Cleaning the Data

In [1]:
```
# Importing the required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [2]:
```
# To ignore warnings
import warnings
warnings.filterwarnings("ignore")
```

In [3]:
```
# Reading the csv file and putting it into 'df' object.
df = pd.read_csv('adult_dataset.csv')
```

In [4]:
```python
# Let's understand the type of values in each column of our dataframe 'df'.
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32561 entries, 0 to 32560
Data columns (total 15 columns):
age               32561 non-null int64
workclass         32561 non-null object
fnlwgt            32561 non-null int64
education         32561 non-null object
education.num     32561 non-null int64
marital.status    32561 non-null object
occupation        32561 non-null object
relationship      32561 non-null object
race              32561 non-null object
sex               32561 non-null object
capital.gain      32561 non-null int64
capital.loss      32561 non-null int64
hours.per.week    32561 non-null int64
native.country    32561 non-null object
income            32561 non-null object
dtypes: int64(6), object(9)
memory usage: 3.7+ MB
```

In [5]:
```
# Let's understand the data, how it look like.
df.head(20)
```

Out[5]:

| class | fnlwgt | education | education.num | marital.status | occupation | relationship | race | sex | cap |
|---|---|---|---|---|---|---|---|---|---|
| ? | 77053 | HS-grad | 9 | Widowed | ? | Not-in-family | White | Female | |
| 'rivate | 132870 | HS-grad | 9 | Widowed | Exec-managerial | Not-in-family | White | Female | |
| ? | 186061 | Some-college | 10 | Widowed | ? | Unmarried | Black | Female | |
| 'rivate | 140359 | 7th-8th | 4 | Divorced | Machine-op-inspct | Unmarried | White | Female | |
| 'rivate | 264663 | Some-college | 10 | Separated | Prof-specialty | Own-child | White | Female | |
| 'rivate | 216864 | HS-grad | 9 | Divorced | Other-service | Unmarried | White | Female | |
| 'rivate | 150601 | 10th | 6 | Separated | Adm-clerical | Unmarried | White | Male | |
| :e-gov | 88638 | Doctorate | 16 | Never-married | Prof-specialty | Other-relative | White | Female | |
| deral-gov | 422013 | HS-grad | 9 | Divorced | Prof-specialty | Not-in-family | White | Female | |
| 'rivate | 70037 | Some-college | 10 | Never-married | Craft-repair | Unmarried | White | Male | |
| 'rivate | 172274 | Doctorate | 16 | Divorced | Prof-specialty | Unmarried | Black | Female | |
| -emp-ıot-inc | 164526 | Prof-school | 15 | Never-married | Prof-specialty | Not-in-family | White | Male | |
| 'rivate | 129177 | Bachelors | 13 | Widowed | Other-service | Not-in-family | White | Female | |
| 'rivate | 136204 | Masters | 14 | Separated | Exec-managerial | Not-in-family | White | Male | |
| ? | 172175 | Doctorate | 16 | Never-married | ? | Not-in-family | White | Male | |
| 'rivate | 45363 | Prof-school | 15 | Divorced | Prof-specialty | Not-in-family | White | Male | |
| 'rivate | 172822 | 11th | 7 | Divorced | Transport-moving | Not-in-family | White | Male | |
| 'rivate | 317847 | Masters | 14 | Divorced | Exec-managerial | Not-in-family | White | Male | |
| 'rivate | 119592 | Assoc-acdm | 12 | Never-married | Handlers-cleaners | Not-in-family | Black | Male | |
| 'rivate | 203034 | Bachelors | 13 | Separated | Sales | Not-in-family | White | Male | |

You can observe that the columns workclass and occupation consist of missing values which are represented as '?' in the dataframe.

In [6]:
```python
# rows with missing values represented as'?'.
df_1 = df[df.workclass == '?']
df_1
```

Out[6]:

| | age | workclass | fnlwgt | education | education.num | marital.status | occupation | relationship |
|---|---|---|---|---|---|---|---|---|
| **0** | 90 | ? | 77053 | HS-grad | 9 | Widowed | ? | Not-in-family |
| **2** | 66 | ? | 186061 | Some-college | 10 | Widowed | ? | Unmarried |
| **14** | 51 | ? | 172175 | Doctorate | 16 | Never-married | ? | Not-in-family |
| **24** | 61 | ? | 135285 | HS-grad | 9 | Married-civ-spouse | ? | Husband |
| **44** | 71 | ? | 100820 | HS-grad | 9 | Married-civ-spouse | ? | Husband |
| **48** | 68 | ? | 192052 | Some-college | 10 | Married-civ-spouse | ? | Wife |
| **49** | 67 | ? | 174995 | Some-college | 10 | Married-civ-spouse | ? | Husband |
| **76** | 41 | ? | 27187 | Assoc-voc | 11 | Married-civ-spouse | ? | Husband |

Now we can check the number of rows in df_1.

In [7]:
```python
df_1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1836 entries, 0 to 32544
Data columns (total 15 columns):
age               1836 non-null int64
workclass         1836 non-null object
fnlwgt            1836 non-null int64
education         1836 non-null object
education.num     1836 non-null int64
marital.status    1836 non-null object
occupation        1836 non-null object
relationship      1836 non-null object
race              1836 non-null object
sex               1836 non-null object
capital.gain      1836 non-null int64
capital.loss      1836 non-null int64
hours.per.week    1836 non-null int64
native.country    1836 non-null object
income            1836 non-null object
dtypes: int64(6), object(9)
memory usage: 229.5+ KB
```

There are 1836 rows with missing values, which is about 5% of the total data. We choose to simply drop these rows.

In [8]:
```python
# dropping the rows having missing values in workclass
df = df[df['workclass'] != '?']
df.head()
```

Out[8]:

| | age | workclass | fnlwgt | education | education.num | marital.status | occupation | relationship | rac |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 82 | Private | 132870 | HS-grad | 9 | Widowed | Exec-managerial | Not-in-family | Whi |
| **3** | 54 | Private | 140359 | 7th-8th | 4 | Divorced | Machine-op-inspct | Unmarried | Whi |
| **4** | 41 | Private | 264663 | Some-college | 10 | Separated | Prof-specialty | Own-child | Whi |
| **5** | 34 | Private | 216864 | HS-grad | 9 | Divorced | Other-service | Unmarried | Whi |
| **6** | 38 | Private | 150601 | 10th | 6 | Separated | Adm-clerical | Unmarried | Whi |

Let's see whether any other columns contain a "?". Since "?" is a string, we can apply this check only on the categorical columns.

In [9]:
```python
# select all categorical variables
df_categorical = df.select_dtypes(include=['object'])

# checking whether any other columns contain a "?"
df_categorical.apply(lambda x: x=="?", axis=0).sum()
```

Out[9]:
```
workclass           0
education           0
marital.status      0
occupation          7
relationship        0
race                0
sex                 0
native.country    556
income              0
dtype: int64
```

Thus, the columns occupation and native.country contain some "?"s. Let's get rid of them.

In [10]:
```python
# dropping the "?"s
df = df[df['occupation'] != '?']
df = df[df['native.country'] != '?']
```

Now we have a clean dataframe which is ready for model building.

In [11]:
```
# clean dataframe
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30162 entries, 1 to 32560
Data columns (total 15 columns):
age               30162 non-null int64
workclass         30162 non-null object
fnlwgt            30162 non-null int64
education         30162 non-null object
education.num     30162 non-null int64
marital.status    30162 non-null object
occupation        30162 non-null object
relationship      30162 non-null object
race              30162 non-null object
sex               30162 non-null object
capital.gain      30162 non-null int64
capital.loss      30162 non-null int64
hours.per.week    30162 non-null int64
native.country    30162 non-null object
income            30162 non-null object
dtypes: int64(6), object(9)
memory usage: 3.7+ MB
```

## Data Preparation

There are a number of preprocessing steps we need to do before building the model.

Firstly, note that we have both categorical and numeric features as predictors. In previous models such as linear and logistic regression, we had created **dummy variables** for categorical variables, since those models (being mathematical equations) can process only numeric variables.

All that is not required in decision trees, since they can process categorical variables easily. However, we still need to **encode the categorical variables** into a standard format so that sklearn can understand them and build the tree. We'll do that using the `LabelEncoder()` class, which comes with `sklearn.preprocessing`.

In [12]:
```python
from sklearn import preprocessing


# encode categorical variables using Label Encoder

# select all categorical variables
df_categorical = df.select_dtypes(include=['object'])
df_categorical.head()
```

Out[12]:

| | workclass | education | marital.status | occupation | relationship | race | sex | native.country | inc |
|---|---|---|---|---|---|---|---|---|---|
| **1** | Private | HS-grad | Widowed | Exec-managerial | Not-in-family | White | Female | United-States | < |
| **3** | Private | 7th-8th | Divorced | Machine-op-inspct | Unmarried | White | Female | United-States | < |
| **4** | Private | Some-college | Separated | Prof-specialty | Own-child | White | Female | United-States | < |
| **5** | Private | HS-grad | Divorced | Other-service | Unmarried | White | Female | United-States | < |
| **6** | Private | 10th | Separated | Adm-clerical | Unmarried | White | Male | United-States | < |

In [13]:
```python
# apply Label encoder to df_categorical

le = preprocessing.LabelEncoder()
df_categorical = df_categorical.apply(le.fit_transform)
df_categorical.head()
```

Out[13]:

| | workclass | education | marital.status | occupation | relationship | race | sex | native.country | income |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 2 | 11 | 6 | 3 | 1 | 4 | 0 | 38 | 0 |
| **3** | 2 | 5 | 0 | 6 | 4 | 4 | 0 | 38 | 0 |
| **4** | 2 | 15 | 5 | 9 | 3 | 4 | 0 | 38 | 0 |
| **5** | 2 | 11 | 0 | 7 | 4 | 4 | 0 | 38 | 0 |
| **6** | 2 | 0 | 5 | 0 | 4 | 4 | 1 | 38 | 0 |

In [14]:
```
# concat df_categorical with original df
df = df.drop(df_categorical.columns, axis=1)
df = pd.concat([df, df_categorical], axis=1)
df.head()
```

Out[14]:

| | age | fnlwgt | education.num | capital.gain | capital.loss | hours.per.week | workclass | education | ma |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 82 | 132870 | 9 | 0 | 4356 | 18 | 2 | 11 | |
| **3** | 54 | 140359 | 4 | 0 | 3900 | 40 | 2 | 5 | |
| **4** | 41 | 264663 | 10 | 0 | 3900 | 40 | 2 | 15 | |
| **5** | 34 | 216864 | 9 | 0 | 3770 | 45 | 2 | 11 | |
| **6** | 38 | 150601 | 6 | 0 | 3770 | 40 | 2 | 0 | |

In [15]:
```
# look at column types
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30162 entries, 1 to 32560
Data columns (total 15 columns):
age               30162 non-null int64
fnlwgt            30162 non-null int64
education.num     30162 non-null int64
capital.gain      30162 non-null int64
capital.loss      30162 non-null int64
hours.per.week    30162 non-null int64
workclass         30162 non-null int32
education         30162 non-null int32
marital.status    30162 non-null int32
occupation        30162 non-null int32
relationship      30162 non-null int32
race              30162 non-null int32
sex               30162 non-null int32
native.country    30162 non-null int32
income            30162 non-null int32
dtypes: int32(9), int64(6)
memory usage: 2.6 MB
```

In [16]:
```
# convert target variable income to categorical - Target variable not needed in
Decision node to Check
#df['income'] = df['income'].astype('category')
#df.head()
```

Now all the categorical variables are suitably encoded. Let's build the model.

## Model Building and Evaluation

Let's first build a decision tree with default hyperparameters. Then we'll use cross-validation to tune them.

In [17]:
```
# Importing train-test-split
from sklearn.model_selection import train_test_split
```

In [18]:
```
# Putting feature variable to X
X = df.drop('income',axis=1)

# Putting response variable to y
y = df['income']
```

In [19]:
```
# Splitting the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.30,
                                                    random_state = 99)
X_train.head()
```

Out[19]:

| | age | fnlwgt | education.num | capital.gain | capital.loss | hours.per.week | workclass | education | mari |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 42 | 289636 | 9 | 0 | 0 | 46 | 2 | 11 | |
| **6** | 37 | 52465 | 9 | 0 | 0 | 40 | 1 | 11 | |
| **7** | 38 | 125933 | 14 | 0 | 0 | 40 | 0 | 12 | |
| **2** | 44 | 183829 | 13 | 0 | 0 | 38 | 5 | 9 | |
| **3** | 35 | 198841 | 11 | 0 | 0 | 35 | 2 | 8 | |

In [20]:
```
# Importing decision tree classifier from sklearn library
from sklearn.tree import DecisionTreeClassifier

# Fitting the decision tree with default hyperparameters, apart from
# max_depth which is 5 so that we can plot and read the tree.
dt_default = DecisionTreeClassifier(max_depth=5)
dt_default.fit(X_train, y_train)
```

Out[20]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=5,
                       max_features=None, 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=None, splitter='best')
```

In [21]:
```python
# Let's check the evaluation metrics of our default model

# Importing classification report and confusion matrix from sklearn metrics
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score

# Making predictions
y_pred_default = dt_default.predict(X_test)

# Printing classification report
print(classification_report(y_test, y_pred_default))
```

```
              precision    recall  f1-score   support

           0       0.86      0.95      0.91      6867
           1       0.78      0.52      0.63      2182

    accuracy                           0.85      9049
   macro avg       0.82      0.74      0.77      9049
weighted avg       0.84      0.85      0.84      9049
```

In [22]:
```python
# Printing confusion matrix and accuracy
print(confusion_matrix(y_test,y_pred_default))
print(accuracy_score(y_test,y_pred_default))
```

```
[[6553  314]
 [1038 1144]]
0.8505912255497845
```

---

### Hyperparameter Tuning

### Tuning max_depth

Let's first try to find the optimum values for max_depth and understand how the value of max_depth affects the decision tree.

Here, we are creating a dataframe with max_depth in range 1 to 80 and checking the accuracy score corresponding to each max_depth.

To reiterate, a grid search scheme consists of:

- an estimator (classifier such as SVC() or decision tree)
- a parameter space
- a method for searching or sampling candidates (optional)
- a cross-validation scheme, and
- a score function (accuracy, roc_auc etc.)

In [55]:
```python
# GridSearchCV to find optimal max_depth
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(1, 40)}

# instantiate the model
dtree = DecisionTreeClassifier(criterion = "gini",
                               random_state = 100)

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

Out[55]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', ma
x_depth=None,
            max_features=None, 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=100,
            splitter='best'),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'max_depth': range(1, 40)}, pre_dispatch='2*n_jobs',
       refit=True, return_train_score='warn', scoring='accuracy',
       verbose=0)
```

In [56]:

```
# scores of GridSearch CV
scores = tree.cv_results_
pd.DataFrame(scores).head()
```

Out[56]:

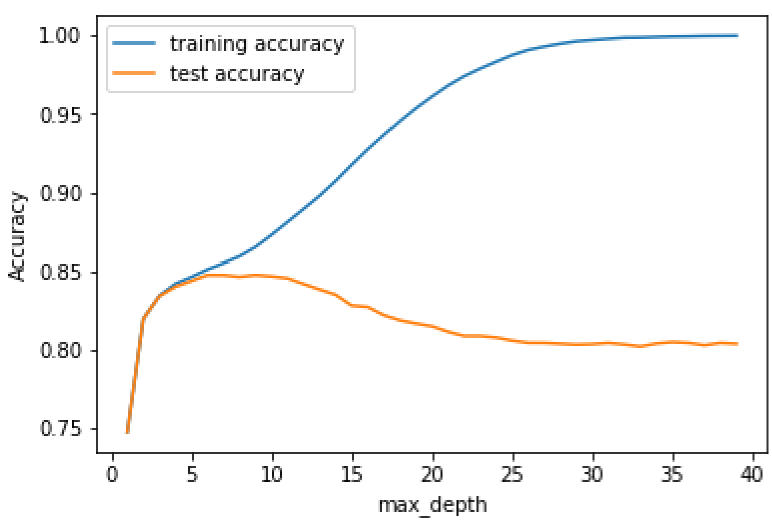
| | mean_fit_time | std_fit_time | mean_score_time | std_score_time | param_max_depth | params | s |
|---|---|---|---|---|---|---|---|
| **0** | 0.013414 | 0.006326 | 0.001799 | 0.000979 | 1 | {'max_depth': 1} | |
| **1** | 0.018751 | 0.006248 | 0.006253 | 0.007658 | 2 | {'max_depth': 2} | |
| **2** | 0.018158 | 0.003160 | 0.006247 | 0.007651 | 3 | {'max_depth': 3} | |
| **3** | 0.051170 | 0.018552 | 0.002007 | 0.004013 | 4 | {'max_depth': 4} | |
| **4** | 0.035856 | 0.007133 | 0.006250 | 0.007654 | 5 | {'max_depth': 5} | |

5 rows × 21 columns

Now let's visualize how train and test score changes with max_depth.

In [57]:

```
# 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()
```



You can see that as we increase the value of max_depth, both training and test score increase till about max-depth = 10, after which the test score gradually reduces. Note that the scores are average accuracies across the 5-folds.

Thus, it is clear that the model is overfitting the training data if the max_depth is too high. Next, let's see how the model behaves with other hyperparameters.

---

## Tuning min_samples_leaf

The hyperparameter **min_samples_leaf** indicates the minimum number of samples required to be at a leaf.

So if the values of min_samples_leaf is less, say 5, then the will be constructed even if a leaf has 5, 6 etc. observations (and is likely to overfit).

Let's see what will be the optimum value for min_samples_leaf.

In [39]:
```
# GridSearchCV to find optimal max_depth
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(5, 200, 20)}

# instantiate the model
dtree = DecisionTreeClassifier(criterion = "gini",
                               random_state = 100)

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

Out[39]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', ma
x_depth=None,
            max_features=None, 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=100,
            splitter='best'),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'min_samples_leaf': range(5, 200, 20)},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='accuracy', verbose=0)
```

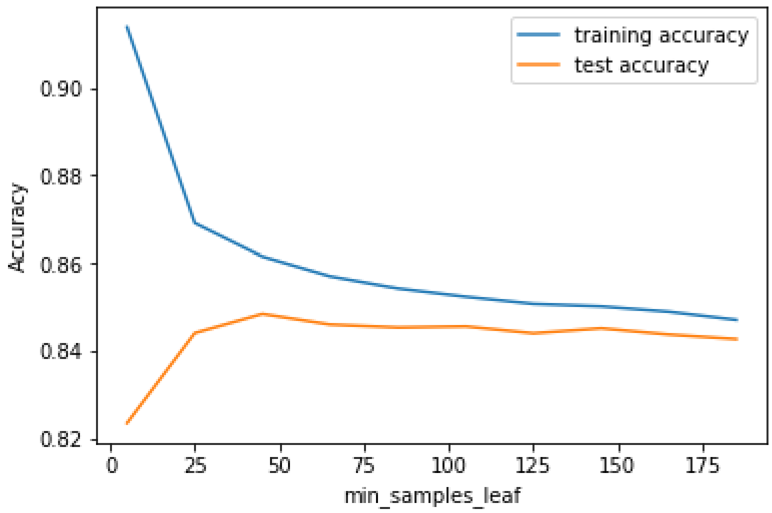
In [32]:
```
# scores of GridSearch CV
scores = tree.cv_results_
pd.DataFrame(scores).head()
```

Out[32]:

| | mean_fit_time | mean_score_time | mean_test_score | mean_train_score | param_min_samples_leaf | |
|---|---|---|---|---|---|---|
| **0** | 0.076940 | 0.002072 | 0.823663 | 0.913785 | 5 | { |
| **1** | 0.070042 | 0.002049 | 0.844172 | 0.869180 | 25 | { |
| **2** | 0.055783 | 0.001711 | 0.848529 | 0.861554 | 45 | { |
| **3** | 0.061370 | 0.002087 | 0.846114 | 0.857067 | 65 | { |
| **4** | 0.051622 | 0.002022 | 0.845451 | 0.854320 | 85 | { |

5 rows × 21 columns

In [33]:
```
# 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()
```




You can see that at low values of min_samples_leaf, the tree gets a bit overfitted. At values > 100, however, the model becomes more stable and the training and test accuracy start to converge.

---

## Tuning min_samples_split

The hyperparameter **min_samples_split** is the minimum no. of samples required to split an internal node. Its default value is 2, which means that even if a node is having 2 samples it can be furthur divided into leaf nodes.

In [40]:
```python
# 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(5, 200, 20)}

# instantiate the model
dtree = DecisionTreeClassifier(criterion = "gini",
                               random_state = 100)

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

Out[40]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', ma
x_depth=None,
            max_features=None, 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=100,
            splitter='best'),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'min_samples_split': range(5, 200, 20)},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='accuracy', verbose=0)
```

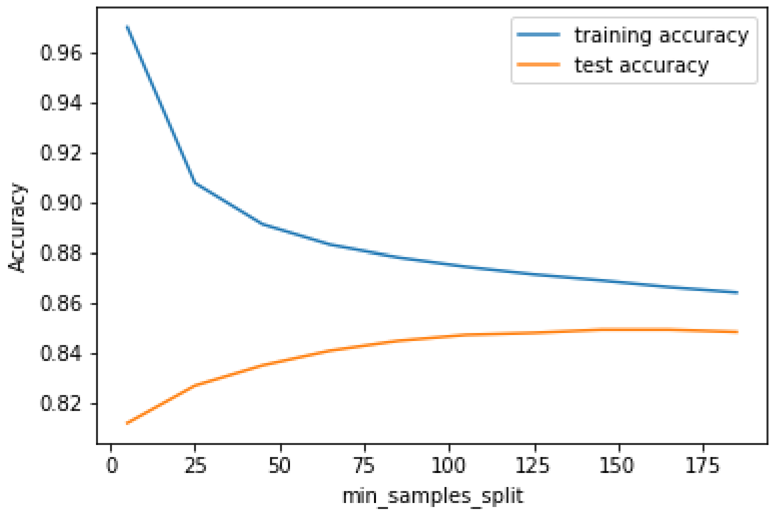
In [41]:
```python
# scores of GridSearch CV
scores = tree.cv_results_
pd.DataFrame(scores).head()
```

Out[41]:

| | mean_fit_time | std_fit_time | mean_score_time | std_score_time | param_min_samples_split | |
|---|---|---|---|---|---|---|
| **0** | 0.249302 | 0.023308 | 0.007176 | 0.001296 | 5 | {'min_sa |
| **1** | 0.254080 | 0.041291 | 0.006585 | 0.002792 | 25 | {'min_sa |
| **2** | 0.248087 | 0.022977 | 0.008883 | 0.002362 | 45 | {'min_sa |
| **3** | 0.271269 | 0.056348 | 0.008110 | 0.001872 | 65 | {'min_sa |
| **4** | 0.216440 | 0.032674 | 0.007011 | 0.002618 | 85 | {'min_sa |

5 rows × 21 columns

In [36]:
```
# plotting accuracies with min_samples_leaf
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()
```




This shows that as you increase the min_samples_split, the tree overfits lesser since the model is less complex.

---

## Grid Search to Find Optimal Hyperparameters

We can now use GridSearchCV to find multiple optimal hyperparameters together. Note that this time, we'll also specify the criterion (gini/entropy or IG).

In [58]:
```python
# Create the parameter grid
param_grid = {
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
    'criterion': ["entropy", "gini"]
}

n_folds = 5

# Instantiate the grid search model
dtree = DecisionTreeClassifier()
grid_search = GridSearchCV(estimator = dtree, param_grid = param_grid,
                          cv = n_folds, verbose = 1)

# Fit the grid search to the data
grid_search.fit(X_train,y_train)
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent worker
s.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:    4.4s finished
```

Out[58]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
       estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', ma
x_depth=None,
            max_features=None, 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=None,
            splitter='best'),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'max_depth': range(5, 15, 5), 'min_samples_leaf': range(50,
150, 50), 'min_samples_split': range(50, 150, 50), 'criterion': ['entropy', 'gi
ni']},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=1)
```

In [59]:
```
# cv results
cv_results = pd.DataFrame(grid_search.cv_results_)
cv_results
```

Out[59]:

| | mean_fit_time | std_fit_time | mean_score_time | std_score_time | param_criterion | param_max_de |
|---|---|---|---|---|---|---|
| 0 | 0.050325 | 0.008642 | 0.001601 | 0.001358 | entropy | |
| 1 | 0.032814 | 0.003100 | 0.003125 | 0.006250 | entropy | |
| 2 | 0.038284 | 0.004984 | 0.003126 | 0.006251 | entropy | |
| 3 | 0.040109 | 0.006026 | 0.000000 | 0.000000 | entropy | |

In [60]:
```
# printing the optimal accuracy score and hyperparameters
print("best accuracy", grid_search.best_score_)
print(grid_search.best_estimator_)
```

```
best accuracy 0.8514659214701843
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=10,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=50, min_samples_split=50,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

**Running the model with best parameters obtained from grid search.**

In [61]:
```
# model with optimal hyperparameters
clf_gini = DecisionTreeClassifier(criterion = "gini",
                                  random_state = 100,
                                  max_depth=10,
                                  min_samples_leaf=50,
                                  min_samples_split=50)
clf_gini.fit(X_train, y_train)
```

Out[61]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=10,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=50, min_samples_split=50,
            min_weight_fraction_leaf=0.0, presort=False, random_state=100,
            splitter='best')
```

In [62]:
```
# accuracy score
clf_gini.score(X_test,y_test)
```

Out[62]:
```
0.850922753895458
```

In [63]:
```
# plotting the tree
dot_data = StringIO()
export_graphviz(clf_gini,
out_file=dot_data,feature_names=features,filled=True,rounded=True)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-63-62884116144f> in <module>()
      1 # plotting the tree
      2 dot_data = StringIO()
----> 3 export_graphviz(clf_gini, out_file=dot_data,feature_names=features,fill
ed=True,rounded=True)
      4 
      5 graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

NameError: name 'features' is not defined
```

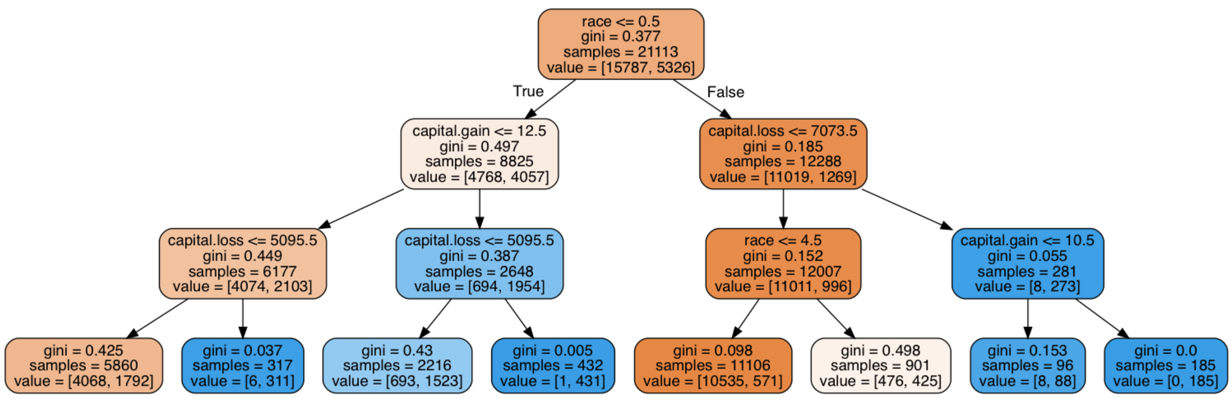
You can see that this tree is too complex to understand. Let's try reducing the max_depth and see how the tree looks.

In [64]:
```
# tree with max_depth = 3
clf_gini = DecisionTreeClassifier(criterion = "gini",
                                  random_state = 100,
                                  max_depth=3,
                                  min_samples_leaf=50,
                                  min_samples_split=50)
clf_gini.fit(X_train, y_train)

# score
print(clf_gini.score(X_test,y_test))
```

```
0.8393192617968837
```

In [44]:
```
# plotting tree with max_depth=3
dot_data = StringIO()
export_graphviz(clf_gini,
out_file=dot_data,feature_names=features,filled=True,rounded=True)

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

Out[44]:




In [45]:
```
# classification metrics
from sklearn.metrics import classification_report,confusion_matrix
y_pred = clf_gini.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
             precision    recall  f1-score   support

          0       0.85      0.96      0.90      6867
          1       0.77      0.47      0.59      2182

avg / total       0.83      0.84      0.82      9049
```

In [46]:
```
# confusion matrix
print(confusion_matrix(y_test,y_pred))
```

```
[[6564  303]
 [1151 1031]]
```