LAB8 Decision Trees and Random Forests

December 3, 2020

1 BME 455 Lab 8

Nia Sanchez, Tiffany Nguyen, Hideki Nakazawa, Xochitl Alvarado

Due: 3 December 2020

[1]:
```
#conda install python-graphviz
```

[2]:
```
import numpy as np #numerical computation
import pandas as pd #data wrangling
import matplotlib.pyplot as plt #plotting package
#Next line helps with rendering plots
%matplotlib inline
import matplotlib as mpl #add'l plotting functionality
mpl.rcParams['figure.dpi'] = 400 #high res figures
import graphviz #to visualize decision trees
```

[3]:
```
df = pd.read_csv('data.csv')
```

[4]:
```
df
```

[4]:
```
                 ID  LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  PAY_1  PAY_2  \
0      798fc410-45c1      20000    2          2         1   24      2      2
1      8a8c8f3b-8eb4     120000    2          2         2   26     -1      2
2      85698822-43f5      90000    2          2         2   34      0      0
3      0737c11b-be42      50000    2          2         1   37      0      0
4      3b7f77cc-dbc0      50000    1          2         1   57     -1      0
…                …        …  …        …         …  …    …    …
26659  ecff42d0-bdc6     220000    1          3         1   39      0      0
26660  99d1fa0e-222b     150000    1          3         2   43     -1     -1
26661  95cdd3e7-4f24      30000    1          2         2   37      4      3
26662  00d03f02-04cd      80000    1          3         1   41      1     -1
26663  15d69f9f-5ad3      50000    1          2         1   46      0      0

       PAY_3  PAY_4  …  PAY_AMT4  PAY_AMT5  PAY_AMT6  \
0         -1     -1  …         0         0         0
1          0      0  …      1000         0      2000
2          0      0  …      1000      1000      5000
```

```
3          0       0  ...      1100       1069       1000
4         -1       0  ...      9000        689        679
...      ...     ...  ...       ...        ...        ...
26659      0       0  ...      3047       5000       1000
26660     -1      -1  ...       129          0          0
26661      2      -1  ...      4200       2000       3100
26662      0       0  ...      1926      52964       1804
26663      0       0  ...      1000       1000       1000

       default payment next month  EDUCATION_CAT  graduate school  \
0                               1     university                0
1                               1     university                0
2                               0     university                0
3                               0     university                0
4                               0     university                0
...                           ...            ...              ...
26659                           0    high school                0
26660                           0    high school                0
26661                           1     university                0
26662                           1    high school                0
26663                           1     university                0

       high school  none  others  university
0                0     0       0           1
1                0     0       0           1
2                0     0       0           1
3                0     0       0           1
4                0     0       0           1
...            ...   ...     ...         ...
26659            1     0       0           0
26660            1     0       0           0
26661            0     0       0           1
26662            1     0       0           0
26663            0     0       0           1

[26664 rows x 31 columns]
```

3. Get a list of column names of the DataFrame:

[5]:
```
features_response = df.columns.tolist()
```

4. Make a list of columns to remove that aren't features or the response variable:

[6]:
```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4',␣
↪'PAY_5','PAY_6','EDUCATION_CAT',
                   'graduate school', 'high school','none','others',␣
↪'university']
```

5. Use a list comprehension to remove these column names from our list of features and the response variable:

[7]:
```
features_response = [item for item in features_response if item not in␣
 ↪items_to_remove]
features_response
```

[7]:
```
['LIMIT_BAL',
 'EDUCATION',
 'MARRIAGE',
 'AGE',
 'PAY_1',
 'BILL_AMT1',
 'BILL_AMT2',
 'BILL_AMT3',
 'BILL_AMT4',
 'BILL_AMT5',
 'BILL_AMT6',
 'PAY_AMT1',
 'PAY_AMT2',
 'PAY_AMT3',
 'PAY_AMT4',
 'PAY_AMT5',
 'PAY_AMT6',
 'default payment next month']
```

6. Run this code to make imports from scikit-learn:

[8]:
```
from sklearn.model_selection import train_test_split
from sklearn import tree
```

7. Split the data into training and testing, using the same random seed we have throughout the book for the train/test split:

[9]:
```
X_train, X_test, y_train, y_test = \
train_test_split(df[features_response[:-1]].values,
df['default payment next month'].values,
test_size=0.2, random_state=24)
```

8. Instantiate the decision tree class by specifying the max_depth parameter to be 2:

[10]:
```
dt = tree.DecisionTreeClassifier(max_depth=2)
```

9. Use this code to fit the decision tree model and grow the tree:

[11]:
```
dt.fit(X_train, y_train)
```

[11]:
```
DecisionTreeClassifier(max_depth=2)
```

10. Export the trained model in a format that can be read by the graphviz package using this code:

[12]:
```
dot_data = tree.export_graphviz(dt, out_file=None, filled=True,
rounded=True, feature_names=features_response[:-1],
proportion=True, class_names=['Not defaulted', 'Defaulted'])
```

11. Use the .Source method of the graphviz package to create an image from dot_data and display it:

[2]:
```
#graph = graphviz.Source(dot_data)
#graph
```

12. To confirm the proportion of training samples where the PAY_1 feature is less than or equal to 1.5, first identify the index of this feature in the list of features_ response[:-1] feature names:

[14]:
```
features_response[:-1].index('PAY_1')
```

[14]:
```
4
```

13. Now observe the shape of the training data:

[15]:
```
X_train.shape
```

[15]:
```
(21331, 17)
```

14. Use this code to confirm the proportion of samples after the first split of the decision tree:

[16]:
```
sum(X_train[:,4] <= 1.5)/X_train.shape[0]
```

[16]:
```
0.8946134733486475
```

15. Calculate the class fraction in the training set with this code:

[17]:
```
np.mean(y_train)
```

[17]:
```
0.223102526838873
```

This is equal to the second member of the pair of numbers following "value" in the top node; the first number is just one minus this, in other words, the fraction of negative training samples. In each subsequent node, the class fraction of the samples that are contained in just that node are displayed. The class fractions are also how the nodes are colored: those with a higher proportion of the negative class than the positive class are orange, with darker orange signifying higher proportions, while those with a higher proportion of the positive class have a similar scheme using a blue color. Finally, the line starting with "class" indicates how the decision tree will make predictions from a given node, if that node were a leaf node. Decision trees for classification make predictions by determining which leaf node a sample will be sorted in to, given the values of the features, and then predicting the class of the majority of the training samples in that leaf node. This strategy means that the class proportions in each node are the necessary information that is needed to make

a prediction.
For example, if we've made no splits and we are forced to make a prediction knowing nothing but the class fractions for the overall training data, we will simply choose the majority class. Since most people don't default, the class on the top node is "Not defaulted." However, the class fractions in the nodes of deeper levels are different, leading to different predictions. We'll discuss the training process in the following section.

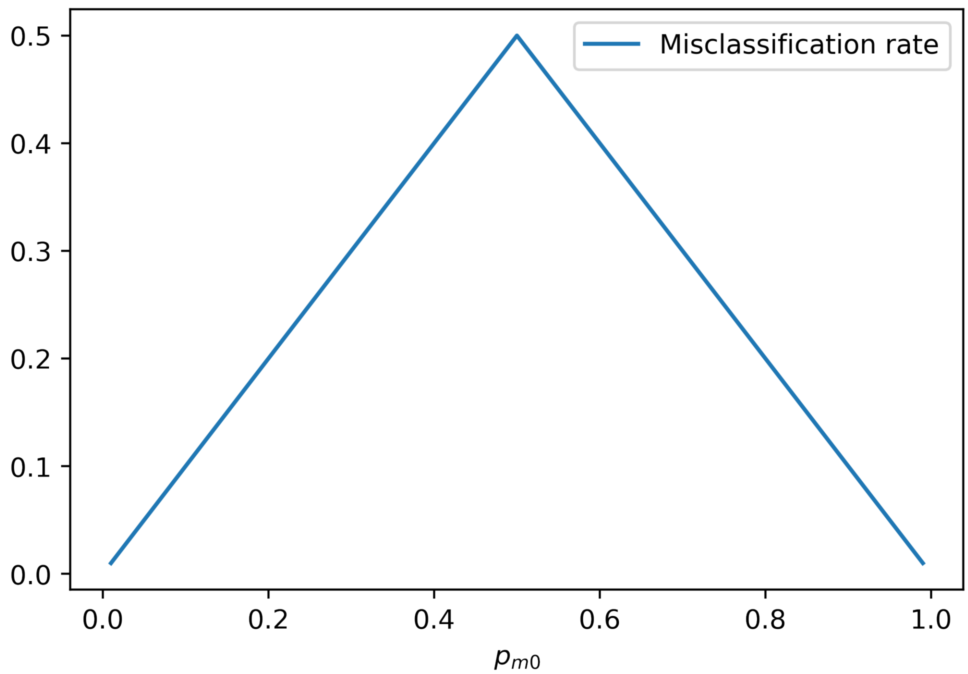
Importance of max_depth Recall that the only hyperparameter we specified in this exercise was max_depth, that is, the maximum depth to which the decision tree can be grown during the model training process. It turns out that this is one of the most important hyperparameters. Without placing a limit on the depth, the tree will be grown until one of the other limitations, specified by other hyperparameters, takes effect. This can lead to very deep trees, with very many nodes. For example, consider how many leaf nodes there could be in a tree with a depth of 20. This would be 220 leaf nodes, which is over 1 million! Do we even have 1 million training samples to sort into all these nodes? In this case, we do not. It would clearly be impossible to grow such a tree, with every node before the final level being split, using this training data. However, if we remove the max_depth limit and rerun the model training of this exercise, observe the effect:

What does the misclassification rate look like plotted against the possible class fractionsof the negative class?
We can plot this using the following code:

[18]:
```
pm0 = np.linspace(0.01,0.99,99)
pm1 = 1-pm0
misclassification_rate = np.minimum(pm0, pm1)
```

[19]:
```
mpl.rcParams['figure.dpi'] = 400
plt.plot(pm0, misclassification_rate, label='Misclassification rate')
plt.xlabel('$p_{m0}$')
plt.legend()
```

[19]:
```
<matplotlib.legend.Legend at 0x1b68454cca0>
```



Here, the summation is taken over all classes. In the case of a binary classification problem, there are only two classes, and we can write this programmatically as follows:

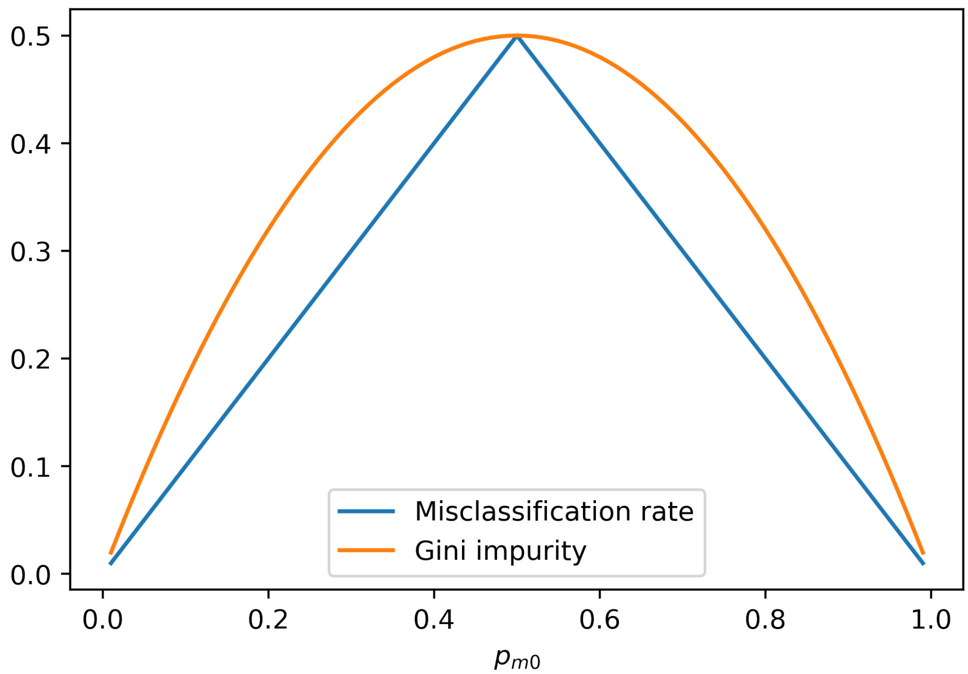
[20]:
```
gini = (pm0*(1-pm0)) + (pm1*(1-pm1))
```

[21]:
```
#Using this code, we can calculate the cross entropy:
cross_ent = -1*((pm0*np.log(pm0)) + (pm1*np.log(pm1)))
```

In order to add the Gini impurity and cross entropy to our plot of misclassification rate and see how they compare, we just need to include the following lines of code, after we plot the misclassification rate:

[22]:
```
mpl.rcParams['figure.dpi'] = 400
plt.plot(pm0, misclassification_rate, label='Misclassification rate')
plt.plot(pm0, gini, label='Gini impurity')
plt.xlabel('$p_{m0}$')
plt.legend()
```

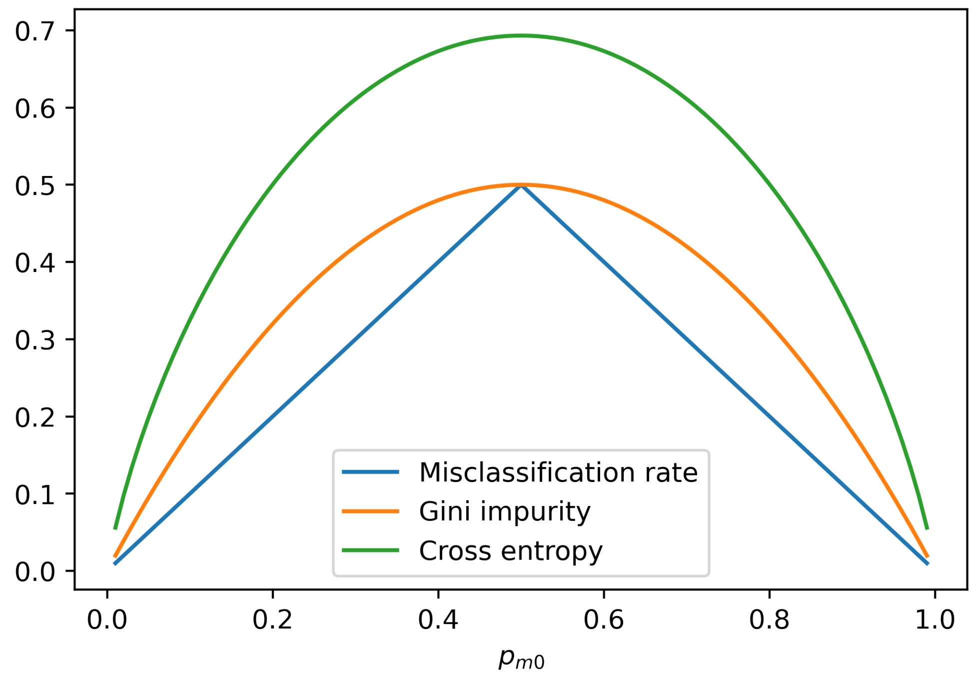
[22]:
```
<matplotlib.legend.Legend at 0x1b6845fe850>
```



[23]:
```
cross_ent = -1*( (pm0 * np.log(pm0)) + (pm1 * np.log(pm1)) )
```

[24]:
```
mpl.rcParams['figure.dpi'] = 400
plt.plot(pm0, misclassification_rate, label='Misclassification rate')
plt.plot(pm0, gini, label='Gini impurity')
plt.plot(pm0, cross_ent, label='Cross entropy')
plt.xlabel('$p_{m0}$')
plt.legend()
```

[24]:
```
<matplotlib.legend.Legend at 0x1b6886f1d90>
```



[ ]:

1.0.1 Exercise 20: Finding Optimal Hyperparameters for a Decision Tree

1. Import the GridSearchCV class with this code:

[25]:
```
from sklearn.model_selection import GridSearchCV
```

2. Find the number of samples in the training data using this code:

[26]:
```
X_train.shape
```

[26]:
```
(21331, 17)
```

3. Define a dictionary with the key being the hyperparameter name and the value being the list of values of this hyperparameter that we want to search in crossvalidation:

[27]:
```
params = {'max_depth':[1, 2, 4, 6, 8, 10, 12]}
# params = {'max_depth':list(range(1,13))}
```

4. Instantiate the GridSearchCV class using these options:

[28]:
```
cv = GridSearchCV(dt, param_grid=params, scoring='roc_auc',
                  n_jobs=None, refit=True, cv=4, verbose=1,
                  pre_dispatch=None, error_score=np.nan,␣
↪return_train_score=True)
```

```
#5/2020: removed arguments fit_params, iid
```

5. Perform a 4-fold cross-validation to search for the optimal maximum depth using this code:

[29]:
```
cv.fit(X_train, y_train)
```

```
Fitting 4 folds for each of 7 candidates, totalling 28 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  28 out of  28 | elapsed:    2.4s finished
```

[29]:
```
GridSearchCV(cv=4, estimator=DecisionTreeClassifier(max_depth=2),
             param_grid={'max_depth': [1, 2, 4, 6, 8, 10, 12]},
             pre_dispatch=None, return_train_score=True, scoring='roc_auc',
             verbose=1)
```

6. Run the following code to create and examine a pandas DataFrame of crossvalidation results:

[30]:
```
cv_results_df = pd.DataFrame(cv.cv_results_)
```

[31]:
```
cv_results_df
```

[31]:
```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       0.015962      0.000719         0.002263        0.000466
1       0.028673      0.001296         0.002246        0.000431
2       0.055351      0.001798         0.002246        0.000434
3       0.084524      0.003680         0.002744        0.001304
4       0.106000      0.000818         0.002212        0.000372
5       0.128922      0.003708         0.002236        0.000428
6       0.149102      0.001516         0.002498        0.000510

  param_max_depth             params  split0_test_score  split1_test_score  \
0               1   {'max_depth': 1}           0.639514           0.643398
1               2   {'max_depth': 2}           0.695134           0.699022
2               4   {'max_depth': 4}           0.732720           0.740116
3               6   {'max_depth': 6}           0.744848           0.745998
4               8   {'max_depth': 8}           0.727836           0.733746
5              10  {'max_depth': 10}           0.712782           0.705396
6              12  {'max_depth': 12}           0.678757           0.658568

   split2_test_score  split3_test_score  mean_test_score  std_test_score  \
0           0.651891           0.650753         0.646389        0.005136
1           0.713376           0.699510         0.701761        0.006917
2           0.746946           0.743731         0.740878        0.005294
3           0.751230           0.740774         0.745713        0.003730
4           0.750407           0.732739         0.736182        0.008512
5           0.718382           0.712559         0.712280        0.004608
6           0.675196           0.676860         0.672345        0.008054
```

```
   rank_test_score  split0_train_score  split1_train_score  \
0                7            0.648680            0.647384
1                5            0.704034            0.702700
2                2            0.756882            0.752256
3                1            0.782202            0.780125
4                3            0.812061            0.808296
5                4            0.848838            0.854234
6                6            0.887832            0.903150

   split2_train_score  split3_train_score  mean_train_score  std_train_score
0            0.644553            0.644934          0.646388         0.001712
1            0.698113            0.702535          0.701845         0.002232
2            0.749368            0.753055          0.752890         0.002682
3            0.775228            0.774750          0.778076         0.003178
4            0.803554            0.802370          0.806570         0.003869
5            0.841254            0.836752          0.845270         0.006741
6            0.885257            0.876149          0.888097         0.009715
```

7. View the names of the remaining columns in the results DataFrame using this code:

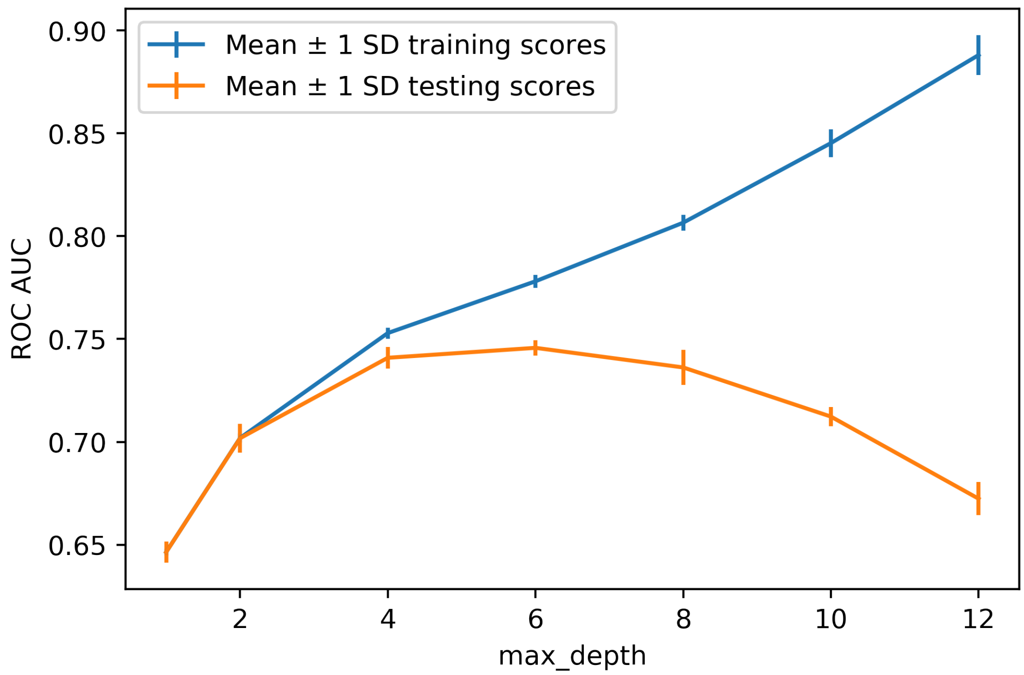
[32]:
```
cv_results_df.columns
```

[32]:
```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_max_depth', 'params', 'split0_test_score', 'split1_test_score',
       'split2_test_score', 'split3_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score', 'split0_train_score',
       'split1_train_score', 'split2_train_score', 'split3_train_score',
       'mean_train_score', 'std_train_score'],
      dtype='object')
```

8. Execute the following code to create an error bar plot of training and testing scores for each value of max_depth that was examined in cross-validation:

[33]:
```
ax = plt.axes()
ax.errorbar(cv_results_df['param_max_depth'],
            cv_results_df['mean_train_score'],
            yerr=cv_results_df['std_train_score'],
            label='Mean $\pm$ 1 SD training scores')
ax.errorbar(cv_results_df['param_max_depth'],
            cv_results_df['mean_test_score'],
            yerr=cv_results_df['std_test_score'],
            label='Mean $\pm$ 1 SD testing scores')
ax.legend()
plt.xlabel('max_depth')
plt.ylabel('ROC AUC')
```

[33]:
```
Text(0, 0.5, 'ROC AUC')
```



[34]:
```
cv_results_df.max()
```

[34]:
```
mean_fit_time          0.149102
std_fit_time           0.003708
mean_score_time        0.002744
std_score_time         0.001304
param_max_depth       12.000000
split0_test_score      0.744848
split1_test_score      0.745998
split2_test_score      0.751230
split3_test_score      0.743731
mean_test_score        0.745713
std_test_score         0.008512
rank_test_score        7.000000
split0_train_score     0.887832
split1_train_score     0.903150
split2_train_score     0.885257
split3_train_score     0.876149
mean_train_score       0.888097
std_train_score        0.009715
dtype: float64
```

1.0.2 Exercise 21: Fitting a Random Forest

1. Import the random forest classifier model class as follows:

[35]:
```python
from sklearn.ensemble import RandomForestClassifier
```

2. Instantiate the class using these options:

[36]:
```python
rf = RandomForestClassifier(
n_estimators=10, criterion='gini', max_depth=3,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,
max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None,
random_state=4, verbose=0, warm_start=False, class_weight=None)
```

3. Create a parameter grid for this exercise in order to search the numbers of trees ranging from 10 to 100:

[37]:
```python
rf_params_ex = {'n_estimators':list(range(10,110,10))}
```

4. Instantiate a grid search cross-validation object for the random forest model using the parameter grid from the previous step. Otherwise, you can use the same options that were used for the cross-validation of the decision tree:

[38]:
```python
cv_rf_ex = GridSearchCV(rf, param_grid=rf_params_ex, scoring='roc_auc',
                        n_jobs=None, refit=True, cv=4, verbose=1,
                        pre_dispatch=None, error_score=np.nan,␣
 ↪return_train_score=True)
#5/2020: removed arguments fit_params, iid
```

5. Fit the cross-validation object as follows:

[ ]:
```python
cv_rf_ex.fit(X_train, y_train)
```

```
Fitting 4 folds for each of 10 candidates, totalling 40 fits
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
```

6. Put the cross-validation results into a pandas DataFrame:

[ ]:
```python
cv_rf_ex_results_df = pd.DataFrame(cv_rf_ex.cv_results_)
```

7. Create two subplots, of the mean time and mean testing scores with standard deviation:

[ ]:
```python
cv_rf_ex_results_df
```

[ ]:
```python
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))
axs[0].plot(cv_rf_ex_results_df['param_n_estimators'],
            cv_rf_ex_results_df['mean_fit_time'],
            '-o')
axs[0].set_xlabel('Number of trees')
axs[0].set_ylabel('Mean fit time (seconds)')
axs[1].errorbar(cv_rf_ex_results_df['param_n_estimators'],
                cv_rf_ex_results_df['mean_test_score'],
```

```
                yerr=cv_rf_ex_results_df['std_test_score'])
axs[1].set_xlabel('Number of trees')
axs[1].set_ylabel('Mean testing ROC AUC $\pm$ 1 SD ')
plt.tight_layout()
```

8. Use this code to see the best hyperparameters from cross-validation:

[ ]:
```
cv_rf_ex.best_params_
```

9. Run this code to create a DataFrame of the feature names and importance, and then show it sorted by importance:

[ ]:
```
feat_imp_df = pd.DataFrame({
    'Feature name':features_response[:-1],
    'Importance':cv_rf_ex.best_estimator_.feature_importances_
})
```

[ ]:
```
feat_imp_df.sort_values('Importance', ascending=False)
```

1.0.3 Checkboard Graph

[ ]:
```
xx_example, yy_example = np.meshgrid(range(5), range(5))
print(xx_example)
print(yy_example)
```

[ ]:
```
z_example = np.arange(1,17).reshape(4,4)
z_example
```

[ ]:
```
ax = plt.axes()
pcolor_ex = ax.pcolormesh(xx_example, yy_example, z_example, cmap=plt.cm.jet)
plt.colorbar(pcolor_ex, label='Color scale')
ax.set_xlabel('X coordinate')
ax.set_ylabel('Y coordinate')
```

[ ]:
```
```