Perspective HW5

March 1, 2020

Yuxin Wu (yuxinwu@uchicago.edu)

Conceptual: Cost functions for classification trees

Consider the Gini index, classification error, and cross-entropy in simple classification settings with two classes. Of these three possible cost functions, which would be best to use when growing a decision tree? Which would be best to use when pruning a decision tree? Why?

We have three options for minimizing error Gini index, classification error, and cross-entropy. When building a classification tree, either the Gini index or the cross-entropy are more commonly used to evaluate the quality of a particular split. The main reason is that these two approaches are sensitive to node purity. While the classification error rate is not sensitive to node purity. Gini index or the cross-entropy tend to grow more accurate trees. Any of these three approaches might be used when pruning the tree, but the classification error rate is preferable if prediction accuracy of the final pruned tree is the goal. As classification error rate is the classification error rate, or the proportion of the training observations in a given region that do not belong to the most common classTherefore. Classification error rate prioritizes accuracy. Either Gini index or cross-entrophy may be preferred when building a tree ans classification error rate may be preferred when pruning a decision tree.

Application: Predicting attitudes towards racist college professors

[20]:
```
import pandas as pd
import numpy as np
import warnings
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import ElasticNet
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from sklearn import metrics
from sklearn.inspection import plot_partial_dependence
```

```python
import random
```

[21]:
```python
random.seed(123)
```

[23]:
```python
warnings.filterwarnings('ignore')
```

[24]:
```python
gss_train = pd.read_csv("gss_train.csv")
```

[25]:
```python
gss_test = pd.read_csv("gss_test.csv")
```

[26]:
```python
gss_test
```

[26]:
```
     age  attend  authoritarianism  black  born  childs  colath  colrac  \
0     22       2                 1      0     0       0       0       1
1     49       0                 4      0     0       2       0       0
2     50       0                 3      0     0       2       1       1
3     55       4                 3      0     1       6       1       1
4     22       3                 4      0     0       0       1       1
..   …     …                 …    …   …     …     …     …
488   49       2                 6      1     0       1       1       1
489   71       7                 5      0     0       6       1       1
490   62       4                 1      0     0       3       0       0
491   28       4                 4      0     0       0       0       0
492   74       1                 2      0     0       2       1       1

     colcom  colmil  …  partyid_3_Ind  partyid_3_Rep  relig_CATHOLIC  \
0         1       1  …              1              0               1
1         1       0  …              0              1               0
2         0       1  …              0              0               0
3         1       1  …              0              0               1
4         0       1  …              0              1               0
..      …     …  …            …            …             …
488       0       1  …              0              0               0
489       1       1  …              0              1               0
490       1       1  …              1              0               0
491       1       0  …              0              1               1
492       0       0  …              1              0               0

     relig_NONE  relig_other  social_cons3_Mod  social_cons3_Conserv  \
0             0            0                 0                     0
1             0            0                 0                     0
2             0            0                 0                     0
3             0            0                 1                     0
4             0            0                 1                     0
..          …          …               …                   …
488           0            0                 0                     1
489           0            1                 0                     1
```

```
490           0            1                0                     1
491           0            0                1                     0
492           0            0                1                     0

     spend3_Mod  spend3_Liberal  zodiac_other
0             0               0             1
1             0               0             1
2             0               1             1
3             1               0             1
4             1               0             1
..          …              …            …
488           1               0             1
489           1               0             1
490           0               0             1
491           0               1             1
492           1               0             1

[493 rows x 56 columns]
```

[27]:
```
x_train = gss_train.drop("colrac", axis = 1)
y_train = gss_train["colrac"]
x_test = gss_test.drop("colrac", axis = 1)
y_test = gss_test["colrac"]
```

Estimate the models

[28]:
```
#logistic Regression
clf = LogisticRegression()
grid_values = {'penalty': ['l1', 'l2'],'C':np.logspace(-1, 1, 20)}
lg_grid_clf_acc = GridSearchCV(clf, param_grid = grid_values, cv = 10, scoring␣
 ↪= 'accuracy')
lg_grid_clf_acc.fit(x_train, y_train)
print(lg_grid_clf_acc.best_score_)
print(lg_grid_clf_acc.best_estimator_)
y_pred = lg_grid_clf_acc.predict(x_test)
print("----------------------------------")
print("Predicted colrac")
print(y_pred)
```

```
0.7981338481338481
LogisticRegression(C=0.33598182862837817, class_weight=None, dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=100, multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
---------------------------------
Predicted colrac
[1 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1
```

```
0 1 1 0 1 0 1 1 1 0 0 0 1 1 0 1 0 0 1 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 1 1 1
0 1 0 0 1 0 1 1 0 1 1 1 0 1 0 0 1 0 0 1 0 1 1 1 0 0 1 0 0 0 0 1 1 1 0 1 0
0 1 1 0 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 0 1 1
0 1 1 1 0 0 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 0 1 1 1 1 1 0
1 1 1 1 0 1 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0 0 1 1 0
0 1 0 0 1 0 1 1 0 0 1 1 0 0 1 1 1 0 0 1 0 0 1 1 0 0 0 1 0 0 1 0 0 1 1 0 0
1 0 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 0 0 1 0 1 0 0 1 0 1 1 0 0 1 0 1 1 1 0 0 1 0 0 1 0 0 1 1 0
0 1 0 1 1 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 0 1 0 0 0 1 1 1 0 1 0 0 1 1 1 0 1
1 0 0 0 1 0 0 0 0 1 0 1 0 1 1 0 1 0 0 1 0 0 1 1 0 1 1 1 1 1 0 1 1 0 0 0 1
0 0 1 0 1 1 0 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 1 1 0 0 1 1 0 1 1 1 1 0 1 0
1 0 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 0 1 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1
1 1 1 0 1 0 1 1 1 1 1 1]
```

[29]:
```python
#Naive Bayes
clf = GaussianNB()
grid_values = {}
nb_grid_clf_acc = GridSearchCV(clf, param_grid = grid_values, cv = 10, scoring
 = 'accuracy')
nb_grid_clf_acc.fit(x_train, y_train)
print(nb_grid_clf_acc.best_score_)
print(nb_grid_clf_acc.best_estimator_)
y_pred = nb_grid_clf_acc.predict(x_test)
print("---------------------------------")
print("Predicted colrac")
print(y_pred)
```

```
0.7344226879941166
GaussianNB(priors=None, var_smoothing=1e-09)
---------------------------------
Predicted colrac
[0 0 1 1 1 1 1 1 0 0 0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 0 0 1 0 1 1 0 1 0 0 0 0
 0 0 1 0 1 0 0 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 0 0 0 1 1 1 0 1 0 1 0 0 1 1 1
 0 0 0 0 0 0 1 1 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 1 0 0 1 0 0 0 0 1 1 1 0 0 0
 0 1 1 1 0 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 0 1 0 0 0 1 1
 0 1 0 1 0 0 0 1 0 0 0 1 1 0 0 0 1 0 1 0 0 0 0 1 0 1 1 0 0 0 1 1 1 1 0 1 0
 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 0 1 0 0 1 0 1 1 0 0 1 1 0
 0 1 0 0 1 0 1 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0
 1 0 1 1 0 0 0 0 0 1 1 0 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 0 0 1 0 1 0 0 1 0 1 1 0 0 1 0 1 1 1 0 1 1 1 0 1 0 0 1 0 0
 0 1 0 0 1 1 1 1 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0 0 1 1 1 0 1 0 0 1 1 1 0 1
 1 0 0 0 1 0 0 1 0 1 0 1 0 1 1 0 1 1 0 1 0 0 0 1 0 1 1 1 1 1 0 1 1 0 0 0 0
 0 0 0 0 1 1 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 0
 1 0 0 1 1 0 1 0 0 1 1 0 1 1 1 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1
 1 1 1 0 0 1 1 1 1 1 0 1]
```

```
[30]: #Elastic Net
clf = ElasticNet()
grid_values = {"max_iter": [1, 5, 10],
               "alpha": [0.0001, 0.001, 0.01, 0.1, 1, 10, 100],
               "l1_ratio": np.arange(0.0, 1.0, 0.1)}
en_grid_clf_acc = GridSearchCV(clf, param_grid = grid_values, cv = 10, scoring␣
 ↪= 'neg_mean_squared_error')
en_grid_clf_acc.fit(x_train, y_train)
print(en_grid_clf_acc.best_score_)
print(en_grid_clf_acc.best_estimator_)
y_pred = en_grid_clf_acc.predict(x_test)
print("---------------------------------")
print("Predicted colrac")
print(y_pred)
```

```
-0.14734617233752556
ElasticNet(alpha=0.01, copy_X=True, fit_intercept=True, l1_ratio=0.2,
           max_iter=10, normalize=False, positive=False, precompute=False,
           random_state=None, selection='cyclic', tol=0.0001, warm_start=False)
---------------------------------
Predicted colrac
[ 5.96517634e-01  1.00413031e-01  1.12832740e+00  1.02325753e+00
  1.28452713e+00  6.22718328e-01  7.48683388e-01  8.47856997e-01
  2.85995796e-01  2.45984203e-01  6.24493474e-01  7.38607287e-01
  1.95747881e-01  7.71360811e-01  6.49951235e-01  9.03467898e-01
  4.71265877e-01  1.02543220e+00  1.60063384e-01  4.94172991e-01
  2.17134377e-01  7.84397234e-01  1.22301581e+00  6.06396262e-01
 -5.98491105e-04  5.24830998e-01  3.95633055e-01  9.40676930e-01
  5.23946268e-01  1.11429947e+00  5.66080827e-01  3.42877292e-01
  5.67444326e-01  5.29057396e-01  2.57977896e-02  5.43858048e-01
  8.42443172e-01  8.11825858e-03  7.77369221e-01  1.17819740e+00
  3.00551904e-01  8.31494202e-01  5.21163949e-01  8.03618820e-01
  5.03394660e-01  6.39146236e-01  2.77816400e-01  4.50506504e-01
  1.06525576e-01  6.59588282e-01  1.00165052e+00  5.32657768e-01
  6.31908443e-01  4.79232439e-01  7.52101618e-02  8.62211432e-01
  5.62489523e-01  1.24143739e+00  1.11356177e+00  1.96608721e-01
  6.32936248e-01  6.03844305e-01  8.46895503e-01  8.31641743e-01
  6.96243898e-01 -3.37279518e-02  7.28064967e-01  2.23805187e-01
  2.48523375e-01  2.97733947e-01  4.17374481e-01  8.33290784e-01
  6.77239097e-01  1.13898122e+00  2.04163811e-01  6.73022249e-01
  1.52543523e-01  8.61583366e-02  6.46584105e-01  3.67608167e-01
  8.51096759e-01  9.09954708e-01  1.77080810e-01  7.09830475e-01
  5.86339272e-01  6.88240994e-01  1.96933814e-01  8.69699949e-01
  4.49695177e-01  1.30844872e-01  5.58639173e-01  9.96459528e-02
  2.24193688e-01  7.58973061e-01  1.59259695e-01  5.94107168e-01
  9.83903528e-01  8.47455392e-01  1.53794225e-01  2.56995259e-01
  8.25467518e-01  1.93879698e-01  1.75643600e-01  3.65555971e-01
```

9.03021526e-02	6.54748621e-01	8.02105374e-01	1.01646531e+00
1.28320996e-01	9.12756199e-01	9.44387515e-02	2.09045826e-01
1.38234219e+00	5.79634085e-01	5.69321203e-01	2.30028848e-01
8.06374821e-01	1.28094406e-01	7.73593083e-01	1.23576174e+00
2.10965809e-01	7.12021909e-01	2.79517863e-01	6.22132526e-01
5.48575148e-01	6.70582067e-01	1.02452523e+00	5.58960164e-01
9.64522136e-01	7.53424110e-01	8.57130487e-01	9.86938974e-01
3.65110898e-01	1.61579202e-01	1.29284153e+00	3.62229507e-02
-8.35671771e-02	8.46796274e-01	7.71514088e-01	8.78723706e-01
1.83106685e-01	2.92839740e-01	9.31580686e-01	4.06913114e-01
5.16531367e-01	2.47393193e-01	1.20432275e+00	8.27064345e-01
1.87452911e-01	9.05239862e-01	8.58459328e-01	7.52315809e-01
2.92273844e-01	1.97969517e-01	2.90115842e-01	9.61430623e-01
1.11623885e-01	5.95552136e-02	6.39830844e-01	9.73849834e-01
9.68613917e-01	4.65147271e-02	6.53883190e-01	2.31689098e-01
1.19601728e+00	5.30573608e-01	9.70199193e-01	2.40229906e-01
2.17208638e-01	8.85349025e-01	1.42996283e-01	9.58545956e-01
2.01255613e-01	8.96699414e-01	8.62051598e-01	5.52095616e-01
1.59272837e-01	2.72735953e-02	4.58892969e-01	8.13334997e-01
8.52768359e-01	7.94515921e-01	5.78774251e-01	7.07740561e-01
9.56514532e-02	1.11875440e+00	1.08665372e+00	6.91338199e-01
5.58343212e-01	3.67252394e-01	8.28801517e-01	7.82038674e-01
6.85255203e-01	1.81254590e-01	1.93936392e-02	6.83441721e-01
3.57397846e-01	2.59535582e-01	1.07779150e-01	6.30876942e-01
6.31448313e-01	6.95000143e-01	2.61653844e-01	3.43224673e-01
-9.65593500e-02	2.74464352e-01	-1.72184061e-02	7.20642187e-01
4.82495439e-01	1.62879021e-01	8.25341718e-01	-5.27230157e-02
1.31487414e-01	8.45399336e-01	5.37228840e-01	5.04066771e-01
1.04026237e+00	4.68935818e-01	3.56853867e-04	6.77898982e-01
5.75097634e-01	2.54049019e-01	3.98954281e-01	7.19253873e-01
4.49784802e-01	3.14418094e-01	7.31340469e-01	7.98603439e-02
7.92892051e-01	6.72529717e-01	6.77914131e-02	1.18224434e-01
7.27009437e-01	4.70629470e-01	9.25673088e-03	8.98856613e-02
9.99630224e-01	6.11269268e-01	9.62056116e-01	5.27621700e-01
1.89683497e-01	7.22821495e-01	6.89379654e-02	4.24940995e-02
7.77810955e-01	8.47198883e-01	1.71138006e-01	4.94433189e-01
5.35476418e-01	9.88668995e-01	5.21883865e-01	3.25220487e-01
5.46445734e-01	2.89432429e-01	3.40677087e-01	5.49040412e-01
7.18699507e-01	1.51994967e-01	4.72058373e-01	8.04603211e-01
1.56107910e-01	6.56839090e-01	1.23400002e+00	4.59978593e-01
8.46445861e-01	4.98508185e-01	1.60047305e-01	2.44835317e-01
5.12445141e-01	8.61908960e-01	6.15319706e-01	2.24853877e-01
7.93698116e-01	4.86502975e-01	6.30947908e-01	1.66116052e-02
9.04396604e-01	9.04277719e-01	7.08916388e-01	3.58139892e-01
8.30216866e-01	7.30314924e-01	4.85670669e-01	8.72963721e-01
6.16163662e-01	8.95969646e-01	7.59862465e-01	-1.46546578e-01
1.11745954e+00	7.13764268e-01	8.26076989e-01	1.10607031e+00
6.25546758e-01	7.01802430e-01	9.95853492e-01	7.45377578e-01

6.31912690e-01	7.46338289e-01	8.84007069e-01	8.94241741e-01
6.39293684e-01	1.10399804e+00	7.05470202e-01	6.15282210e-01
3.62561831e-01	1.17181801e-01	7.53829445e-01	3.37223851e-01
6.06892536e-01	1.29251662e-01	-8.99908659e-02	8.65755336e-01
3.64615401e-01	7.41996479e-01	1.27646567e+00	2.04012136e-01
2.70916986e-01	8.66441525e-01	3.82954286e-01	8.82582990e-01
1.01354273e+00	8.28653362e-01	2.94704738e-01	2.99277674e-01
9.15323842e-01	2.58729956e-01	1.39725288e-01	9.39274297e-01
3.09611022e-01	1.14312221e-01	8.76110076e-01	9.10833265e-01
2.26898212e-03	6.35762665e-02	1.14208685e+00	1.86025331e-02
5.60559993e-01	8.81589844e-01	3.23144944e-01	7.68111770e-01
7.44687232e-01	6.18809954e-02	2.81437222e-01	9.41039750e-01
5.29002318e-01	3.23681071e-01	4.93319279e-01	4.73029398e-01
8.97360382e-01	8.20955083e-01	2.51898129e-01	2.53765062e-01
9.36241717e-02	3.02020942e-01	5.39291178e-01	3.63795453e-01
3.70523613e-01	1.03717169e-01	6.24581773e-01	6.64016407e-01
1.02074462e+00	8.09680449e-02	6.74023795e-01	2.63925795e-01
4.50443014e-01	8.37466641e-01	9.35731642e-01	9.36120862e-01
1.26208669e-01	8.70682032e-01	7.07410012e-01	6.37651409e-02
6.99115245e-02	1.90729333e-01	9.26022853e-01	7.88439747e-02
9.35579793e-02	4.76720195e-01	1.44354256e-01	5.07593573e-01
2.92123299e-01	1.15779021e+00	6.29806135e-02	8.98649089e-01
8.25646781e-01	2.52099622e-01	9.80597717e-01	4.01566810e-01
1.98937142e-01	6.10575159e-01	-5.52859467e-03	5.26219225e-02
7.85644624e-01	1.23616330e+00	1.71330588e-01	7.37013553e-01
5.48627893e-01	1.05930294e+00	9.82207687e-01	1.08459508e+00
3.27371585e-01	8.74349973e-01	5.89511438e-01	4.11521397e-01
4.70454839e-01	7.93991829e-02	8.30288129e-01	5.12536145e-01
4.49583145e-01	5.18907667e-01	2.24493747e-01	5.69200983e-01
1.08822839e+00	3.13067743e-01	6.20934399e-01	1.04411864e+00
7.26936932e-01	5.17344045e-01	8.47783691e-01	1.50870014e-01
1.18137476e+00	6.82506262e-01	1.00849499e+00	6.37173650e-01
5.41684802e-01	8.57032081e-01	3.22153938e-01	6.70248875e-01
1.02456439e+00	9.59884349e-01	4.31645977e-01	6.68510469e-01
4.49090018e-01	1.89291510e-01	8.37802267e-01	7.10442456e-01
1.44733922e-01	4.73053987e-01	9.57023260e-01	8.60367066e-01
5.63726509e-01	-1.78585472e-02	9.02697907e-01	6.76060709e-02
7.97715509e-01	-8.75576798e-02	2.22831874e-02	3.49759712e-01
1.00611020e+00	8.88444097e-02	5.79390271e-01	8.19501993e-01
2.60935971e-01	8.68560220e-01	9.59961180e-01	5.35706349e-01
8.62114930e-01	8.97448868e-01	1.20227014e+00	4.22760845e-01
1.11802505e+00	5.36660449e-01	5.56369930e-01	4.44480742e-01
5.31903539e-01	5.72687032e-01	5.75986662e-01	4.58492213e-01
8.57249655e-01	8.50452702e-01	1.02839402e+00	1.09555347e+00
4.19708085e-01	9.16130087e-01	6.72339799e-01	6.95790171e-01
2.88706598e-01	4.39498084e-01	1.19254973e+00	8.70137193e-01
7.31687135e-01	7.54476977e-01	5.37682129e-01	6.27550666e-01
2.00916750e-01	5.87895094e-01	4.80856809e-01	9.18230842e-01

```
 1.00329523e+00  9.66569857e-01  5.23266185e-01  5.03324766e-01
 7.55416162e-01]
```

[31]:

```
#Descision Tree
clf = DecisionTreeClassifier()
grid_values = {"min_samples_split": [0.1, 0.5, 1, 1.5, 2],
               "min_samples_leaf": [0.1, 0.5, 1, 1.5, 2],
               'max_depth': np.arange(3, 10)}
dt_grid_clf_acc = GridSearchCV(clf, param_grid = grid_values, cv = 10, scoring␣
 ↪= 'accuracy')
dt_grid_clf_acc.fit(x_train, y_train)
print(dt_grid_clf_acc.best_score_)
print(dt_grid_clf_acc.best_estimator_)
y_pred = dt_grid_clf_acc.predict(x_test)
print("---------------------------------")
print("Predicted colrac")
print(y_pred)
```

```
0.7797710976282405
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=4, 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='deprecated',
                       random_state=None, splitter='best')
---------------------------------
Predicted colrac
[0 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 0 0 1 0 1 1
 0 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 1 0 0 0 1 1 1 1
 0 1 0 0 1 1 1 1 0 1 1 1 0 1 1 0 1 0 0 1 0 1 1 1 0 0 1 0 0 0 0 1 1 1 0 1 0
 0 1 1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 0 0 1 0 1 0 1 1
 0 1 1 1 0 0 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0
 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 1 0 0 0 1 1 0 1 0 0 1 1 1 1 1 0 1 1 0
 0 1 1 0 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0 1 0 0 1 1 0 1 1 1 1 0 1 0 0 1 1 0 1
 1 0 1 1 1 1 1 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
 1 1 1 1 0 1 1 1 1 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 1 1 0 1 1 1 0 1 0 0 1 1 0
 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 0 1 0 1 0 1 0 1 1 1 1 0 1
 1 0 0 0 1 0 0 1 0 1 0 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 1 1 1 1 0 1 1 0 1 0 1
 1 1 0 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 0
 1 0 0 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 1 1 1 1 1 1 0 1]
```

[32]:

```
#Bagging
clf = BaggingClassifier()
grid_values = {'base_estimator': [DecisionTreeClassifier()],
               'base_estimator__max_depth' : [1, 2, 3, 4, 5],
               'max_samples' : [0.05, 0.1, 0.2, 0.5]}
```

```python
bg_grid_clf_acc = GridSearchCV(clf, param_grid = grid_values, cv = 10, scoring␣
 ↪= 'accuracy')
bg_grid_clf_acc.fit(x_train, y_train)
print(bg_grid_clf_acc.best_score_)
print(bg_grid_clf_acc.best_estimator_)
y_pred = bg_grid_clf_acc.predict(x_test)
print("---------------------------------")
print("Predicted colrac")
print(y_pred)
```

```
0.7899338113623828
BaggingClassifier(base_estimator=DecisionTreeClassifier(ccp_alpha=0.0,
                                                        class_weight=None,
                                                        criterion='gini',
                                                        max_depth=4,
                                                        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='deprecated',
                                                        random_state=None,
                                                        splitter='best'),
                  bootstrap=True, bootstrap_features=False, max_features=1.0,
                  max_samples=0.1, n_estimators=10, n_jobs=None,
                  oob_score=False, random_state=None, verbose=0,
                  warm_start=False)
---------------------------------
Predicted colrac
[0 0 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 0 0 1
 0 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 1 0 0 0 1 1 1 1
 0 1 0 0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 0 1 0 1 1 1 0 0 1 0 0 1 0 1 1 1 0 1 0
 0 1 1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 0 0 1 0 0 1 1 1
 0 1 1 1 0 0 0 1 0 0 0 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0
 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1 1 0 1 0 0 1 1 0 1 1 0 1 1 0
 0 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 0 0
 1 0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 0 0 1 0 1 0 0 1 0 1 1 0 1 1 0 1 1 1 0 0 1 0 0 1 0 0 1 1 0
 0 1 0 1 1 0 1 1 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 1 1 1 0 1
 1 0 0 0 1 0 0 1 0 1 0 1 0 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1 1 1 0 1 1 0 0 0 1
 1 1 0 0 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 1 0
 1 0 0 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1
 1 1 1 0 1 1 1 1 1 0 0 1]
```

[33]:
```python
#Random Forest
clf = RandomForestClassifier()
grid_values = {"n_estimators": [10, 25, 50, 100],
                "max_depth": [5, 10, 20, 30],
                "min_samples_leaf": [1,2,4]}
rf_grid_clf_acc = GridSearchCV(clf, param_grid = grid_values, cv = 10, scoring␣
 ↪= 'accuracy')
rf_grid_clf_acc.fit(x_train, y_train)
print(rf_grid_clf_acc.best_score_)
print(rf_grid_clf_acc.best_estimator_)
y_pred = rf_grid_clf_acc.predict(x_test)
print("---------------------------------")
print("Predicted colrac")
print(y_pred)
```

```
0.8035070785070785
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=10, max_features='auto',
                       max_leaf_nodes=None, max_samples=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=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
---------------------------------
Predicted colrac
[1 0 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 0 0 0 1 1 1 0 0 1 1 1 1 1 0 1 1 0 1 1
 0 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 1 1 1
 0 1 0 0 1 0 1 1 0 1 1 1 0 1 1 0 1 0 0 1 0 1 1 1 0 0 1 0 0 1 0 1 1 1 0 1 0
 0 1 1 1 0 1 0 1 1 0 1 0 1 0 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 0 1 1
 0 1 1 1 0 0 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0
 1 1 1 1 1 1 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 1 0 1 0 0 1 1 0 1 0 0 1 1 0
 1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 0 1 1 0 1 0 1 1 0 1 0 0 1 1 0 1
 1 0 1 1 1 1 1 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 0 0 1 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 0 1 0 0 1 1 0 1 1 0
 0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 0 0 0 1 1 1 0 1 0 1 1 1 1 0 1
 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1 1 1 1 0 1 1 0 0 0 1
 1 1 0 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 1 0
 1 0 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1
 1 1 1 0 1 0 1 1 1 1 0 1]
```

[34]:
```python
#Gradient Boosting
clf = GradientBoostingClassifier()
grid_values = {"n_estimators": [10, 25, 50, 100],
                "max_depth": [5, 10, 20, 30]}
boo_grid_clf_acc = GridSearchCV(clf, param_grid = grid_values, cv = 10, scoring␣
 ↪= 'accuracy')
```

```
boo_grid_clf_acc.fit(x_train, y_train)
print(boo_grid_clf_acc.best_score_)
print(boo_grid_clf_acc.best_estimator_)
y_pred = boo_grid_clf_acc.predict(x_test)
print("---------------------------------")
print("Predicted colrac")
print(y_pred)
```

```
0.7825197646626219
GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', 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, n_estimators=50,
                           n_iter_no_change=None, presort='deprecated',
                           random_state=None, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
---------------------------------
Predicted colrac
[1 0 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 1 0 0 0 1 1 1 0 0 1 1 1 1 1 0 1 1 0 1 1
 0 1 1 0 1 1 1 1 1 0 1 0 1 1 0 1 0 0 1 1 1 1 0 1 1 1 1 1 0 1 0 0 0 1 1 1 1
 0 1 0 0 1 0 1 1 0 1 1 1 0 1 1 0 1 0 0 1 0 1 1 1 0 0 1 0 0 0 0 1 1 1 0 1 0
 0 1 1 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 0 1 1
 0 1 1 1 0 0 0 1 0 0 0 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 1 1 0 0 0 1 1 1 1 1 0
 1 1 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 0 1 0 0 1 1 0 1 1 0 1 1 0
 0 1 0 0 1 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 0 1 1 0 0 0 1 1 0 1 0 0 1 1 0 0
 1 0 1 1 1 1 1 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0
 0 1 1 1 1 1 1 1 0 0 1 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 0
 0 1 0 1 1 0 1 1 0 1 1 1 0 0 1 1 1 0 0 0 0 1 0 0 0 1 1 1 0 1 0 1 1 1 1 0 1
 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 1 0 1 0 1 1 1 0 1 1 0 0 0 1
 1 1 0 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 0 0 1 0
 1 0 0 0 1 0 1 1 0 1 1 0 1 1 1 0 1 0 1 0 1 1 0 0 1 1 1 1 1 1 1 1 0 0 1 1 1
 1 1 1 0 1 0 1 1 1 1 0 1]
```

Evaluate the models

[35]:
```
#Accuracy
en_cross_error = accuracy_score(y_train, en_grid_clf_acc.predict(x_train) >= 0.
 ↪5)

print("Logistic corss_error:", 1 - lg_grid_clf_acc.best_score_)
print("Naive Bayes corss_error:", 1 - nb_grid_clf_acc.best_score_)
print("Elastic Net corss_error:", 1 - en_cross_error)
print("Decision Tree corss_error:", 1 - dt_grid_clf_acc.best_score_)
print("Bagging corss_error:", 1 - bg_grid_clf_acc.best_score_)
print("Random Forest corss_error:", 1 - rf_grid_clf_acc.best_score_)
```

```
print("Boosting corss_error:", 1 - boo_grid_clf_acc.best_score_)
```

```
Logistic corss_error: 0.20186615186615187
Naive Bayes corss_error: 0.2655773120058834
Elastic Net corss_error: 0.1869918699186992
Decision Tree corss_error: 0.22022890237175952
Bagging corss_error: 0.21006618863761717
Random Forest corss_error: 0.19649292149292152
Boosting corss_error: 0.21748023533737815
```

[36]:

```
lg_roc_auc = roc_auc_score(y_train, lg_grid_clf_acc.predict(x_train))
nb_roc_auc = roc_auc_score(y_train, nb_grid_clf_acc.predict(x_train))
en_roc_auc = roc_auc_score(y_train, en_grid_clf_acc.predict(x_train) >= 0.5)
dt_roc_auc = roc_auc_score(y_train, dt_grid_clf_acc.predict(x_train))
bg_roc_auc = roc_auc_score(y_train, bg_grid_clf_acc.predict(x_train))
rf_roc_auc = roc_auc_score(y_train, rf_grid_clf_acc.predict(x_train))
boo_roc_auc = roc_auc_score(y_train, boo_grid_clf_acc.predict(x_train))

print("Logistic ROC_AUC:", lg_roc_auc)
print("Naive Bayes ROC_AUC:", nb_roc_auc)
print("Elastic Net ROC_AUC:", en_roc_auc)
print("Decision Tree ROC_AUC:", dt_roc_auc)
print("Bagging ROC_AUC:", bg_roc_auc)
print("Random Forest ROC_AUC:", rf_roc_auc)
print("Boosting ROC_AUC:", boo_roc_auc)
```

```
Logistic ROC_AUC: 0.8191247526574938
Naive Bayes ROC_AUC: 0.7431245685886521
Elastic Net ROC_AUC: 0.8123326123970365
Decision Tree ROC_AUC: 0.7915328332796466
Bagging ROC_AUC: 0.7964327458469468
Random Forest ROC_AUC: 0.9832902305462243
Boosting ROC_AUC: 0.9584298927798997
```

As we can see from the result above, Random Forest has the highest ROC_AUC score and second lowest error. Therefore, Random Forest is the optimal choice we would choose. While the other models may perform good on either roc-auc or error. For example, Boosting is good at ROC-AUC but perform not so well at boosting cross-error; and Elastic Net is low on cross-error but perform not so well when consider ROC-AUC. But with a good performance, the overfitting maybe an issue. Nevertheless, I will choose Random Forest as the optimal choice.

Evaluate the best model

[37]:

```
rf_test_cross_error = accuracy_score(y_test, rf_grid_clf_acc.predict(x_test))
rf_test_roc_auc = roc_auc_score(y_test, rf_grid_clf_acc.predict(x_test))

print("Random Forest Cross Error (Test):", 1 - rf_test_cross_error)
print("Random Forest ROC_AUC (Test):", rf_test_roc_auc)
```

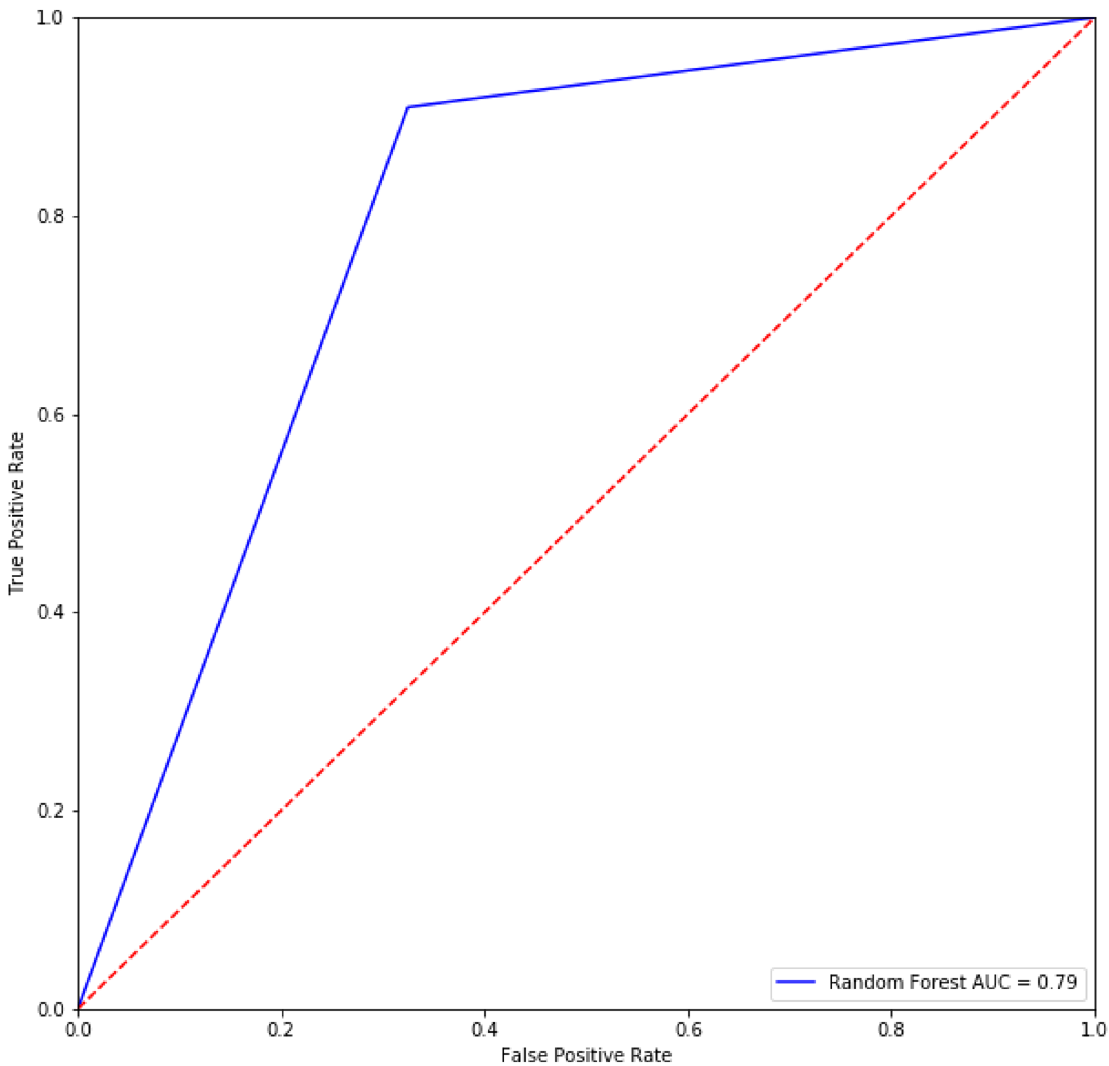
```
Random Forest Cross Error (Test): 0.19878296146044627
Random Forest ROC_AUC (Test): 0.7924362793776896
```

[38]:

```
plt.figure(figsize=(10,10))
preds = rf_grid_clf_acc.predict(x_test)
fpr, tpr, threshold = metrics.roc_curve(y_test, preds)
rf_roc_auc = metrics.auc(fpr, tpr)
plt.plot(fpr, tpr, 'b', label = 'Random Forest AUC = %0.2f' % rf_roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
```

[38]:

```
Text(0.5, 0, 'False Positive Rate')
```

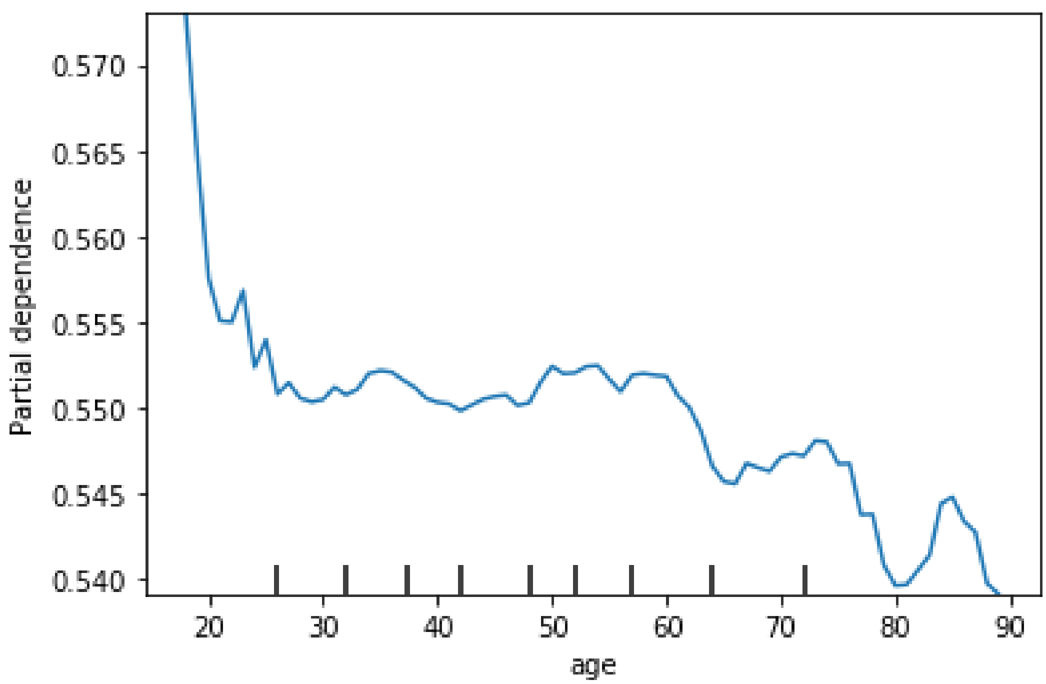


As we can see from the result above, when using the test set, the ROC-AUC value for Random Forest goes down from 0.9832902305462243(train set) to 0.7924362793776896(test set). And the Cross Error goes up from 0.19649292149292152 to 0.19878296146044627. This indicates that the "best" model does not generalize well. Just as I mentioned above, the overfitting may still be an issue here.

Bonus: PDPs/ICE

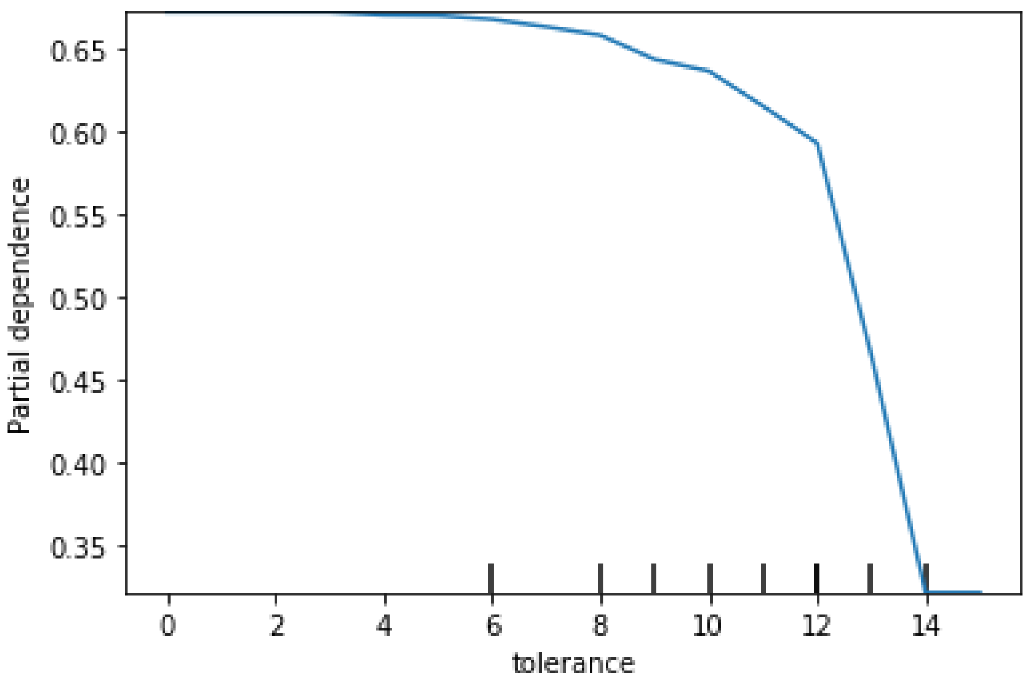
[44]:
```
feature = [x_train.columns.get_loc("age")]
plot_partial_dependence(rf_grid_clf_acc, x_test, feature)
```

[44]:
```
<sklearn.inspection._partial_dependence.PartialDependenceDisplay at
0x1e2cf08aa58>
```



[46]:
```
feature = [x_train.columns.get_loc("tolerance")]
plot_partial_dependence(rf_grid_clf_acc, x_test, feature)
```

[46]:
```
<sklearn.inspection._partial_dependence.PartialDependenceDisplay at
0x1e2cf299c50>
```



Partial dependence plots (PDP) show the dependence between the target response and a set of / one 'target' feature(s), marginalizing over the values of all other features. One-way PDPs tell us about the interaction between the target response and the target feature. Here in the graphs above, it shows clearly that relationship between colrac and age or tolerance (effect of the age or tolerance on colrac). As age goes up, its partical dependence effect goes down sightly. Therefore, age may not be a strong predictor for colrac. As tolerance goes up, its partical dependence effect goes down more fiercely (compare to the age). Therefore, we could say tolerance is a relatively stronger predictor for colrac.

Given the graphs above, assume that the target features are independent from the complement features. The question here is "Consider a person who believes that Blacks are genetically inferior Should such a person be allowed to teach in a college or university, or not?". For all ages the probability not allowed remain pretty constant ranging from 0.54 to 0.57; while for the tolerance level, before level 14, the probability not allowed range from 0.6 to 0.7, but as the tolerance level reaches 14, the probability not allowed goes down to around 0.3.

Note that PDPs assume that the target features are independent from the complement features, and whether this assumption holds for this situation remains questionable.