# Xu_Yilun_HW05

March 1, 2020

## 1 Conceptual

### 1.1 1

I think in binary classification, although classification error is the most intuitive one, the others have advantages in terms of being characterized by negative curvature, which will increase the preference of splits that create region pairs where at least one is of highly purity and this may bring impurity in other regions. In addition, the other two functions are smooth curves which are better for optimization.That is to say, the Gini Indexa nd cross-entropy are better to grow a decision tree since they perfom better to purity.

When it comes to pruning a decision tree, I think all of the three standards can be used. However, if we use the prediction accuracy as the standard to evaluate the tree, we should choose classification error to prune the decision tree.

## 2 Application

[1]:
```
import pandas as pd
from sklearn import tree
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegressionCV
import random
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import ElasticNetCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn import metrics
from sklearn.metrics import roc_auc_score
import warnings
warnings.filterwarnings("ignore")
from sklearn.metrics import accuracy_score
from sklearn.inspection import plot_partial_dependence
from pdpbox import pdp, get_dataset, info_plots
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from pycebox.ice import *
random.seed(9001)
```

[2]:
```
train = pd.read_csv("C:/Users/mac/Desktop/temp/hw05/gss_train.csv")
test = pd.read_csv("C:/Users/mac/Desktop/temp/hw05/gss_test.csv")
x_cols = [x for x in train.columns if x != 'colrac']
x_train = train[x_cols]
y_train = train[['colrac']]
x_test = test[x_cols]
y_test = test[['colrac']]
```

## 2.1 2

### 2.1.1 logistic regression

[3]:
```
param_lr = {'Cs': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}
lr_grid = GridSearchCV(LogisticRegressionCV(cv = 10),param_lr, cv = 10, scoring␣
↪= 'accuracy')
lr_grid.fit(x_train, y_train)
```

[3]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=LogisticRegressionCV(Cs=10, class_weight=None, cv=10,
                                            dual=False, fit_intercept=True,
                                            intercept_scaling=1.0,
                                            l1_ratios=None, max_iter=100,
                                            multi_class='auto', n_jobs=None,
                                            penalty='l2', random_state=None,
                                            refit=True, scoring=None,
                                            solver='lbfgs', tol=0.0001,
                                            verbose=0),
             iid='deprecated', n_jobs=None,
             param_grid={'Cs': [0.001, 0.01, 0.1, 1, 10, 100, 1000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)
```

### 2.1.2 Naive Bayes

[4]:
```
param_nb = {'var_smoothing': np.logspace(0,-9, num = 100)}
nb_grid = GridSearchCV(GaussianNB(), param_nb, cv = 10, scoring = 'accuracy')
nb_grid.fit(x_train, y_train)
```

[4]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=GaussianNB(priors=None, var_smoothing=1e-09),
             iid='deprecated', n_jobs=None,
             param_grid={'var_smoothing': array([1.00000000e+00, 8.11130831e-01,
```

```
6.57933225e-01, 5.33669923e-01,
       4.32876128e-01, 3.51119173e-01, 2.84803587e-01, 2.31012970e-01,
       1.87381742e-01, 1.51991108e-01, 1.23284674e-01, 1.00000000e-01,
       8.11130831e-02, 6…
       5.33669923e-08, 4.32876128e-08, 3.51119173e-08, 2.84803587e-08,
       2.31012970e-08, 1.87381742e-08, 1.51991108e-08, 1.23284674e-08,
       1.00000000e-08, 8.11130831e-09, 6.57933225e-09, 5.33669923e-09,
       4.32876128e-09, 3.51119173e-09, 2.84803587e-09, 2.31012970e-09,
       1.87381742e-09, 1.51991108e-09, 1.23284674e-09, 1.00000000e-09])},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)
```

### 2.1.3 Elastic net regression

[5]:
```
param_en = {'max_iter':[100,110,120,130,140]}
en_grid = GridSearchCV(ElasticNetCV(cv = 10), param_en, scoring = 'r2', cv = 10)
en_grid.fit(x_train, y_train)
```

[5]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=ElasticNetCV(alphas=None, copy_X=True, cv=10, eps=0.001,
                                    fit_intercept=True, l1_ratio=0.5,
                                    max_iter=1000, n_alphas=100, n_jobs=None,
                                    normalize=False, positive=False,
                                    precompute='auto', random_state=None,
                                    selection='cyclic', tol=0.0001, verbose=0),
             iid='deprecated', n_jobs=None,
             param_grid={'max_iter': [100, 110, 120, 130, 140]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='r2', verbose=0)
```

### 2.1.4 Decision tree (CART)

[6]:
```
param_dt = {'criterion':['gini', 'entropy'], 'min_samples_split' :␣
 ↪range(10,500,20),'max_depth': range(1,56)}
dt_grid = GridSearchCV(DecisionTreeClassifier(),param_grid=param_dt, cv = 10,␣
 ↪scoring = 'accuracy')
dt_grid.fit(x_train, y_train)
```

[6]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_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='deprecated',
                                       random_state=None,
                                       splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': range(1, 56),
                         'min_samples_split': range(10, 500, 20)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)
```

### 2.1.5 Bagging

[7]:
```
param_bagging = {"max_samples": [0.5, 1.0],"max_features":␣
↪range(1,56),"bootstrap": [True, False],
                "bootstrap_features": [True, False]}
bagging_grid = GridSearchCV(BaggingClassifier(),param_grid = param_bagging, cv␣
↪= 10, scoring = 'accuracy')
bagging_grid.fit(x_train, y_train)
```

[7]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=BaggingClassifier(base_estimator=None, bootstrap=True,
                                         bootstrap_features=False,
                                         max_features=1.0, max_samples=1.0,
                                         n_estimators=10, n_jobs=None,
                                         oob_score=False, random_state=None,
                                         verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'bootstrap': [True, False],
                         'bootstrap_features': [True, False],
                         'max_features': range(1, 56),
                         'max_samples': [0.5, 1.0]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)
```

### 2.1.6 Random Forest

[8]:
```
param_rf = {'max_features': ['auto', 'sqrt', 'log2'],'max_depth' :␣
↪[4,5,6,7,8],'criterion' :['gini', 'entropy']}
cf_grid = GridSearchCV(RandomForestClassifier(random_state = 42), param_rf, cv␣
↪= 10)
cf_grid.fit(x_train, y_train)
```

[8]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
```

```
                                          class_weight=None,
                                          criterion='gini', max_depth=None,
                                          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=42,
                                          verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': [4, 5, 6, 7, 8],
                         'max_features': ['auto', 'sqrt', 'log2']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

### 2.1.7 Boosting

[9]:
```
param_boosting = {'max_features': ['auto', 'sqrt', 'log2'],'max_depth' :␣
↪[4,5,6,7,8],"criterion": ["friedman_mse",  "mae"]}
boosting_grid = GridSearchCV(GradientBoostingClassifier(), param_boosting, cv =␣
↪10)
boosting_grid.fit(x_train, y_train)
```

[9]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=GradientBoostingClassifier(ccp_alpha=0.0,
                                                  criterion='friedman_mse',
                                                  init=None, learning_rate=0.1,
                                                  loss='deviance', max_depth=3,
                                                  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=100,
                                                  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),
             iid='deprecated', n_jobs=None,
             param_grid={'criterion': ['friedman_mse', 'mae'],
                         'max_depth': [4, 5, 6, 7, 8],
                         'max_features': ['auto', 'sqrt', 'log2']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

## 2.2 3

### 2.2.1 logistic regression

[10]:
```
print("cross-validated error :", 1-lr_grid.best_score_)
y_predicted_lr = lr_grid.predict(x_train)
print("ROC/AUC :", roc_auc_score(y_train, y_predicted_lr))
```

```
cross-validated error : 0.20324508181651035
ROC/AUC : 0.8160351571487735
```

### 2.2.2 Naive Bayes

[48]:
```
print("cross-validated error :", 1 - nb_grid.best_score_)
y_predicted_nb = nb_grid.predict(x_train)
print("ROC/AUC :", roc_auc_score(y_train, y_predicted_nb))
```

```
cross-validated error : 0.26557271557271567
ROC/AUC : 0.7419023514794533
```

### 2.2.3 Elastic net regression

[49]:
```
print("cross-validated error :", 1 - en_grid.best_score_)
y_predicted_en = en_grid.predict(x_train)
print("ROC/AUC :", roc_auc_score(y_train, y_predicted_en))
```

```
cross-validated error : 0.595623133233003
ROC/AUC : 0.8914233123188071
```

### 2.2.4 Decision tree (CART)

[50]:
```
print("cross-validated error :", 1 - dt_grid.best_score_)
y_predicted_dt = dt_grid.predict(x_train)
print("ROC/AUC :", roc_auc_score(y_train, y_predicted_dt))
```

```
cross-validated error : 0.21820187534473257
ROC/AUC : 0.7903106161704477
```

### 2.2.5 Bagging

[51]:
```
print("cross-validated error :", 1 - bagging_grid.best_score_)
y_predicted_bagging = bagging_grid.predict(x_train)
print("ROC/AUC :", roc_auc_score(y_train, y_predicted_bagging))
```

```
cross-validated error : 0.21810075381503968
ROC/AUC : 1.0
```

### 2.2.6 Random Forest

[53]:
```
print("cross-validated error :", 1 - cf_grid.best_score_)
y_predicted_rf = cf_grid.predict(x_train)
print("ROC/AUC :", roc_auc_score(y_train, y_predicted_rf))
```

```
cross-validated error : 0.20530428387571242
ROC/AUC : 0.8374221158713359
```

### 2.2.7 Boosting

[52]:
```
print("cross-validated error :", 1 - boosting_grid.best_score_)
y_predicted_boosting = boosting_grid.predict(x_train)
print("ROC/AUC :", roc_auc_score(y_train, y_predicted_boosting))
```

```
cross-validated error : 0.1944521051663909
ROC/AUC : 1.0
```

## 2.3 4

[40]:
```
cv_accuracy = {'Logistic regression': lr_grid.best_score_, 'Naive bayes':␣
↪nb_grid.best_score_, 'Elastic net regression': en_grid.best_score_,
          'Decision tree (CART)': dt_grid.best_score_, 'Bagging': bagging_grid.
↪best_score_, 'Random forest': cf_grid.best_score_,
          'Boosting': boosting_grid.best_score_}
```

[41]:
```
cv_accuracy
```

[41]:
```
{'Logistic regression': 0.7967549181834896,
 'Naive bayes': 0.7344272844272843,
 'Elastic net regression': 0.404376866766997,
 'Decision tree (CART)': 0.7817981246552674,
 'Bagging': 0.7818992461849603,
 'Random forest': 0.7946957161242876,
 'Boosting': 0.8055478948336091}
```

[42]:
```
roc_auc = {'Logistic regression': roc_auc_score(y_train, y_predicted_lr),␣
↪'Naive bayes': roc_auc_score(y_train, y_predicted_nb),
```

```
          'Elastic net regression': roc_auc_score(y_train, y_predicted_en),␣
↪'Decision tree (CART)': roc_auc_score(y_train, y_predicted_dt),
          'Bagging': roc_auc_score(y_train, y_predicted_bagging), 'Random␣
↪forest': roc_auc_score(y_train, y_predicted_rf),
          'Boosting': roc_auc_score(y_train, y_predicted_boosting)}
```

[43]:
```
roc_auc
```

[43]:
```
{'Logistic regression': 0.8160351571487735,
 'Naive bayes': 0.7419023514794533,
 'Elastic net regression': 0.8914233123188071,
 'Decision tree (CART)': 0.7903106161704477,
 'Bagging': 1.0,
 'Random forest': 0.8374221158713359,
 'Boosting': 1.0}
```

[44]:
```
print("Ranks of cv accuracy of models:")
sorted(cv_accuracy.items(), key=lambda d:d[1], reverse = True)
```

```
Ranks of cv accuracy of models:
```

[44]:
```
[('Boosting', 0.8055478948336091),
 ('Logistic regression', 0.7967549181834896),
 ('Random forest', 0.7946957161242876),
 ('Bagging', 0.7818992461849603),
 ('Decision tree (CART)', 0.7817981246552674),
 ('Naive bayes', 0.7344272844272843),
 ('Elastic net regression', 0.404376866766997)]
```

[45]:
```
print("Ranks of ROC/AUC of models:")
sorted(roc_auc.items(), key=lambda d:d[1], reverse = True)
```

```
Ranks of ROC/AUC of models:
```

[45]:
```
[('Bagging', 1.0),
 ('Boosting', 1.0),
 ('Elastic net regression', 0.8914233123188071),
 ('Random forest', 0.8374221158713359),
 ('Logistic regression', 0.8160351571487735),
 ('Decision tree (CART)', 0.7903106161704477),
 ('Naive bayes', 0.7419023514794533)]
```

According to the above analysis, we can see that the boosting model ranks first and second in the two different standards, and the CV accuracy and ROC/AUC indexes are both high. In comparison, logistic regression does not have a very high ROC/AUC and bagging does not have a very high cv accuracy. Therefore, we prefer to say that the tuned boosting model here is the best model. (I think the ranking is related to seed. I ran the code before and boosting ranks first in both standards.)

### 2.4 5

[46]:
```
y_predicted_test_boosting = boosting_grid.predict(x_test)
classification_error = 1 - accuracy_score(y_test, y_predicted_test_boosting)
auc_test = roc_auc_score(y_test, y_predicted_test_boosting)
print("classification error in test dataset:", classification_error)
print("AUC in test dataset:", auc_test)
```

```
classification error in test dataset: 0.21095334685598377
AUC in test dataset: 0.782340284673949
```

[47]:
```
print("cross-validated error in train dataset:", 1 - boosting_grid.best_score_)
y_predicted_boosting = boosting_grid.predict(x_train)
print("ROC/AUC in train dataset:", roc_auc_score(y_train, y_predicted_boosting))
```

```
cross-validated error in train dataset: 0.1944521051663909
ROC/AUC in train dataset: 1.0
```

Generalization demonstrates the ability of a model to well fit new data not previously seen. To achive better generalization of a model, we should pay attention to both the problems of underfitting and overfitting.Overfitting means that the model predicts the train dataset too well, while underfitting means that the model can not predicts the train dataset and new data well neither. Actually, decision trees are a nonparametric machine learning algorithm which is considered to be highly subject to overfitting training data. We can use the method of pruning the decision tree to avoid this problem.

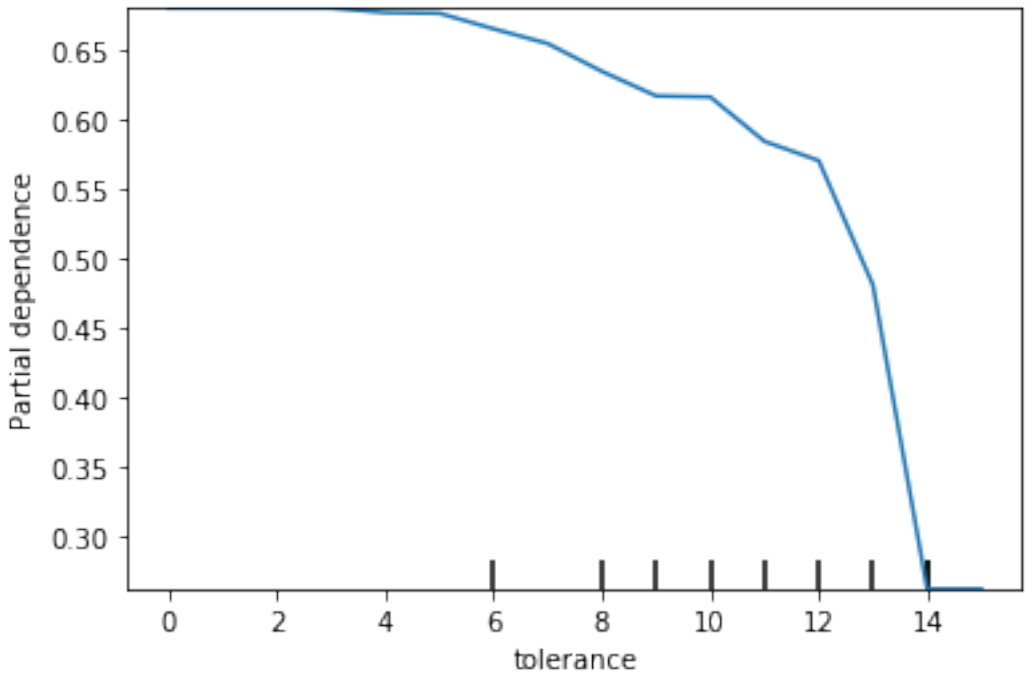
Here I think we can use the performance of the specific model on the train dataset and test dataset to evaluate this model's generalization ability.

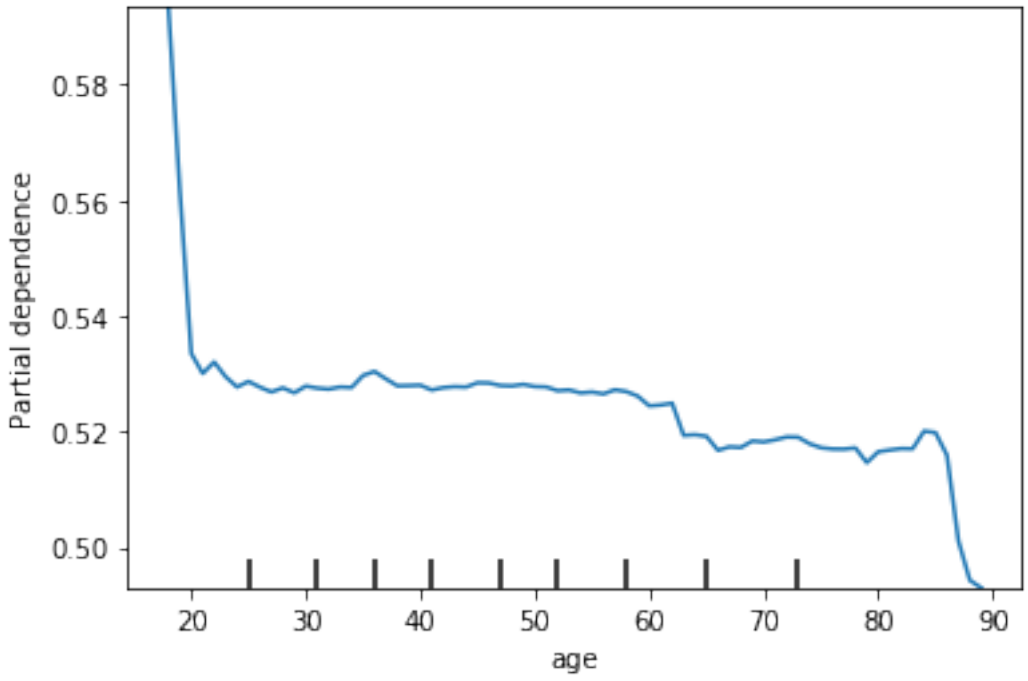
According to the above comparison, we can see that the cv accuracy and ROC/AUC in the train dataset and test dataset are all relatively high, especially in the train dataset. In addition, the indexes in the test dataset are both lower than the indexes in the train dataset. Therefore, I think we do not encounter the problems of underfitting and overfitting. Thus, I maintain that the "best" model generalizes well.

### 2.5 6

[25]:
```
plot_partial_dependence(boosting_grid,x_train, features = ['tolerance'])
plot_partial_dependence(boosting_grid,x_train, features = ['age'])
```

[25]:
```
<sklearn.inspection._partial_dependence.PartialDependenceDisplay at
0x25470c76c88>
```

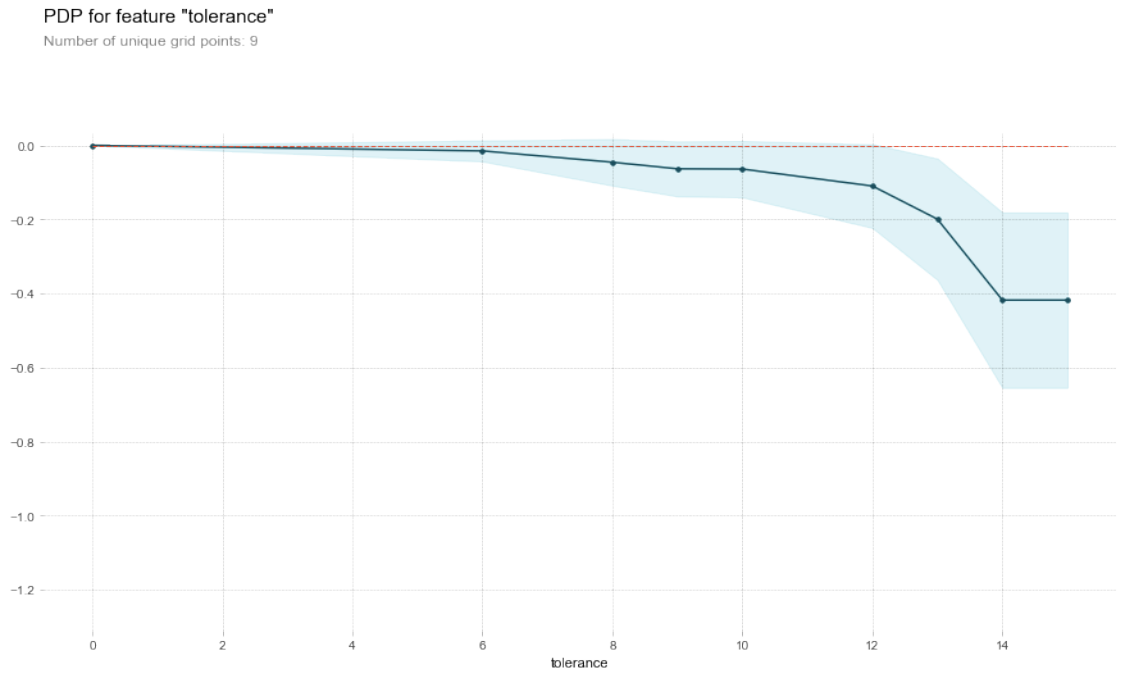


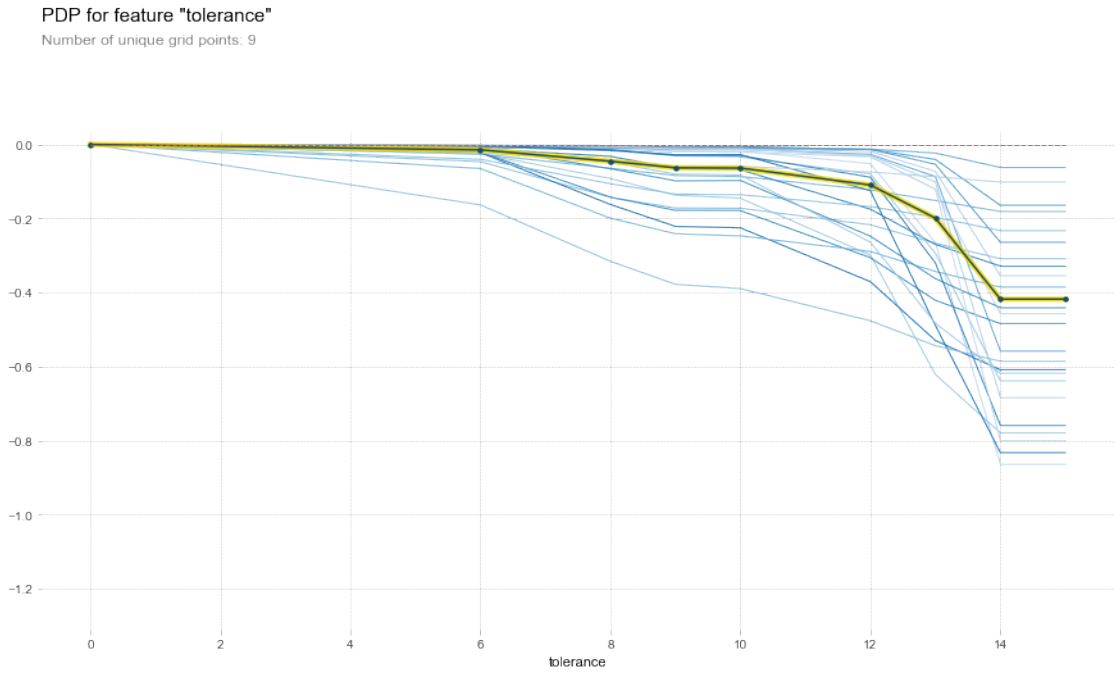

[26]:
```
from pdpbox.pdp import pdp_isolate, pdp_plot
```

```
def plot_pdp(model, df, feature, cluster_flag=False, nb_clusters=None,␣
↪lines_flag=False):

    # Create the data that we will plot
    pdp_goals = pdp_isolate(model=model, dataset=df, model_features=df.columns.
↪tolist(), feature=feature)

    # plot it
    pdp_plot(pdp_goals, feature, cluster=cluster_flag,␣
↪n_cluster_centers=nb_clusters, plot_lines=lines_flag)
    plt.show()
```

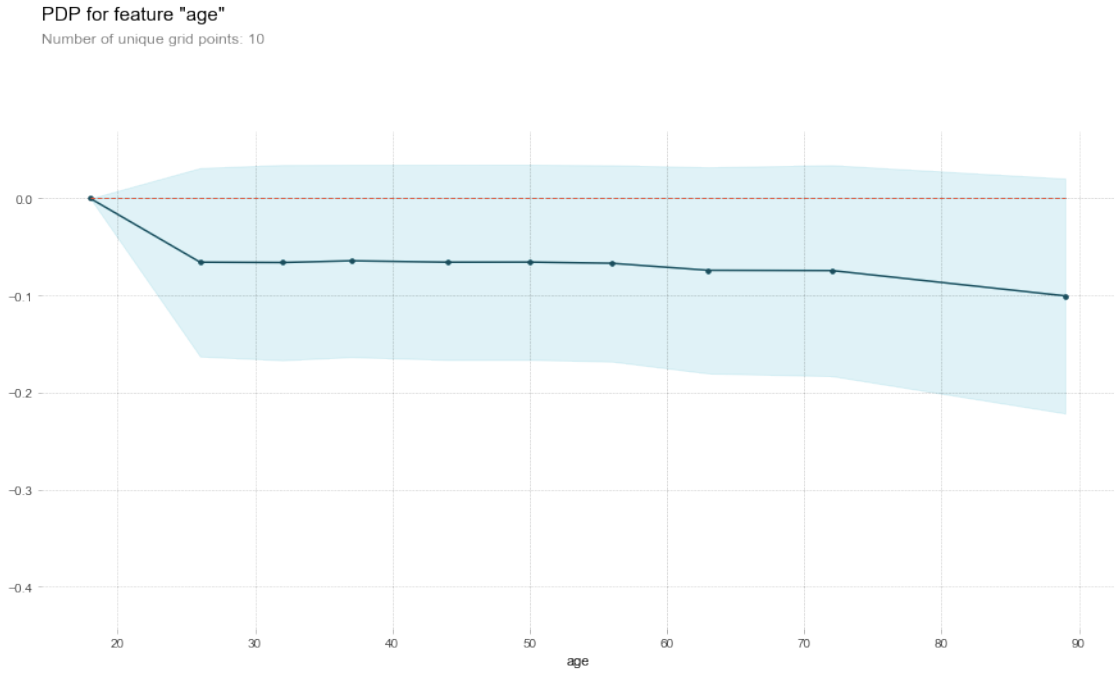
[27]:
```
from pdpbox.pdp import pdp_isolate
plot_pdp(boosting_grid, x_train, 'tolerance')
```



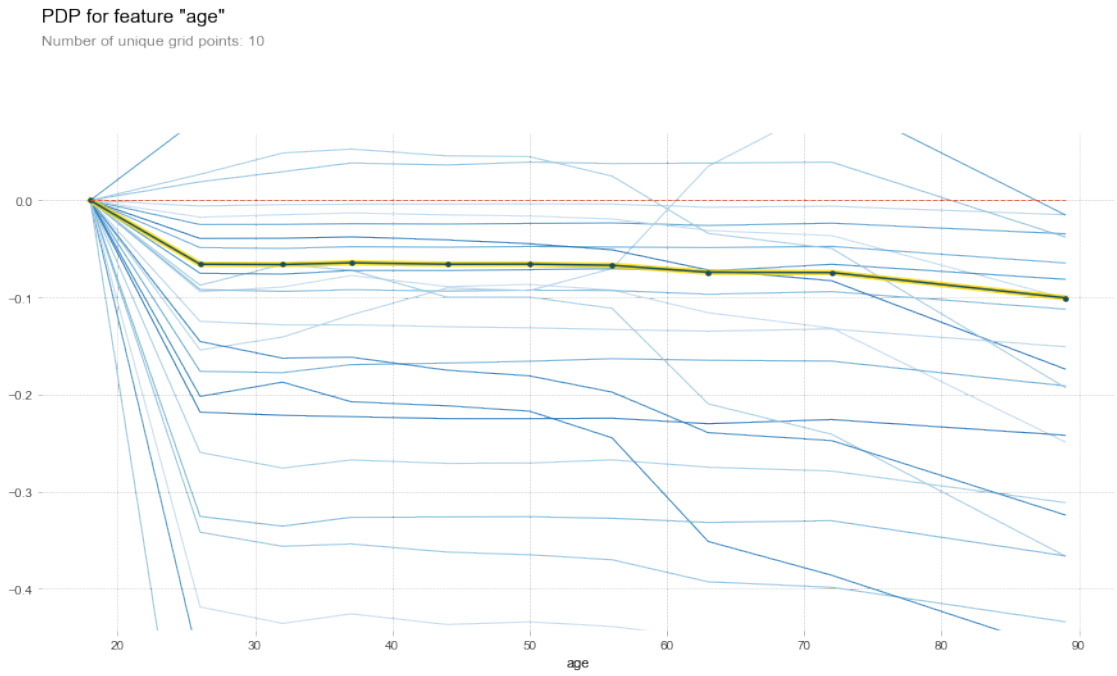

[28]:
```
plot_pdp(boosting_grid, x_train, 'tolerance', cluster_flag=True,␣
↪nb_clusters=24, lines_flag=True)
```




[29]:
```
plot_pdp(boosting_grid, x_train, 'age')
```



[30]:
```
plot_pdp(boosting_grid, x_train, 'age', cluster_flag=True, nb_clusters=24,␣
↪lines_flag=True)
```




[31]:
```
from pdpbox.pdp import *
features_to_plot = ['tolerance', 'age']
inter1  =  pdp_interact(model=boosting_grid, dataset=x_train,␣
↪model_features=x_train.columns, features=features_to_plot)
pdp_interact_plot(pdp_interact_out=inter1, feature_names=features_to_plot,␣
↪plot_type='grid')
```

[31]:
```
(<Figure size 540x684 with 3 Axes>,
 {'title_ax': <matplotlib.axes._subplots.AxesSubplot at 0x2546ff40908>,
  'pdp_inter_ax': <matplotlib.axes._subplots.AxesSubplot at 0x2546fd75c48>})
```


Partial independence shows how the dependent variable partially depends on a specific independent variable or a specific group of independent variables. Or we can say this probability describes the marginal effect of small number of specific independent variables. Only in terms of the variable 'tolerance', we can see the dependent variable partially depends less and less on this independent variable as the value of 'tolerance' increases, and this is also the case which happens to the independent variable 'age'. In the last figure of the PDP interaction of these two variables, we can see that with 'tolerance' and 'age' increases, the probability decreases (the color grows darker). To be specific, we can see that when people are over roughly 22 years old (above the bottom row of 'age'), when people's age increases, the influence of their age on their attitudes towards racist

college professors decreases. To some extent, we can understand this point since when people become an adult their thoughts can be some kind of more stable.In terms of tolerance', we can see that when the degree is above 13, the influence of their tolerance on their attitudes towards racist college professors decreases sharply. In addtion, we can see that when the age is fixed, whether the tolerance degree is 14 or 15 does not change the marginal effect of people's tolerance on their attitudes towards racist college professors decreases sharply.