# Wang_Miaohan_HW5

March 1, 2020

[1]:
```
import random
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier,␣
 ↪GradientBoostingClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import cross_val_score, GridSearchCV
from sklearn.inspection import plot_partial_dependence
```

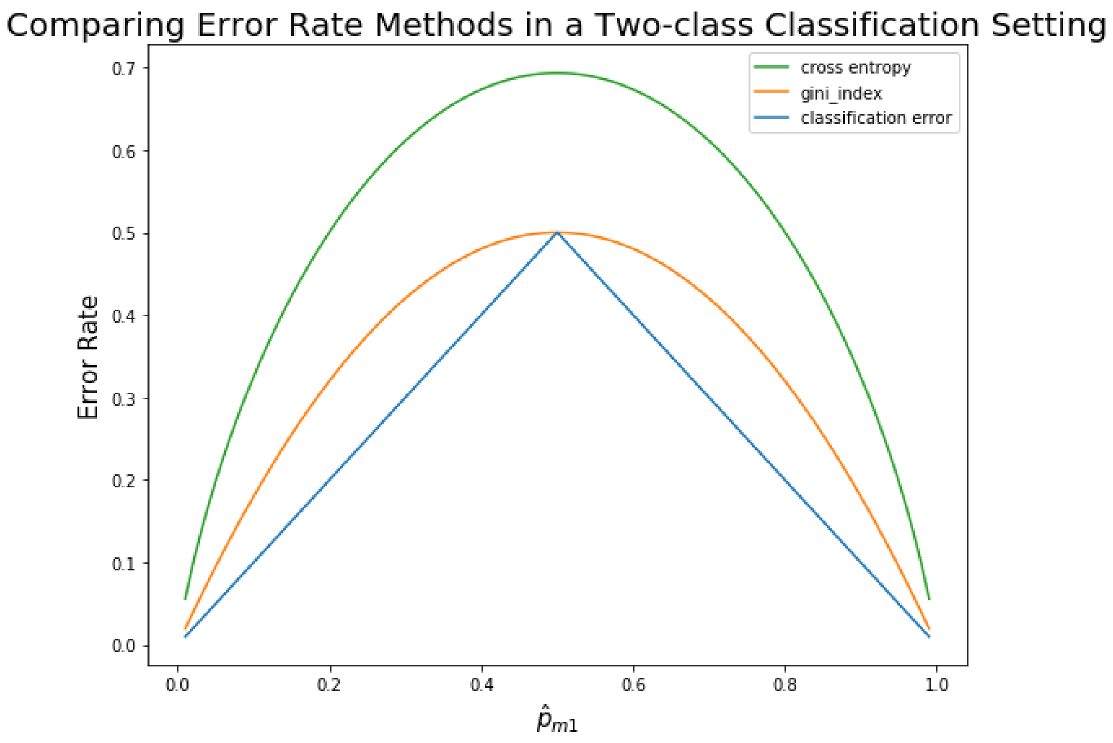
## Conceptual Exercise

When we have only two classes of outcome (supposedly outcome 1 and outcome 2), we will only have two probabilities $\hat{p}_{m1}$ and $\hat{p}_{m2}$ in our

[2]:
```
# calculation of error rates in a simple two-class case
def c_error(p):
    return 1 - max(p, 1 - p)
def gini(p):
    return 2 * p * (1 - p)
def cross(p):
    return -(p * np.log(p) + (1 - p) * np.log(1 - p))

# generating plot
x_range = np.arange(0.01,1,0.01)
classification_error = list(map(c_error, x_range))
gini_index = list(map(gini, x_range))
cross_entropy = list(map(cross, x_range))

plt.figure(figsize=(9,7))
plt.plot(x_range, cross_entropy, label='cross entropy', color='tab:green')
plt.plot(x_range, gini_index, label='gini_index', color='tab:orange')
plt.plot(x_range, classification_error, label='classification error')
plt.xlabel('$\hat{p}_{m1}$', fontsize=15)
plt.ylabel('Error Rate', fontsize=15)
plt.title('Comparing Error Rate Methods in a Two-class Classification Setting',␣
 ↪fontsize=20)
plt.legend();
```



When growing a tree, we want to find the feature that can best split the data. Given that features can perform very similarly at splits, we need a sensitive measure of prediction error to differetiate feature performance. Among the three error rates, cross entropy is the most sensitive to changes in $\hat{p}_{m1}$, hence the best error rates to use in growing trees.

When pruning a tree, we want to cut redundant features to obtain the model with best predictive accuracy. Therefore, we would like to obtain the optimal tree depth with least mistakes in classification. Hence, in pruning, we incorporate the classification error, which tell us about model accuracy at each step in the most straightforward way.

## Application Exercise

```
[3]: train_df = pd.read_csv('data/gss_train.csv')
     test_df = pd.read_csv('data/gss_test.csv')

     y_train = train_df.colrac
```

```
X_train = train_df.drop('colrac', axis=1)
y_test = test_df.colrac
X_test = test_df.drop('colrac', axis=1)

seed = 1234
error_rates = {}
```

### Logistic Regression

[4]:
```
log_mod = LogisticRegression(random_state=seed, max_iter=1000)
log_err = 1 - cross_val_score(log_mod, X_train, y_train, scoring='accuracy',␣
 ↪cv=10).mean()

log_mod.fit(X_train, y_train)
log_auc = roc_auc_score(y_train, log_mod.predict(X_train))

error_rates['Logistic Regression'] = [log_err, log_auc]
```

### Naive Bayes

[5]:
```
nb_mod = GaussianNB()
nb_err = 1 - cross_val_score(nb_mod, X_train, y_train, scoring='accuracy',␣
 ↪cv=10).mean()

nb_mod.fit(X_train, y_train)
nb_auc = roc_auc_score(y_train, nb_mod.predict(X_train))

error_rates['Naive Bayes'] = [nb_err, nb_auc]
```

### Elastic Net Regression

[6]:
```
# hyperpameter tuning
el_mod = LogisticRegressionCV(cv=10, penalty='elasticnet', scoring='accuracy',␣
 ↪max_iter=2000,
                              solver='saga', random_state=seed, l1_ratios = np.
 ↪arange(0, 1.1, 0.1))

el_mod.fit(X_train, y_train)
el_mod.l1_ratio_
```

[6]:
```
array([0.2])
```

[7]:
```
# fitting best model
el_mod_best = LogisticRegression(penalty='elasticnet', solver='saga',␣
↪l1_ratio=0.2, max_iter=1500)
el_err = 1 - cross_val_score(el_mod_best, X_train, y_train, scoring='accuracy',␣
↪cv=10).mean()
el_err

el_mod_best.fit(X_train, y_train)
el_auc = roc_auc_score(y_train, el_mod_best.predict(X_train))

error_rates['Elastic Net Regression'] = [el_err, el_auc]
```

### Decision Tree (CART)

[8]:
```
# hyperpameter tuning
tree_params = {'criterion':['gini', 'entropy'],
               'max_depth': np.arange(1, 56),
               'min_samples_leaf':np.arange(1,11)}
tree_mod = GridSearchCV(DecisionTreeClassifier(), tree_params,␣
↪scoring='accuracy', cv=10)
tree_mod.fit(X_train, y_train)
tree_mod.best_estimator_
```

[8]:
```
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=5, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
```

[9]:
```
# fitting best model
tree_mod_best = DecisionTreeClassifier(criterion='gini', max_depth=4,␣
↪min_samples_leaf=5)
tree_err = 1 - cross_val_score(tree_mod_best, X_train, y_train,␣
↪scoring='accuracy', cv=10).mean()

tree_mod_best.fit(X_train, y_train)
tree_auc = roc_auc_score(y_train, tree_mod_best.predict(X_train))

error_rates['Decision Tree'] = [tree_err, tree_auc]
```

### Bagging

```
[12]: # hyperpameter tuning
      bag_params= {'n_estimators': np.arange(100, 900, 100)}
      bag_mod = GridSearchCV(BaggingClassifier(max_samples=0.6, # suggested by https:/
       ↪/www.youtube.com/watch?v=2Mg8QD0F1dQ
                                               random_state=seed), bag_params,␣
       ↪scoring='accuracy', cv=10)
      bag_mod.fit(X_train, y_train)
      bag_mod.best_estimator_
```

```
[12]: BaggingClassifier(base_estimator=None, bootstrap=True, bootstrap_features=False,
                        max_features=1.0, max_samples=0.6, n_estimators=700,
                        n_jobs=None, oob_score=False, random_state=1234, verbose=0,
                        warm_start=False)
```

```
[13]: # fitting best model
      bag_mod_best = BaggingClassifier(n_estimators=700, max_samples=0.6,␣
       ↪random_state=seed)
      bag_err = 1 - cross_val_score(bag_mod_best, X_train, y_train,␣
       ↪scoring='accuracy', cv=10).mean()

      bag_mod_best.fit(X_train, y_train)
      bag_auc = roc_auc_score(y_train, bag_mod_best.predict(X_train))

      error_rates['Bagging'] = [bag_err, bag_auc]
```

### Random Forest

```
[19]: # hyperpameter tuning
      rf_params= {'n_estimators': np.arange(100, 600, 100),
                  'min_samples_leaf': np.arange(5, 11),
                  'max_features': ['sqrt', 'log2']}
      rf_mod = GridSearchCV(RandomForestClassifier(random_state=seed), rf_params,␣
       ↪scoring='accuracy', cv=10)
      rf_mod.fit(X_train, y_train)
      rf_mod.best_estimator_
```

```
[19]: RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                             criterion='gini', max_depth=None, max_features='sqrt',
                             max_leaf_nodes=None, max_samples=None,
                             min_impurity_decrease=0.0, min_impurity_split=None,
                             min_samples_leaf=8, min_samples_split=2,
                             min_weight_fraction_leaf=0.0, n_estimators=200,
                             n_jobs=None, oob_score=False, random_state=1234,
                                verbose=0, warm_start=False)
```

[20]:
```
# fitting best model
rf_mod_best = RandomForestClassifier(n_estimators=200, min_samples_leaf=8,␣
 ↪max_features='sqrt', random_state=seed)
rf_err = 1 - cross_val_score(rf_mod_best, X_train, y_train, scoring='accuracy',␣
 ↪cv=10).mean()

rf_mod_best.fit(X_train, y_train)
rf_auc = roc_auc_score(y_train, rf_mod_best.predict(X_train))

error_rates['Random Forest'] = [rf_err, rf_auc]
```

### Boosting

[22]:
```
# hyperpameter tuning
bo_params = {'learning_rate': [0.001, 0.01],
             'n_estimators': [10, 100, 1000],
             'max_depth': [1, 2, 3]}
bo_mod = GridSearchCV(GradientBoostingClassifier(random_state=seed), bo_params,␣
 ↪cv=10, scoring='accuracy')
bo_mod.fit(X_train, y_train)
bo_mod.best_estimator_
```

[22]:
```
GradientBoostingClassifier(ccp_alpha=0.0, criterion='friedman_mse', init=None,
                           learning_rate=0.01, loss='deviance', max_depth=2,
                           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=1000,
                           n_iter_no_change=None, presort='deprecated',
                           random_state=1234, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

[23]:
```
# fitting best model
bo_mod_best = GradientBoostingClassifier(learning_rate=0.01, n_estimators=1000,␣
 ↪max_depth=2, random_state=seed)
bo_err = 1 - cross_val_score(bo_mod_best, X_train, y_train, scoring='accuracy',␣
 ↪cv=10).mean()

bo_mod_best.fit(X_train, y_train)
bo_auc = roc_auc_score(y_train, bo_mod_best.predict(X_train))

error_rates['Boosting'] = [bo_err, bo_auc]
```

## Model Evaluation

```
[24]: errors_df = pd.DataFrame(error_rates, index=['10-fold CV Error Rate', 'ROC/
       ↪AUC']).T
      errors_df.sort_values(by='10-fold CV Error Rate')
```

```
[24]:                            10-fold CV Error Rate   ROC/AUC
      Random Forest                           0.197844  0.870774
      Boosting                                0.202579  0.839971
      Elastic Net Regression                  0.203245  0.820179
      Logistic Regression                     0.203916  0.815254
      Bagging                                 0.205304  0.992426
      Decision Tree                           0.219549  0.790820
      Naive Bayes                             0.265577  0.743125
```

Given that random forest has the lowest 10-fold cv error rate and the second highest ROC/AUC score, we will choose random forest as the best model for the gss data. Although random forest model does not yield the best ROC/AUC score, its ability to differentiate 0's as 0 and 1's as 1 is well enough among our selection of models. Bagging model performs with the greatest ROC/AUC score but it suffers from high variance, which gives it a less desirable error rate. Boosting also performs with low error rate, but it suffers from a lower ROC/AUC score, which means it makes more false positive and negatives in prediction. Also, boosting has a higher risk of overfitting, hence, we choose random forest as the best model to be implemented.

```
[30]: test_err = 1 - cross_val_score(rf_mod_best, X_test, y_test, scoring='accuracy',␣
       ↪cv=10).mean()
      test_auc = roc_auc_score(y_test, rf_mod_best.predict(X_test))

      print(f'When applied to the test set, the best random forest model yields␣
       ↪10-fold cv error rate of {test_err} \nand ROC/AUC score of {test_auc}')
```

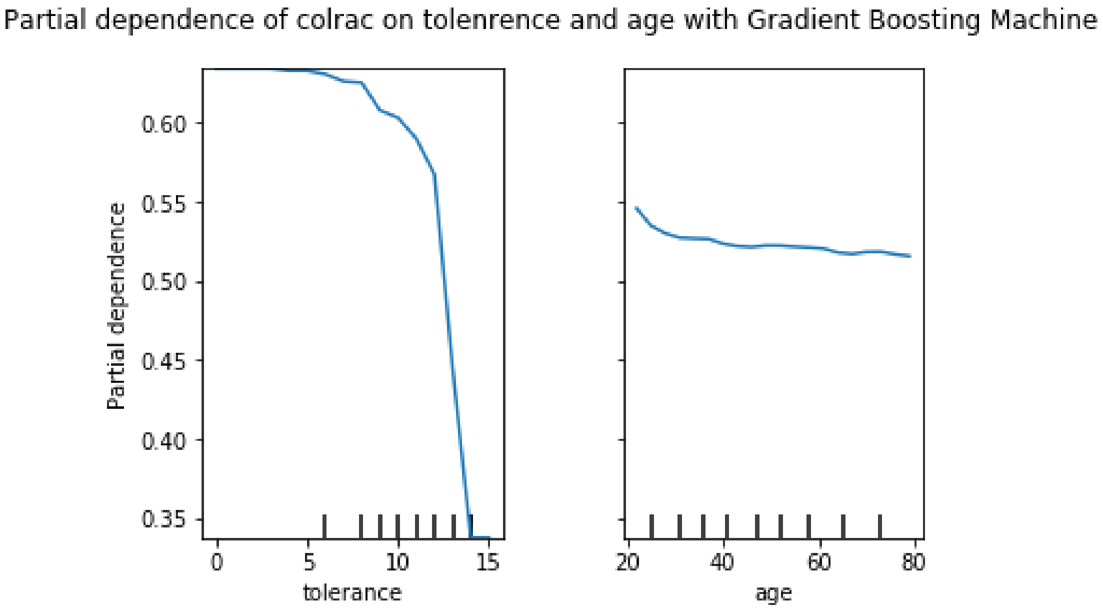
```
When applied to the test set, the best random forest model yields 10-fold cv
error rate of 0.20681632653061222
and ROC/AUC score of 0.7826961271102284
```

The random forest model falls in test error rate by only about 0.008, which is still lower than the training error of decision tree and naive bayes, hence maintains good predictive ability. The ROC/AUC score falls about 0.09 from train to test dataset. Given that the score is within 0.7-0.8, the random forest model maintains a fair power of avoiding false positives/negatives. Therefore, we can conclude that the random forest model does have good generalizability.

## Partial Dependence Plots

```
[33]: features = ['tolerance', 'age']
      plot_partial_dependence(rf_mod_best, X_train, features,
                              n_jobs=3, grid_resolution=20)
      fig = plt.gcf()
```

```
fig.suptitle('Partial dependence of colrac on tolerence and age with Gradient␣
↪Boosting Machine')
fig.subplots_adjust(wspace=0.4, hspace=0.3)
```



The PDPs shows that in our random forest model, there exists an negative non-linear relationship between political tolerance and the probability that the racist professor is allowed to teach. The higher value the tolerance variable takes, the less likely the professor is deemed "allowed to teach." The slope of the PDP on tolerence is very steep, indicating that the tolerance variable has great influence on the prediction of colrac. The PDP of age, however, has a weak, negative, and seemingly linear relationship with the prediction. This shows that age has negative effect on colrac but not as strong as tolerance. The higher the professor's age, the less likely the professor is allowed to teach.

(p.s. I have trouble finding the annotation of the 'tolerance' variable in GSS Data Explorer…therefore I can't interpret the tolerance PDP concretely.)

[31]:
```
rf_result = pd.DataFrame({'feature': X_train.columns, 'importance': rf_mod_best.
↪feature_importances_})
rf_result.sort_values(by='importance', ascending=False, inplace=True)
rf_result.head(10)
```

[31]:
```
         feature  importance
32     tolerance    0.241531
10       colmslm    0.145349
6         colath    0.101771
8         colmil    0.089509
7         colcom    0.042849
0            age    0.030211
```

```
12  egalit_scale    0.022110
18      income06    0.020313
23          pray    0.019338
26  science_quiz    0.016042
```

Looking at the feature importances in the random forest model, tolerance has the highest importance score, confirming its steep slope in PDP. The age variable has a much lower feature importance score, but there is still an effect, it's the seventh importance predictor in the random forest model.

```
[ ]:
```