Mingtao_Gao_HW5

March 1, 2020

0.1 Conceptual: Cost functions for classification trees

Classification error rate, as an alternative of RSS, measures the fraction of the training observations that do not belong to the most common class for the target attribute. On the other hand, based on how the gini index and cross-entropy are calculated, they are direct measurements for node purity. Gini index is a measure of total variance across all different classes for the target attribute. Cross-entropy takes a similar numerical approach as the gini index. Thus, both of them will take on a small value if the node is close to pure, which means the node predominantly contains observations with one class. Compared to classification error rate, **gini index and cross-entropy are more sensitive to node purity, which works better for tree-growing**. During tree-growing process, either the Gini index or the cross-entropy is good to measure the quality of a particular split, because we want the tree to make the most accurate prediction and the quality of spliting needs to be measured precisely. However, **during tree-pruning process, classification error rate would be better approach** to achieve higher prediction accuracy, because it is less sensitive to node purity and it can prevent over-fitting caused by gini index and cross-entropy measures used when growing the tree. Reference: * Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2014. An Introduction to Statistical Learning: with Applications in R. Springer Publishing Company, Incorporated.

0.2 Application: Predicting attitudes towards racist college professors

[1]:
```
import numpy as np
from statistics import mean
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import nltk
from time import time
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV,␣
 ↪ElasticNet, SGDClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier,␣
 ↪GradientBoostingClassifier
from sklearn.metrics import accuracy_score, roc_auc_score, roc_curve
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```python
seed = 1234
```

[3]:
```python
train = pd.read_csv('data/gss_train.csv')
test = pd.read_csv('data/gss_test.csv')
```

[4]:
```python
# Split data into X and Y
X_train = train.loc[:, train.columns != 'colrac'].values
y_train = train['colrac'].values
X_test = test.loc[:, test.columns != 'colrac'].values
y_test = test['colrac'].values
```

[5]:
```python
# Dict to store evaluation and compare across models
auc_performance = {}
err_rate = {}
```

[6]:
```python
def evaluate_model(model, X, Y, name, performance):
    '''
    Function used in this question to evaluate a model's performance on test set
    Inputs:
        model - the trained model object
        X - dataset
        Y - response data
        name - string, the name of the model
        error_rate - a dictionary that stores the error rate of each model
    Outputs:
        Test error rate and Receiver Operating Characteristic(ROC) curve with␣
↪Area Under Curve(AUC) calculated
    '''
    # Predict test set with model
    y_pred = model.predict(X)
    y_pred_prob = model.predict_proba(X)[:, 1]

    # Calculate and print the test error rate of the model
    test_err = round(1 - accuracy_score(Y, y_pred), 4)
    performance[name] = [test_err]

    # Calculate AUC value
    logit_roc_auc = round(roc_auc_score(Y, y_pred), 4)
    performance[name] += [logit_roc_auc]

    # Draw ROC curve
    fpr, tpr, thresholds = roc_curve(Y, y_pred_prob)
    plt.plot(fpr, tpr, label='{} (AUC = {})'.format(name, logit_roc_auc))
    plt.plot([0, 1], [0, 1],'r--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
```

```
    plt.ylabel('True Positive Rate')
    plt.title('ROC curve for {} Model'.format(name))
    plt.legend(loc="lower right")
    plt.show()
```

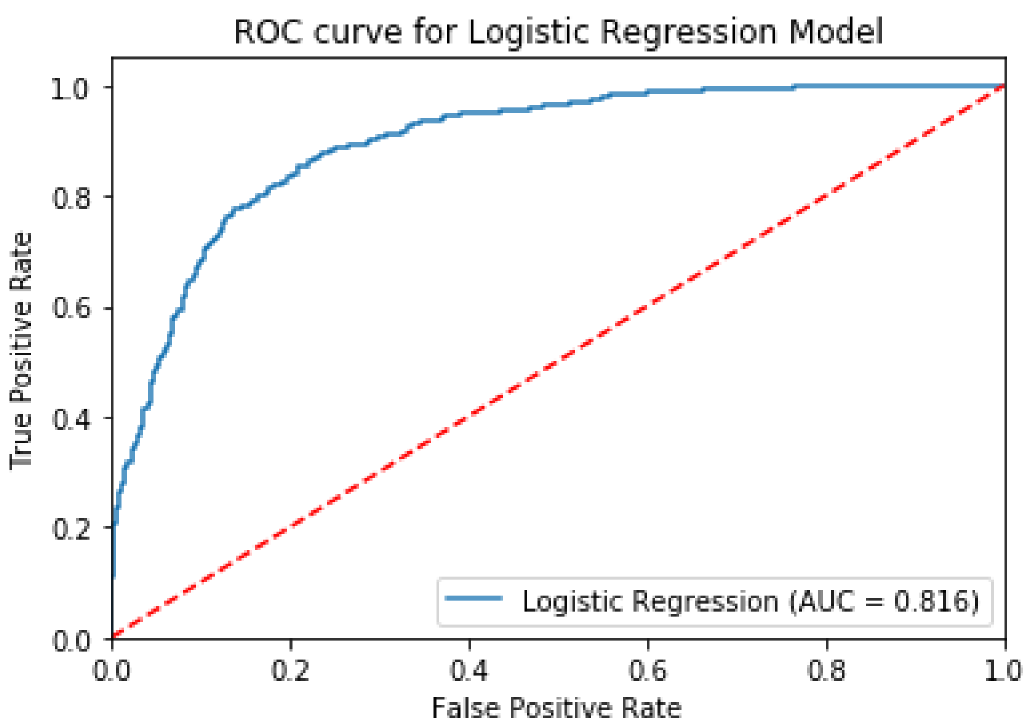
0.2.1 Logistic Regression

[7]:
```
# Logistic Regression
logre = LogisticRegressionCV(cv=10, random_state=seed).fit(X_train, y_train)
```

[8]:
```
# Logistic Regression - CV error rate
kf = model_selection.KFold(n_splits=10, shuffle=True, random_state=seed)
logre_cv_errs = model_selection.cross_val_score(LogisticRegression(), X_train,␣
 ↪y_train,
                                                scoring="accuracy", n_jobs=1,␣
 ↪cv=kf)
err_rate['Logistic Regression'] = 1 - logre_cv_errs.mean()
err_rate['Logistic Regression']
```

[8]: 0.2005056076484648

[9]:
```
# Logistic regression - ROC/AUC
evaluate_model(logre, X_train, y_train, 'Logistic Regression', auc_performance)
```



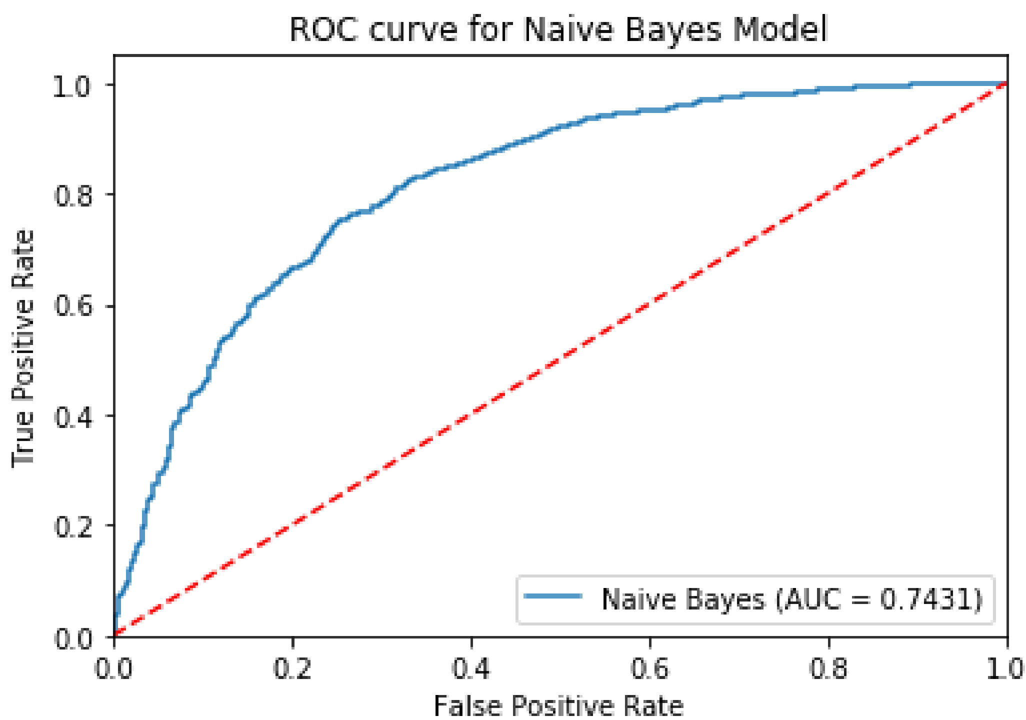
0.2.2 Naive Bayes

[10]:
```
# Naive Bayes
gnb = GaussianNB()
gnb = gnb.fit(X_train, y_train)
```

[11]:
```
# Naive Bayes - CV error rate
gnb_cv_errs = model_selection.cross_val_score(gnb, X_train, y_train,␣
↪scoring="accuracy", cv=kf)
err_rate['Naive Bayes'] = 1 - gnb_cv_errs.mean()
err_rate['Naive Bayes']
```

[11]:
```
0.26222651222651217
```

[12]:
```
# Naive Bayes - AUC
evaluate_model(gnb, X_train, y_train, 'Naive Bayes', auc_performance)
```



0.2.3 Elastic Net Regression

[21]:
```
# Elastic Net Regression - hyperparameter tuning
parametersGrid = {"alpha": [0.0001, 0.001, 0.01, 0.1, 1, 10, 100],
                  "l1_ratio": np.arange(0.0, 1.0, 0.1)}
eNet = ElasticNet()
```

```
grid = model_selection.GridSearchCV(eNet, parametersGrid,␣
↪scoring='neg_mean_squared_error', cv=kf)
eNet = grid.fit(X_train, y_train)
grid.best_params_
```

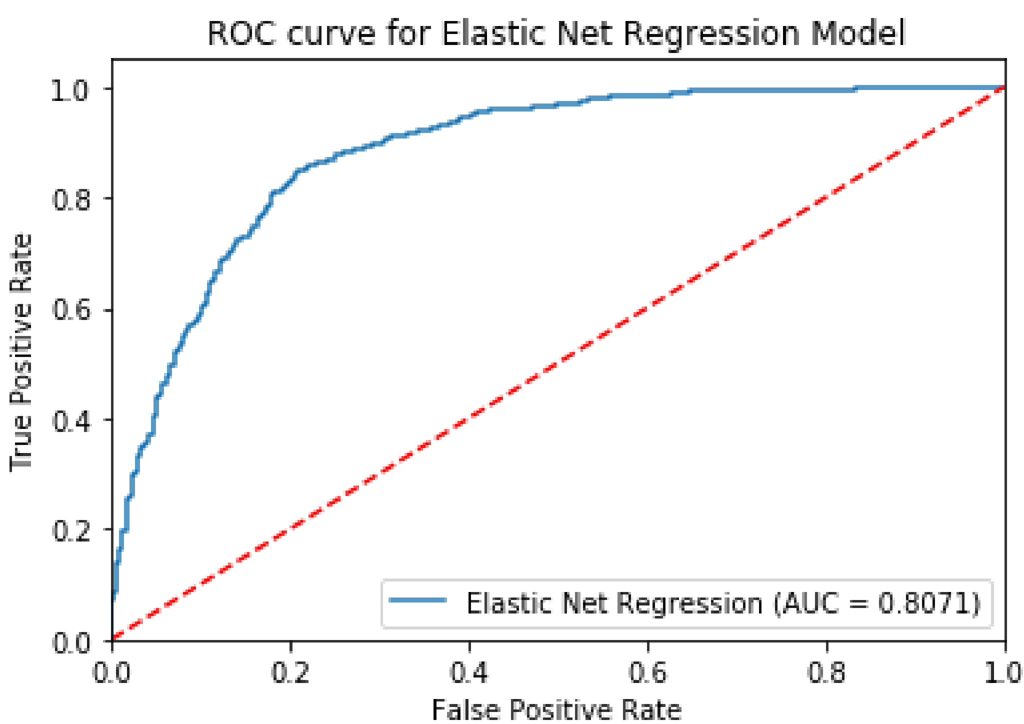
[21]: {'alpha': 0.01, 'l1_ratio': 0.2}

[22]:
```
# Elastic Net Regression
eNet_tuned = SGDClassifier(loss='log', penalty='elasticnet',
                           alpha=grid.best_params_['alpha'],
                           l1_ratio=grid.best_params_['l1_ratio'])
eNet_tuned = eNet_tuned.fit(X_train, y_train)
```

[23]:
```
# Elastic Net - CV error rate
eNet_cv_errs = model_selection.cross_val_score(eNet_tuned, X_train, y_train,␣
↪scoring="accuracy", cv=kf)
err_rate['Elastic Net Regression'] = 1 - eNet_cv_errs.mean()
err_rate['Elastic Net Regression']
```

[23]: 0.26688269902555617

[24]:
```
# Elastic Net - ROC/AUC
evaluate_model(eNet_tuned, X_train, y_train, 'Elastic Net Regression',␣
↪auc_performance)
```



0.2.4 Decision Tree

[25]:
```
# Decision Tree - hyperparameter tuning
param_grid = {"criterion": ["gini", "entropy"],
              "min_samples_split": [2, 10, 20],
              "max_depth": [None, 2, 5, 10],
              "min_samples_leaf": [1, 5, 10],
              "max_leaf_nodes": [None, 5, 10, 20],
              }
dt = DecisionTreeClassifier()
grid = model_selection.GridSearchCV(dt, param_grid=param_grid, cv=kf,␣
 ↪scoring="accuracy")
grid = grid.fit(X_train, y_train)
grid.best_params_
```

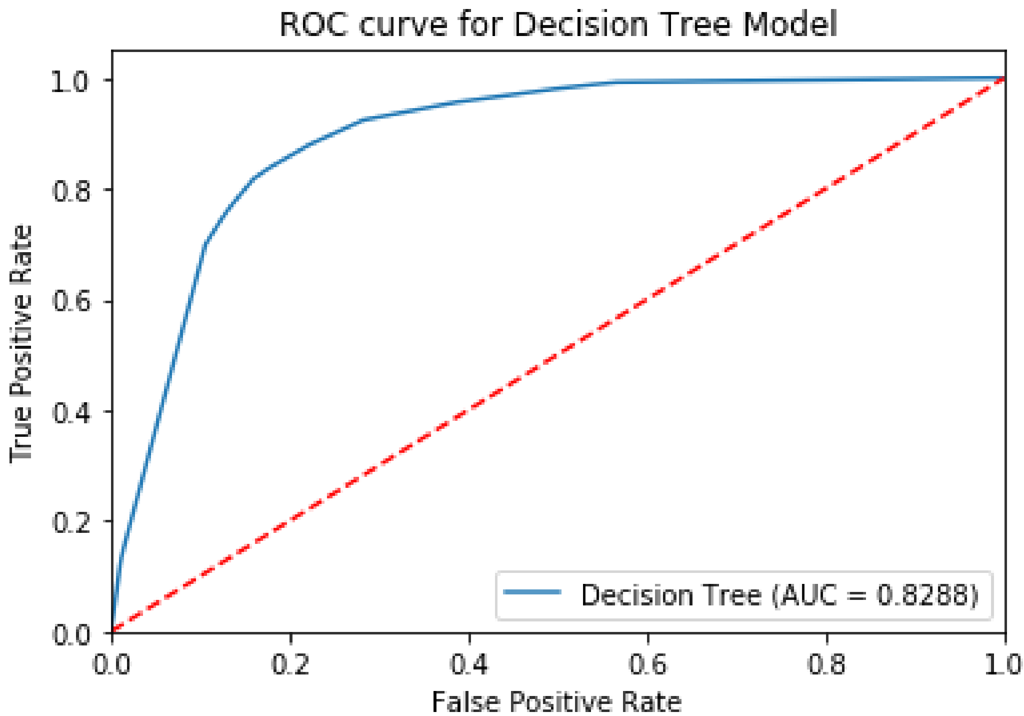
[25]:
```
{'criterion': 'gini',
 'max_depth': 10,
 'max_leaf_nodes': 20,
 'min_samples_leaf': 5,
 'min_samples_split': 20}
```

[26]:
```
# Decision Tree
dt_tuned = DecisionTreeClassifier(criterion=grid.best_params_['criterion'],
                                  max_depth=grid.best_params_['max_depth'],
                                  max_leaf_nodes=grid.
 ↪best_params_['max_leaf_nodes'],
                                  min_samples_leaf=grid.
 ↪best_params_['min_samples_leaf'],
                                  min_samples_split=grid.
 ↪best_params_['min_samples_split'])
dt_tuned = dt_tuned.fit(X_train, y_train)
```

[27]:
```
# Decision Tree - CV error rate
dt_cv_errs = model_selection.cross_val_score(dt_tuned, X_train, y_train,␣
 ↪scoring="accuracy", cv=kf)
err_rate['Decision Tree'] = 1 - dt_cv_errs.mean()
err_rate['Decision Tree']
```

[27]:
```
0.2174986210700497
```

[28]:
```
evaluate_model(dt_tuned, X_train, y_train, 'Decision Tree', auc_performance)
```



0.2.5 Bagging

[29]:
```
# Bagging - hyperparameter tuning
param_grid = {"n_estimators":[10, 100, 1000],
              "max_samples" : [0.05, 0.1, 0.2, 0.5]}
bagging = BaggingClassifier()
grid = model_selection.GridSearchCV(bagging, param_grid=param_grid, cv=kf,␣
 ↪scoring="accuracy")
grid = grid.fit(X_train, y_train)
grid.best_params_
```

[29]:
```
{'max_samples': 0.5, 'n_estimators': 1000}
```

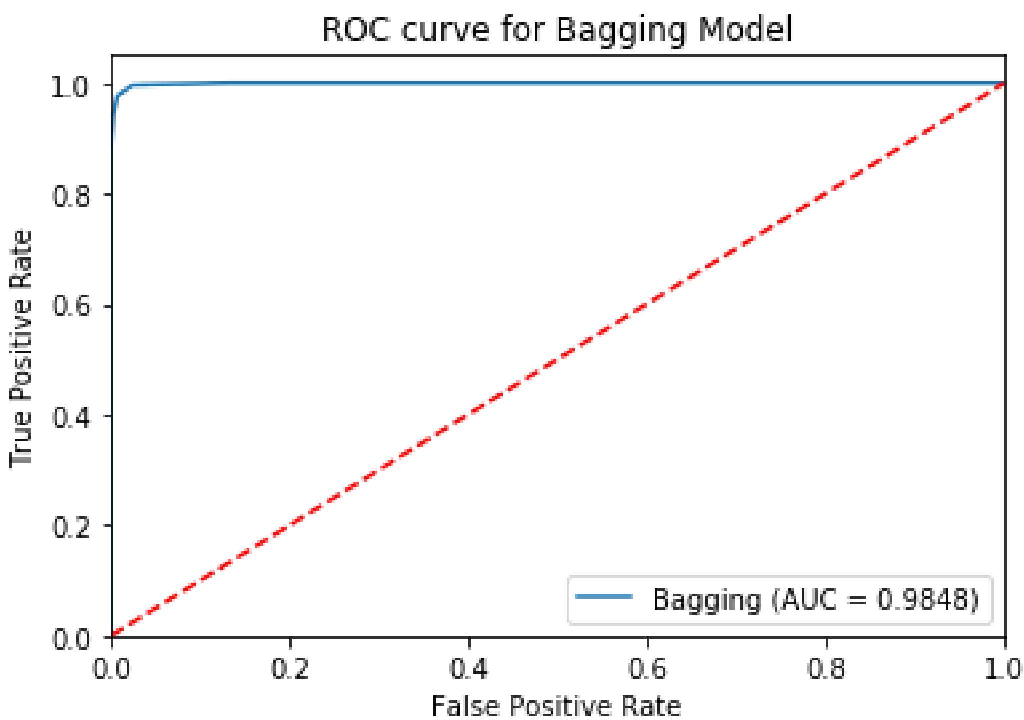
[30]:
```
# Bagging
bagging_tuned = BaggingClassifier(base_estimator=DecisionTreeClassifier(),
                                  n_estimators=grid.
 ↪best_params_['n_estimators'],
                                  max_samples=grid.best_params_['max_samples'])
bagging_tuned = bagging.fit(X_train, y_train)
```

[31]:
```
# Bagging - CV error rate
bagging_cv_errs = model_selection.cross_val_score(bagging_tuned, X_train,␣
 ↪y_train, cv=kf, scoring='accuracy')
```

```
err_rate['Bagging'] = 1 - bagging_cv_errs.mean()
err_rate['Bagging']
```

[31]: 0.23576025004596435

[32]:
```
evaluate_model(bagging_tuned, X_train, y_train, 'Bagging', auc_performance)
```



0.2.6 Random Forest

[33]:
```
# Random Forest - hyperparameter tuning
param_grid = {"n_estimators": [10, 100, 1000],
              "max_features": ['sqrt', 'log2'],
              "min_samples_leaf": [1, 5, 10],
              'max_depth': [10, 30, 50, 70, 90, None]}
rf = RandomForestClassifier()
grid = model_selection.GridSearchCV(rf, param_grid=param_grid, cv=kf,␣
 ↪scoring="accuracy")
grid = grid.fit(X_train, y_train)
grid.best_params_
```

[33]:
```
{'max_depth': 30,
 'max_features': 'sqrt',
 'min_samples_leaf': 1,
```

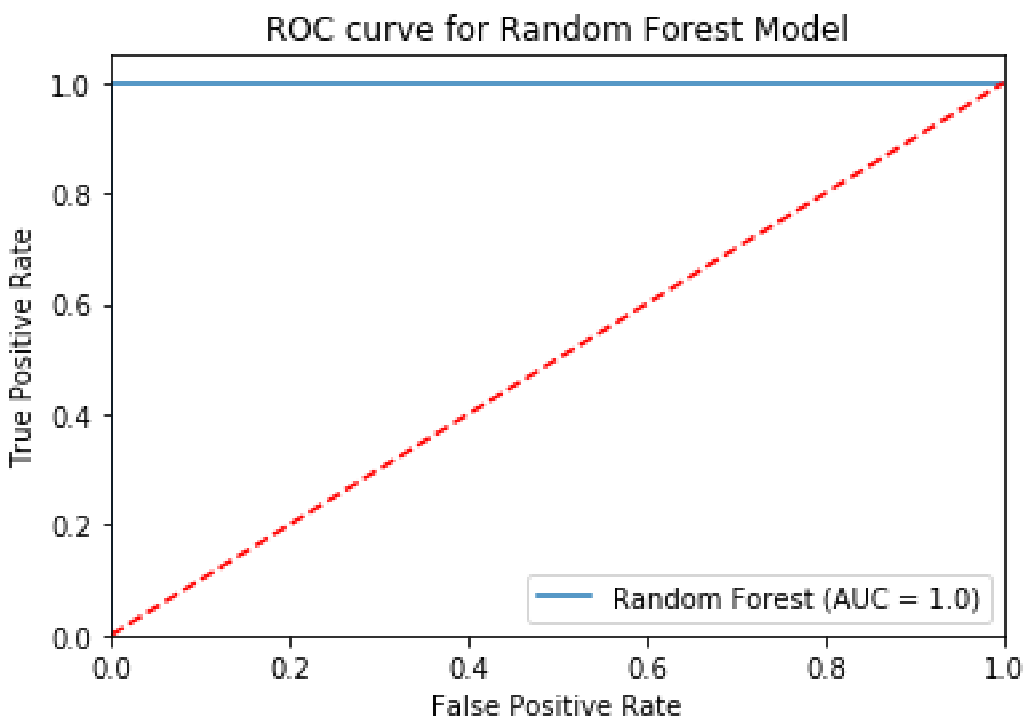
'n_estimators': 1000}

[34]:
```
# Random Forest
rf_tuned = RandomForestClassifier(n_estimators=grid.
 ↪best_params_['n_estimators'],
                                  max_features=grid.
 ↪best_params_['max_features'],
                                  min_samples_leaf=grid.
 ↪best_params_['min_samples_leaf'],
                                  max_depth=grid.best_params_['max_depth'])
rf_tuned = rf_tuned.fit(X_train, y_train)
```

[35]:
```
# Random Forest - CV error rate
rf_cv_errs = model_selection.cross_val_score(rf_tuned, X_train, y_train, cv=kf,␣
 ↪scoring='accuracy')
err_rate['Random Forest'] = 1 - rf_cv_errs.mean()
err_rate['Random Forest']
```

[35]: 0.19646534289391426

[36]:
```
# Random Forest - ROC/AUC
evaluate_model(rf_tuned, X_train, y_train, 'Random Forest', auc_performance)
```



0.2.7 Boosting

[37]:
```
# Boosting - hyperparameter tuning
param_grid = {"learning_rate": [0.001, 0.01, 0.1],
              "n_estimators": [10, 100, 1000],
              "max_depth": [10, 30, 50, 70, 90, None]}
boosting = GradientBoostingClassifier()
grid = model_selection.GridSearchCV(boosting, param_grid=param_grid, cv=kf,␣
 ↪scoring='accuracy')
grid = grid.fit(X_train, y_train)
grid.best_params_
```

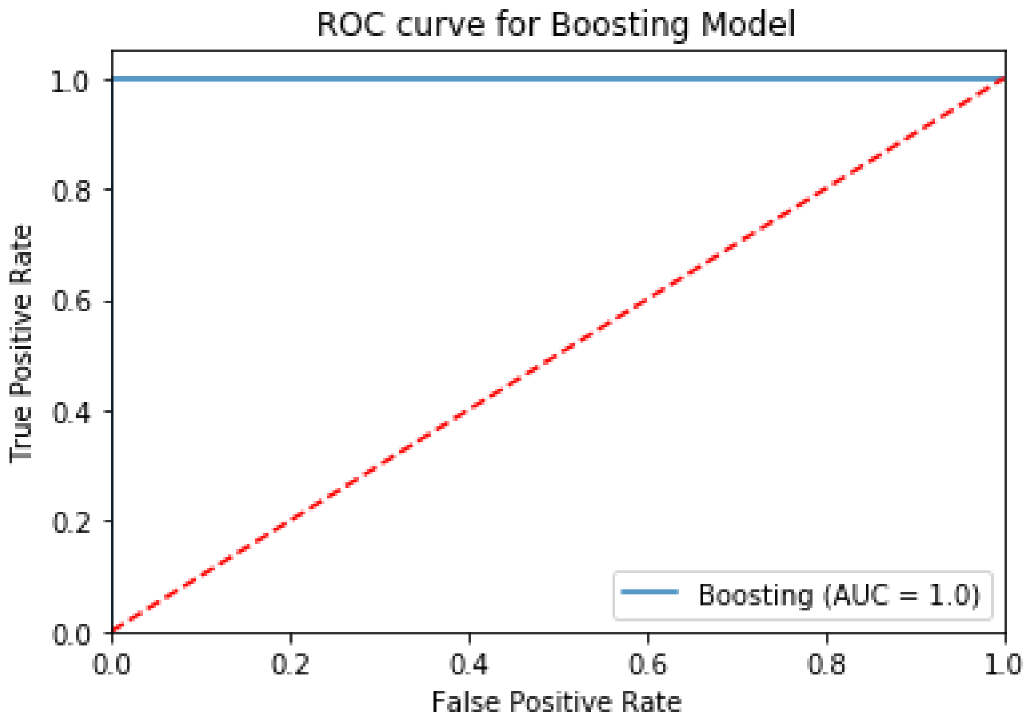
[37]:
```
{'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 1000}
```

[39]:
```
# Boosting
boosting_tuned = GradientBoostingClassifier(n_estimators=grid.
 ↪best_params_['n_estimators'],
                                            learning_rate=grid.
 ↪best_params_['learning_rate'],
                                            max_depth=grid.
 ↪best_params_['max_depth'])
boosting_tuned = boosting_tuned.fit(X_train, y_train)
```

[40]:
```
# Boosting - CV error rate
boosting_cv_errs = model_selection.cross_val_score(boosting_tuned, X_train,␣
 ↪y_train, cv=kf, scoring='accuracy')
err_rate['Boosting'] = 1 - boosting_cv_errs.mean()
err_rate['Boosting']
```

[40]:
```
0.20864129435558
```

[41]:
```
# Boosting - ROC/AUC
evaluate_model(boosting_tuned, X_train, y_train, 'Boosting', auc_performance)
```



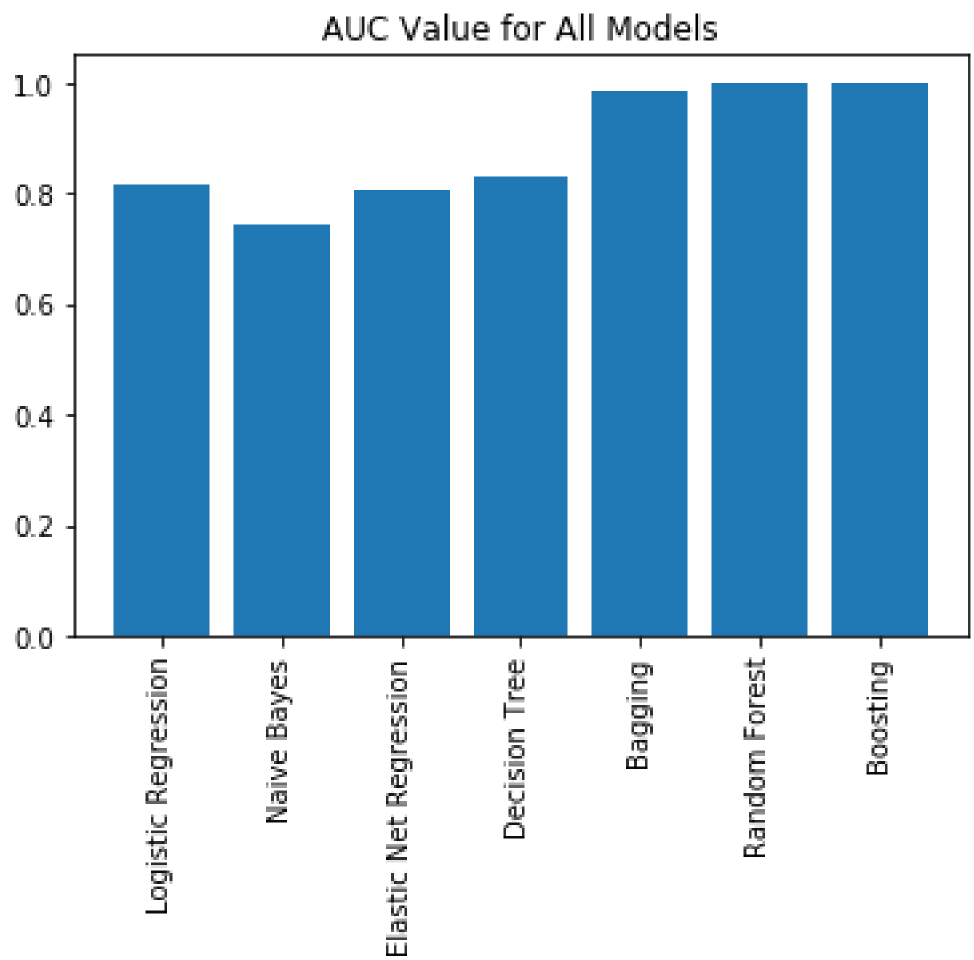
0.2.8 Comparison and Find the Best Model

[51]:
```
pd.DataFrame(auc_performance).T
```

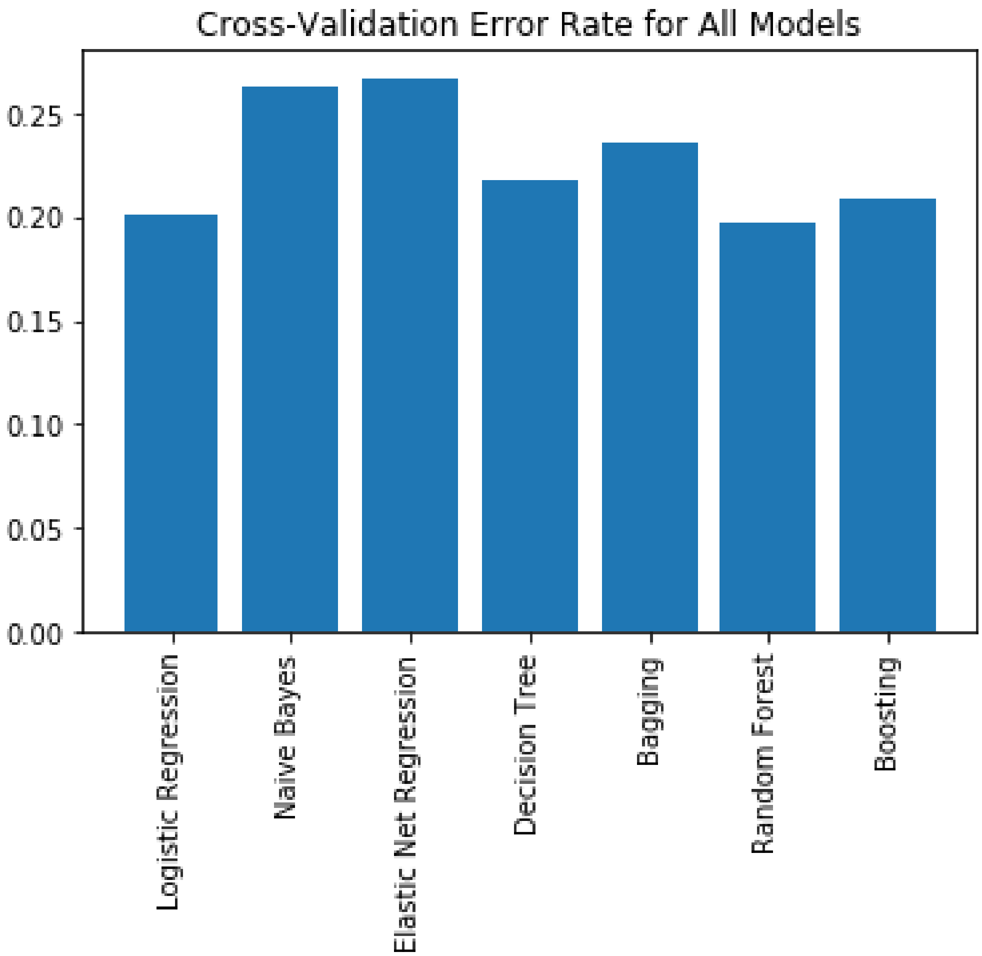
[51]:

	0	1
Logistic Regression	0.1836	0.8160
Naive Bayes	0.2575	0.7431
Elastic Net Regression	0.1890	0.8071
Decision Tree	0.1687	0.8288
Bagging	0.0156	0.9848
Random Forest	0.0000	1.0000
Boosting	0.0000	1.0000

[61]:
```
plt.bar(range(len(auc_performance)), [i[1] for i in auc_performance.values()],␣
 ↪align='center')
plt.xticks(range(len(auc_performance)), list(auc_performance.keys()),␣
 ↪rotation=90)
plt.title('AUC Value for All Models')
plt.show()
```



[62]:
```
plt.bar(range(len(err_rate)), list(err_rate.values()), align='center')
plt.xticks(range(len(err_rate)), list(err_rate.keys()), rotation=90)
plt.title('Cross-Validation Error Rate for All Models')
plt.show()
```



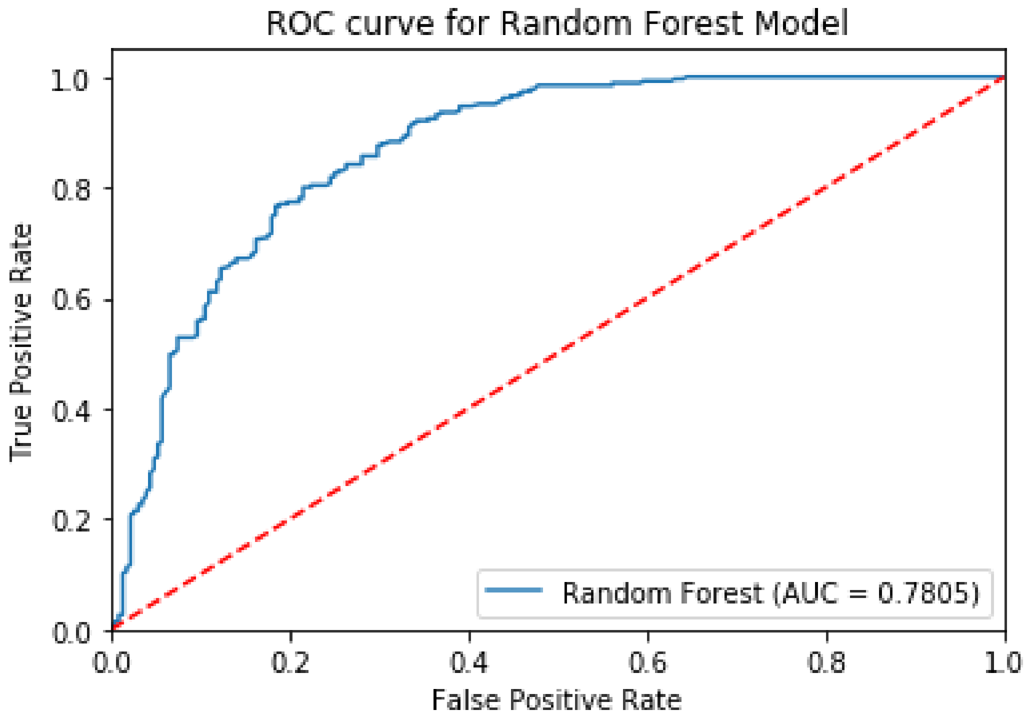
From above evaluation of the performances of all 7 models, we can see bagging, random forest, and boosting model display very high AUC values and low CV error rates. Comparing across all models, **Random Forest** generated the best prediction for the training dataset. Random Forest model has the lowest CV error rate, which means it achieved the best performance on predicting the training dataset. Besides, it has the highest ROC/AUC, which is 1. When AUC is equal to 1, it means the predictions are 100% correct, which very likely to overfit the dataset. Thus, based on the result, bagging, random forest, and boosting models are very likely to overfit the data and achieving high classification error when predicting the testing dataset.

0.2.9 Evaluate the Best Model

[79]:
```
y_pred = rf_tuned.predict(X_test)
test_err = 1 - accuracy_score(y_test, y_pred)
test_err
```

[79]:
```
0.21095334685598377
```

[80]:
```
rf_evaluation = {}
evaluate_model(rf_tuned, X_test, y_test, 'Random Forest', rf_evaluation)
```

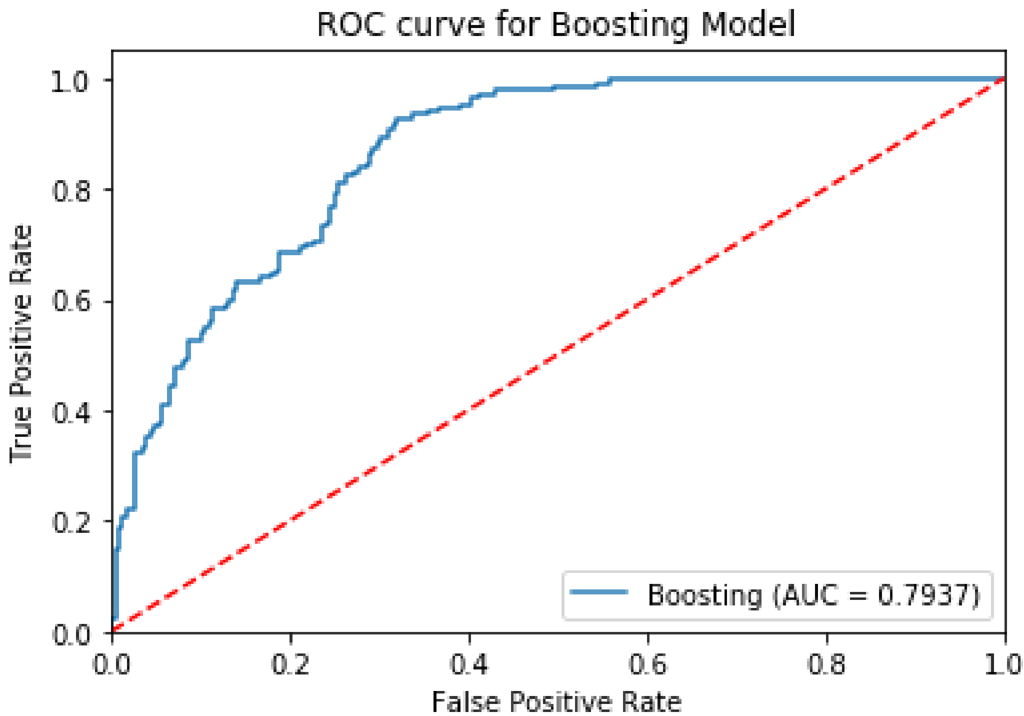


As we discussed above, the random forest model overfits the training dataset, which leads to poor performance on the testing dataset. It does not generalize well. The CV error for training set of Random Forest model is 0.1965 and the classification error for testing set is 0.211, while the AUC for training set is 1 and the AUC for testing set is only 0.7805. It shows that this model overfitted the dataset. We can also compare its performance with Boosting model, which has the same AUC value as Random Forest, but a higher CV error rate, which means it's less likely to overfit the training dataset.

[87]:
```
y_pred = boosting_tuned.predict(X_test)
test_err = 1 - accuracy_score(y_test, y_pred)
test_err
```

[87]: 0.19878296146044627

[88]:
```
evaluate_model(boosting_tuned, X_test, y_test, 'Boosting', rf_evaluation)
```



From above analysis, we found boosting indeed performed better on the testing dataset with a smaller error and higher AUC value. Thus, when we choose which model to use for the dataset, we need to be careful that the smallest error rate on testing data does not necessarily mean it is the best model. Such model may lead to overfitting and thus poor performance on the testing dataset.