Xu_Weijie_HW5

March 1, 2020

Name: Weijie Xu
CNetID: weijiexu
Student ID: 12245277

1 Conceptual: Cost functions for classification trees

1.1 Task 1

When growing the tree, we may prefer cross-entropy and gini index to classifiction error. This is because the estimation procedure to build a tree is to split the data on the feature that gives rise to the largest reduction in error. According to this logic, the point of the highest interest is not only about the absolute value of classification error, but the change of error that a specific feature brings about. To measure this change, cross-entropy and gini index, which evaluate the impurity of a split result, are definitely better metrics.

However, the logic of pruning a tree is a little bit different. Since the goal in this step is to avoid overfitting, the trade-off we are balancing here is between the size of the tree and the performance of prediction. That is, after having built a large and deep tree, we are looking for the smallest sub-tree that still has an acceptable abosolute classfication error. Therefore, during the process of pruning a tree, we would prefer classification error to other evaluation metrics.

2 Application: Predicting attitudes towards racist college professors

2.1 Task 2

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

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.inspection import plot_partial_dependence
from sklearn.exceptions import ConvergenceWarning
from warnings import simplefilter
```

[2]:
```
SEED = 970608

df_train = pd.read_csv('data/gss_train.csv')
df_test = pd.read_csv('data/gss_test.csv')

X_train, X_test = df_train.drop('colrac', axis=1), df_test.drop('colrac',␣
↪axis=1)
y_train, y_test = df_train['colrac'], df_test['colrac']
```

[3]:
```
# Mute ConvergenceWarning
warnings.filterwarnings("ignore", category=ConvergenceWarning, module="sklearn")
simplefilter(action='ignore', category=FutureWarning)
```

2.1.1 a. Logistic Regression

[4]:
```
clf_log = LogisticRegression()
clf_log.fit(X_train, y_train)
```

[4]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

2.1.2 b. Naive Bayes

[5]:
```
clf_nb = GaussianNB()
clf_nb.fit(X_train, y_train)
```

[5]:
```
GaussianNB(priors=None, var_smoothing=1e-09)
```

2.1.3 c. Elastic net regression

[6]:
```
clf_elasticlog = LogisticRegressionCV(cv=10,
                                      l1_ratios=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6,␣
↪0.7, 0.8, 0.9, 1],
                                      penalty='elasticnet',
                                      solver='saga',
                                      random_state=SEED)
clf_elasticlog.fit(X_train, y_train)
```

[6]:
```
LogisticRegressionCV(Cs=10, class_weight=None, cv=10, dual=False,
                     fit_intercept=True, intercept_scaling=1.0,
                     l1_ratios=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1],
                     max_iter=100, multi_class='warn', n_jobs=None,
                     penalty='elasticnet', random_state=970608, refit=True,
                     scoring=None, solver='saga', tol=0.0001, verbose=0)
```

[7]:
```
clf_elasticlog.l1_ratio_
```

[7]:
```
array([0.2])
```

[8]:
```
clf_elasticlog = LogisticRegression(penalty='elasticnet',
                                    l1_ratio=0.2,
                                    solver='saga',
                                    random_state=SEED)
clf_elasticlog.fit(X_train, y_train)
```

[8]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=0.2, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='elasticnet',
                   random_state=970608, solver='saga', tol=0.0001, verbose=0,
                   warm_start=False)
```

2.1.4 d. Decision tree (CART)

Since there's no package in python to do cost complexity pruning, here we tuned "max_depth", "min_samples_split", and "min_samples_leaf" by cross validation to avoid overfitting.

[9]:
```
# Hyperparameter tuning
grid_dtree = {'max_depth': [int(x) for x in np.linspace(10, 110, num = 11)],
              'min_samples_split': [2, 5, 10],
              'min_samples_leaf': [1, 2, 4]}
clf_dtree = DecisionTreeClassifier(random_state=SEED)
dtree_grid = GridSearchCV(estimator=clf_dtree,
                          param_grid=grid_dtree,
                          cv=10,
                          n_jobs = -1)
dtree_grid.fit(X_train, y_train)
dtree_grid.best_params_
```

[9]:
```
{'max_depth': 10, 'min_samples_leaf': 4, 'min_samples_split': 2}
```

[10]:
```
# Fit the model with the optimal hyperparameters.
clf_dtree = DecisionTreeClassifier(min_samples_split=2,
                                   min_samples_leaf=4,
                                   max_depth=10,
                                   random_state=SEED)
```

```
clf_dtree.fit(X_train, y_train)
```

[10]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=10,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=4, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=970608, splitter='best')
```

2.1.5 e. Bagging

The number of base estimators is the hyperparameter to be tuned.

[11]:
```
# Hyperparameter tuning
grid_bag = {'n_estimators': np.arange(1,101)}
clf_bag = BaggingClassifier(random_state=SEED)
bag_grid = GridSearchCV(estimator=clf_bag,
                        param_grid=grid_bag,
                        cv=10,
                        n_jobs = -1)
bag_grid.fit(X_train, y_train)
bag_grid.best_params_
```

[11]:
```
{'n_estimators': 84}
```

[12]:
```
# Fit the model with the optimal hyperparameter.
clf_bag = BaggingClassifier(n_estimators=84,
                            random_state=SEED)
clf_bag.fit(X_train, y_train)
```

[12]:
```
BaggingClassifier(base_estimator=None, bootstrap=True, bootstrap_features=False,
                  max_features=1.0, max_samples=1.0, n_estimators=84,
                  n_jobs=None, oob_score=False, random_state=970608, verbose=0,
                  warm_start=False)
```

2.1.6 f. Random forest

For the sake of computational efficiency, here we use RandomizedSearch instead of exhaustive GridSearch to tune hyperparameters.

[13]:
```
# Hyperparameter tuning
random_grid_rf = {'n_estimators': np.arange(1,101),
                  'max_features': np.arange(2, len(list(X_train.columns)) + 1),
                  'max_depth': [int(x) for x in np.linspace(10, 110, num = 11)],
                  'min_samples_split': [2, 5, 10],
                  'min_samples_leaf': [1, 2, 4]
                  }
clf_rf = RandomForestClassifier(random_state=SEED)
```

```
rf_random = RandomizedSearchCV(estimator=clf_rf,
                               param_distributions=random_grid_rf,
                               n_iter=100,
                               cv=10,
                               random_state=SEED, n_jobs = -1)
rf_random.fit(X_train, y_train)
rf_random.best_params_
```

[13]:
```
{'n_estimators': 68,
 'min_samples_split': 5,
 'min_samples_leaf': 4,
 'max_features': 9,
 'max_depth': 40}
```

[14]:
```
# Fit the model with the optimal hyperparameter.
clf_rf = RandomForestClassifier(n_estimators=68,
                                max_features=9,
                                max_depth=40,
                                min_samples_split=5,
                                min_samples_leaf=4,
                                random_state=SEED)
clf_rf.fit(X_train, y_train)
```

[14]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=40, max_features=9, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=4, min_samples_split=5,
                       min_weight_fraction_leaf=0.0, n_estimators=68,
                       n_jobs=None, oob_score=False, random_state=970608,
                       verbose=0, warm_start=False)
```

2.1.7 g. Boosting

For the sake of computational efficiency, here we use RandomizedSearchCV instead of exhaustive GridSearchCV to tune hyperparameters.

[15]:
```
# Hyperparameter tuning
random_grid_gb = {'n_estimators': np.arange(1,101),
                  'learning_rate': np.linspace(0.001, 0.01, num=10),
                  'max_depth': np.arange(1,5),
                  }
clf_gb = GradientBoostingClassifier(random_state=SEED)
gb_random = RandomizedSearchCV(estimator=clf_gb,
                               param_distributions=random_grid_gb,
                               n_iter=100,
                               cv=10,
                               random_state=SEED, n_jobs = -1)
```

```
gb_random.fit(X_train, y_train)
gb_random.best_params_
```

[15]:
```
{'n_estimators': 86, 'max_depth': 4, 'learning_rate': 0.01}
```

[16]:
```
# Fit the model with the optimal hyperparameter.
clf_gb = GradientBoostingClassifier(n_estimators=86,
                                    learning_rate=0.01,
                                    max_depth=4,
                                    random_state=SEED)
clf_gb.fit(X_train, y_train)
```

[16]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.01, loss='deviance', 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, n_estimators=86,
                           n_iter_no_change=None, presort='auto',
                           random_state=970608, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

2.2 Task 3

The cross-validation error rate (or "accuracy" in the plot) and the ROC-AUC score are reported in the table below. According to the result of model evaluation, we would choose Random Forest as the model that has the best performance on training data because it has the lowest cross-validation error (0.189) and the highest auc score (0.884).

[17]:
```
def compute_cv_err(X, y, model, model_name, cv_err_dict):
    '''
    Compute cross-validated error rate
    '''
    cv_scores = cross_val_score(model, X, y, cv=10)
    avg_accuracy = cv_scores.mean()
    cv_err_dict[model_name] = 1 - avg_accuracy
```

[18]:
```
def compute_auc(X, y, model, model_name, auc_dict):
    '''
    Compute AUC
    '''
    cv_scores = cross_val_score(model, X, y, scoring='roc_auc', cv=10)
    avg_accuracy = cv_scores.mean()
    auc_dict[model_name] = avg_accuracy
```

[19]:
```
cv_err_dict, auc_dict = {}, {}
```

[20]:
```
compute_cv_err(X_train, y_train, clf_log, 'Logistics', cv_err_dict)
compute_auc(X_train, y_train, clf_log, 'Logistics', auc_dict)
```

[21]:
```
compute_cv_err(X_train, y_train, clf_nb, 'Naive Bayes', cv_err_dict)
compute_auc(X_train, y_train, clf_nb, 'Naive Bayes', auc_dict)
```

[22]:
```
compute_cv_err(X_train, y_train, clf_elasticlog, 'Elastic Net Regression',␣
↪cv_err_dict)
compute_auc(X_train, y_train, clf_elasticlog, 'Elastic Net Regression',␣
↪auc_dict)
```

[23]:
```
compute_cv_err(X_train, y_train, clf_dtree, 'Decision Tree (CART)', cv_err_dict)
compute_auc(X_train, y_train, clf_dtree, 'Decision Tree (CART)', auc_dict)
```

[24]:
```
compute_cv_err(X_train, y_train, clf_bag, 'Bagging', cv_err_dict)
compute_auc(X_train, y_train, clf_bag, 'Bagging', auc_dict)
```

[25]:
```
compute_cv_err(X_train, y_train, clf_rf, 'Random Forest', cv_err_dict)
compute_auc(X_train, y_train, clf_rf, 'Random Forest', auc_dict)
```

[26]:
```
compute_cv_err(X_train, y_train, clf_gb, 'Boosting', cv_err_dict)
compute_auc(X_train, y_train, clf_gb, 'Boosting', auc_dict)
```

[27]:
```
accurc_dict = {'CV Error': cv_err_dict, 'AUC': auc_dict}
accurc_df = pd.DataFrame(accurc_dict).sort_values(by=['CV Error', 'AUC'])
accurc_df
```

[27]:

	CV Error	AUC
Random Forest	0.189044	0.884267
Logistics	0.207320	0.870311
Elastic Net Regression	0.209360	0.870700
Bagging	0.209420	0.875584
Boosting	0.214798	0.867391
Decision Tree (CART)	0.250696	0.796230
Naive Bayes	0.265553	0.808050

2.3 Task 5

In this task, we evaluate Random Forest, which is the best model selected in the last task, by calculating its cv error rate and auc score on testing data. According the result reported in the table below, the model performes quite well. Although the metrics evaluating prediction performance of this model on testing data has decreased a little bit compared with on training data, this decreasing is very limited. Therefore, we could claim that the Random Forest model generalizes well from training data to testing data.

Also note that, since we use RandomizedSearchCV instead of GridSearchCV to tune the hyperparameters for the sake of computational efficiency, we do not exclude the possibility that other models could have achieved better performance with different hyperparameters.

[28]:
```
cv_err_dict, auc_dict = {}, {}
compute_cv_err(X_test, y_test, clf_rf, 'Random Forest', cv_err_dict)
compute_auc(X_test, y_test, clf_rf, 'Random Forest', auc_dict)
accurc_dict = {'CV Error': cv_err_dict, 'AUC': auc_dict}
accurc_df = pd.DataFrame(accurc_dict).sort_values(by=['CV Error', 'AUC'])
accurc_df
```

[28]:

	CV Error	AUC
Random Forest	0.211017	0.849965

2.4 Task 6

Here we've plotted PDP curves for the Random Forest model selected in the previous tasks. PDP curves below gives us information of the marginal effect of a specific feature (x) on the target response (y).

From the PDP curve below, we can see that, the feature of "tolerance has a positive marginal effect on"colrac". Furthermore, the PDP curve starts to go down when the degree of tolenrance go beyond 6, which means that the marginal effect of"tolerance" on "colrac" starts to decrease after this point. This trend becomes even more rapid after the degree of 12, where the marginal effect of tolerance shrinks strikingly from around 0.6 to around 0.3.

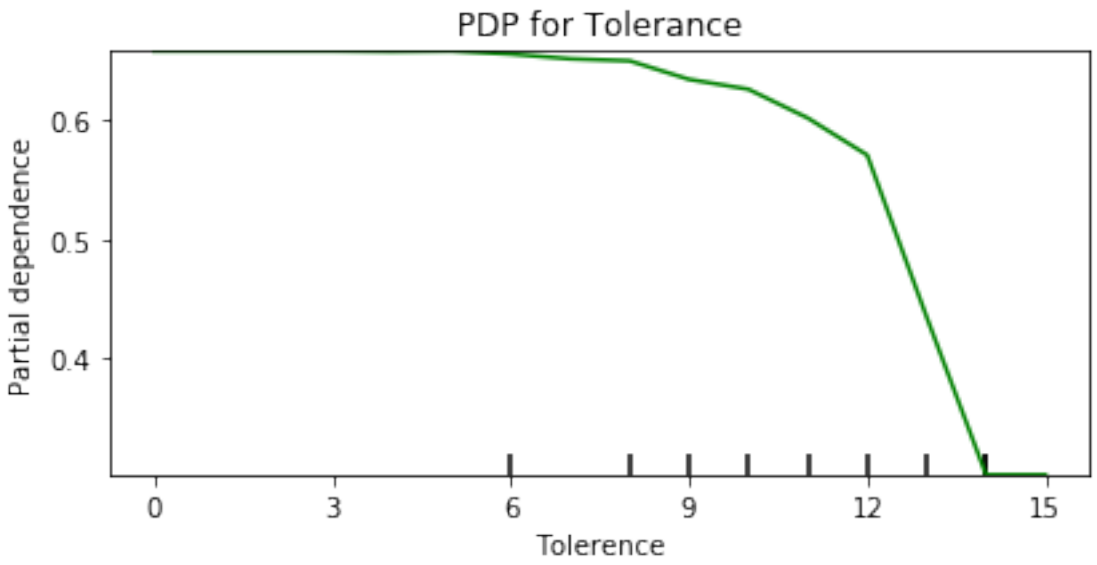
As for the feature of age, as illustrated by the second plot, the marginal effect of this feature on "colrac" is reltively stable, lingering around 0.53. Despite of this fact, there's also slight decreasing trend of marginal effect for "age" as shown by the PDP.

[29]:
```
cols = list(X_train.columns)
idx_tolerence = cols.index('tolerance')
idx_age = cols.index('age')
print('tolerance:', idx_tolerence)
print('age:', idx_age)
```

```
tolerance: 32
age: 0
```

[30]:
```
plot_partial_dependence(clf_rf, X_train, [32])
plt.title('PDP for Tolerance')
plt.xlabel('Tolerence')
```

[30]:
```
Text(0.5, 0, 'Tolerence')
```



[31]:
```
plot_partial_dependence(clf_rf, X_train, [0])
plt.title('PDP for Age')
plt.xlabel('Age')
```

[31]: Text(0.5, 0, 'Age')