Applying Decision Trees and Ensamble Models in Classification and Regression Models

Azhar Chowdhury
Statistical Programmer|Data Scientist[SAS, Python, R]

About the Dataset

This dataset is aimed at the case of customer default payments in Taiwan. From the perspective of risk management, the result of predictive accuracy of the estimated probability of default will be more valuable than the binary result of classification - credible or not credible clients. Because the real probability of default is unknown, this study presented the novel Sorting Smoothing Method to estimate the real probability of default.

Computations Performed

Use of decision trees on a dataset to make a prediction Hyper-parameters tuning for decision trees by using RandomGrid Learning the effectiveness of ensemble algorithms (Random Forest, Adaboost, Extra Trees Classifier, Gradient Boosted Tree)

[1]:
```
# Preparing environment:
import numpy as np
import pandas as pd
import string # to read excel 97-2003 excel files
import seaborn as sns
# to make this notebook's output stable across runs
np.random.seed(123)

# To plot pretty figures
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
```

[2]:
```
#loading the data
dataset = pd.read_excel("https://archive.ics.uci.edu/ml/
↪machine-learning-databases/00350/default%20of%20credit%20card%20clients.xls")
dataset.columns = dataset.iloc[0]
dataset.drop(['ID'], axis=1, index=None,  inplace=True)   # this statement␣
↪creates an error. Mostlikey it has been discarded from the source.
X=dataset.drop(["default payment next month"],  axis=1, index=None)
```

```
# y=dataset["default payment next month"]
```

[3]:
```
response = "default payment next month"  # response variable (y)
predictors = [cols for cols in dataset.columns if cols not in [response]] #␣
 ↪predictor variables (X)

X = dataset[predictors] # Defining predictor columns
y = dataset[response]   # Defining response columns

X2=X[1:] # discarding the 'heading' row (keeping the values)
y2=y[1:] # discarding the 'heading' row (keeping the values)

X1=X2.convert_dtypes()
y1=y2.convert_dtypes()

X=X1.astype(int)
y=y1.astype(int)
```

[4]:
```
# Here we'll build a classifier by using decision tree and calculate the␣
 ↪confusion matrix
# Train a DecisionTreeClassifier

from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier(max_depth=2, random_state = 43)
dtc.fit(X, y)
```

[4]:
```
DecisionTreeClassifier(max_depth=2, random_state=43)
```

[5]:
```
import time
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

# Train a DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier

# distribute Train and Test subsets of data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42)

start=time.time()
dtc = DecisionTreeClassifier(max_depth=2, random_state = 43,  criterion='gini')
dtc.fit(X_train, y_train)
y_pred=dtc.predict(X_test)
time_spent=time.time()-start
print('time using Gini =', time_spent )
```

```
time using Gini = 0.07978200912475586
```

[6]:
```
#Compute the exact cases of 'default payment next month'
y_test.value_counts()
```

[6]:
```
default payment next month
0    7040
1    1960
Name: count, dtype: int64
```

[7]:
```
# Calculating confusion_matrix(); max_depth=2, criteria=gini
confusion_matrix(y_test, y_pred)
```

[7]:
```
array([[6782,  258],
       [1362,  598]], dtype=int64)
```

[8]:
```
# Calculating confusion_matrix  - changing hyperparameters max_depth=3,␣
 ↪criteria=entropy
import time
start=time.time()
dtc = DecisionTreeClassifier(max_depth=3, random_state = 43, ␣
 ↪criterion='entropy')
dtc.fit(X_train, y_train)
y_pred=dtc.predict(X_test)
time_spent=time.time()-start
print('time using Entropy =', time_spent )

confusion_matrix(y_test, y_pred)
```

```
time using Entropy = 0.12573528289794922
```

[8]:
```
array([[6776,  264],
       [1351,  609]], dtype=int64)
```

Hyperparameter #1: CRITERION (values used gini and entropy: gini showed faster computation) Gini criterion (0.0797 second) is slightly faster than the entropy criterion (0.125 second). Both calculated at max_depth=3.

Hyperparameter #2: max_depth (values used 2 and 3: higher max_depth showed higher accuracy when the case is True - in this case 'default payment next month' = 0) The lower the depth of the DecisionTree the prediction of the true class (y=0) when the case is true is high. As we see that 6782 (related to max_depth=2, at criteria = entropy) is closer (higher accuracy) to 7040 comparing to 6776 (related to max_depth=3, at criteria = entropy).

[9]:
```
# Tune parameters with RandomizedSearchCV
```

[10]:
```
# import modules/classes
from sklearn.model_selection import RandomizedSearchCV
from sklearn.tree import DecisionTreeClassifier
from scipy.stats import randint
```

```
# setup parameters
param = {'max_depth' : [3, None],
         'criterion' : ['gini', 'entropy'],
         'min_samples_leaf' : [3, 4, 5]
         }

model = DecisionTreeClassifier()  # DecisionTreeClassifier Object
best = RandomizedSearchCV(dtc, param, cv=5) # RandomizedSearchCV object
best.fit(X_train, y_train)
```

[10]:
```
RandomizedSearchCV(cv=5,
                   estimator=DecisionTreeClassifier(criterion='entropy',
                                                    max_depth=3,
                                                    random_state=43),
                   param_distributions={'criterion': ['gini', 'entropy'],
                                        'max_depth': [3, None],
                                        'min_samples_leaf': [3, 4, 5]})
```

[11]:
```
print(best.best_params_)
```

```
{'min_samples_leaf': 3, 'max_depth': 3, 'criterion': 'gini'}
```

Following from previous results, we already achieved that max_depth=3 and criterion='gini' produced the best result. Here we see that the parameter best_params_ found the paramater values max_depth=3 and criterion='gini' to produce the best result.

[12]:
```
# Random Forest Ensemble
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
rfc= RandomForestClassifier(
    n_estimators=500,
    max_leaf_nodes=16,
    n_jobs=-1
)

rfc.fit(X_train, y_train)
y_Prfc = rfc.predict(X_test)
as_rfc = (accuracy_score(y_test, y_Prfc))
```

[13]:
```
print(as_rfc)
```

```
0.8177777777777778
```

[16]:
```
# AdaBoost Ensemble
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
abc = AdaBoostClassifier(
    DecisionTreeClassifier (max_depth=1),
    n_estimators=200,
    algorithm='SAMME.R',
    learning_rate=0.5
)

abc.fit(X_train, y_train)
y_Pabc = abc.predict(X_test)
as_abc = (accuracy_score(y_test, y_Pabc))
```

[17]:
```
print(as_abc)
```

```
0.8158888888888889
```

[18]:
```
# Extremely Randomized Trees (ExtraTrees) Ensemble
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.metrics import accuracy_score
etc = ExtraTreesClassifier (n_estimators=200)
etc.fit(X_train, y_train)
y_Petc = etc.predict(X_test)
as_etc = (accuracy_score(y_test, y_Petc))
```

[19]:
```
print(as_etc)
```

```
0.8096666666666666
```

[20]:
```
#  Gradient Boosting Ensemble
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
from sklearn.svm import LinearSVR


gbe = GradientBoostingRegressor(n_estimators=100)

gbe.fit(X_train, y_train)
y_Pgbe=mean_squared_error(gbe.predict(X_test), y_test)
as_gbe = (accuracy_score(y_test, y_Petc))
```

[21]:
```
print(as_gbe)
```

```
0.8096666666666666
```

[23]:
```
print("Accuracy Score: Random Forest ", as_rfc,
     "\nAccuracy Score: Ada Boost ", as_abc,
     "\nAccuracy Score: Extra Trees ", as_etc,
     "\nAccuracy Score: Gradient Boosting ", as_gbe)
```

```
Accuracy Score: Random Forest  0.8177777777777778
Accuracy Score: Ada Boost  0.8158888888888889
Accuracy Score: Extra Trees  0.8096666666666666
Accuracy Score: Gradient Boosting  0.8096666666666666
```

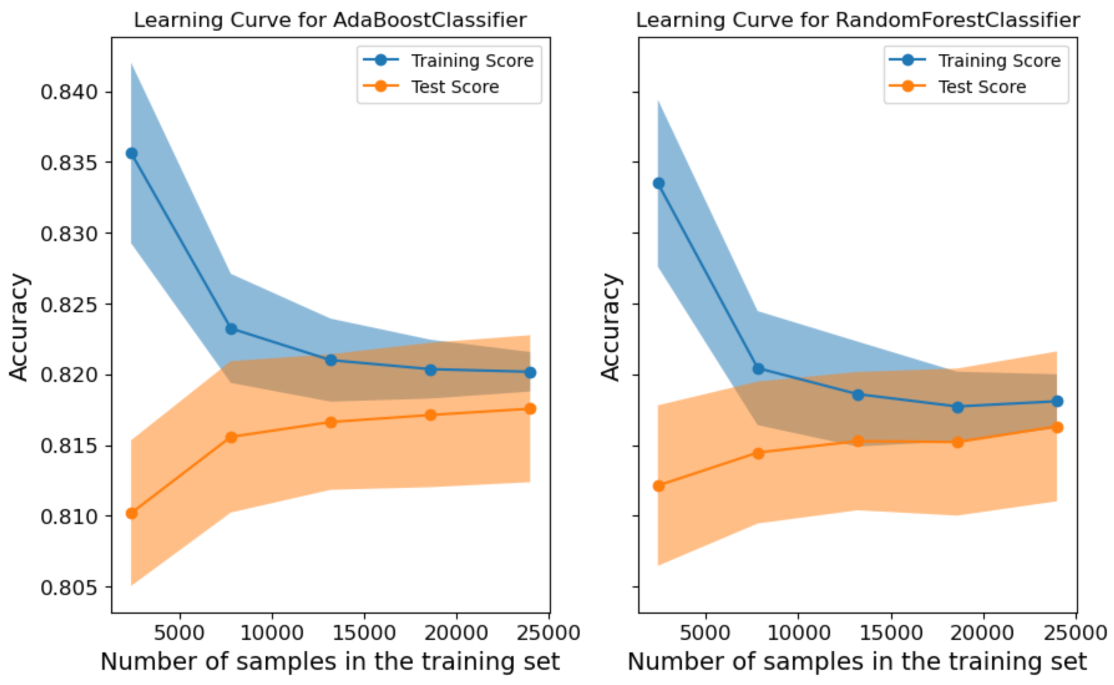
[24]:
```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.model_selection import LearningCurveDisplay, ShuffleSplit

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 6), sharey=True)

common_params = {
    "X": X,
    "y": y,
    "train_sizes": np.linspace(0.1, 1.0, 5),
    "cv": ShuffleSplit(n_splits=50, test_size=0.2, random_state=0),
    "score_type": "both",
    "n_jobs": 4,
    "line_kw": {"marker": "o"},
    "std_display_style": "fill_between",
    "score_name": "Accuracy",
}

for ax_idx, estimator in enumerate([abc, rfc]):
    LearningCurveDisplay.from_estimator(estimator, **common_params,␣
 ↪ax=ax[ax_idx])
    handles, label = ax[ax_idx].get_legend_handles_labels()
    ax[ax_idx].legend(handles[:2], ["Training Score", "Test Score"])
    ax[ax_idx].set_title(f"Learning Curve for {estimator.__class__.__name__}")
```



Analysis of the learning curves

From the display of accuracy score, we see that the Random Forest and the Ada Boost performed in the top.

In the above figures - both the ensemble classifiers the training score is very high when using few samples for training and decreases when increasing the number of samples, whereas the test score is very low at the beginning and then increases when adding samples. The training and test scores become more realistic when all the samples are used for training.

How does changing hyperparms effect model performance? Hyperparameters directly control model structure, function, and performance. Hyperparameter tuning allows data scientists to tweak model performance for optimal results. In question #2 we found that changing to a certain combination of hyperparameters yielded the optimal result which showed the same result when we tuned the hyperparameters using RandomizedSearchCV() object to achieve the optimal combination of parameters and best result.

Why certain models performed better/worse? All models have built-in assumptions. Data that we fit in the model are not in the perfect order. Then, when a model's assumptions match the characteristics of the data after making them in good order, we expect to get a good fit. So this model performs better. Other model that does not match the characteristic of the data does not perform better.

How does this performance line up with known strengths/weakness of these models? When we develop a Machine Learning model we actually get benefits of efficiency, automation and data processing capabilities. These are the strengths of a model. On the other hand if we have strong data dependency, interpretability issues, resource requirements, potential biases we'll have a weaker model. These can be stated as the weakness of a model. When we can build a model with reduced weakness and increased strength the model performs better.