Extreme Gradient Boosting

Kannan Singaravelu, CQF

June 2023

1 Tree Based Models

Tree based models such as decision trees are a class of machine learning models that are used for both classification and regression. The model is essentially a series of binary questions, where the trained models resemble a tree, with branches and nodes that contains all the combination of responses. Tree based models are explicability models and are easy to understand and explain.

2 Ensemble models

Ensemble models are machine learning models that use more than one predictor to arrive at a prediction. A group of predictors form an ensemble. In general, ensemble models perform better than using a single predictor (model) with improved efficiency and accuracy.

Ensemble learning can be classified as

- **Bagging** ensemble learning also referred as parallel ensemble
- **Boosting** ensemble learning also referred as sequential ensemble

Decision trees have been used predominantly to construct bagging and boosting based models in machine learning.

2.1 Bagging

A tree based model that aggregates the predictions of multiple trees is a Random Forest model. For it to be effective, the model needs a diverse collection of trees. Random Forest uses bootstrapping and aggregate results to perform operation and is also referred to as ***bagging ensemble model***.

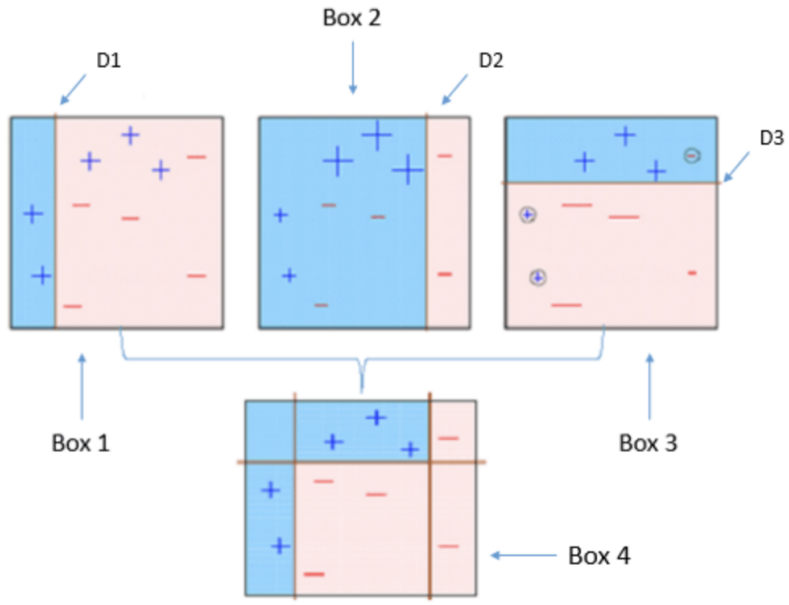
2.2 Boosting

Boosting is a process that uses a set of Machine Learning algorithms to combine weak learner to form strong learners in order to increase the accuracy of the model. The basic principle behind the working of boosting algorithms is to generate multiple weak learners and combine their predictions to form one strong rule.

Step 1: The base algorithm reads the data and assigns equal weight to each sample observation

Step 2: False predictions are assigned to the next base learner with a higher weightage on these incorrect predictions (increase the weightage on the misclassified samples)

Step 3: Repeat step2 until the algo can correctly classify the output



Representation of homogeneous classification of + and - classes

3 Gradient Boosting

Gradient boosting trees are another ensemble model where tree models are arranged in a sequence. The model is built sequentially as each additional tree aims to correct the previously built model's predictions. A model with M trees is equal to

$$f_M(x_j) = \sum_{m=1}^{M} \gamma_m h_m(x_j),$$

where h_m is a **weak learner** decision tree.

Additive Training : It is quite difficult to learn all the trees at once and we use an additive strategy which is fixing what it learned till last time step (tree) and add one new tree at a time. Then, we can write the prediction at step t as $\tilde{y_i}^{(t)}$. Then, we have

$f_0(x_j) = h_0(x_j)$
$f_1(x_j) = f_0(x_j) + \gamma_1 h_1(x_j)$
$f_2(x_j) = f_1(x_j) + \gamma_2 h_2(x_j)$
$\dots$
$f_M(x_j) = f_{M-1}(x_j) + \gamma_M h_M(x_j)$

where m is the number of trees, f is a function in the functional space F.

In random forest, diversification is achieved through many trees that will possibly improve the overall quality of the model. Gradient boosing algorithm combine multiple weak learners to form a strong learner.

The model is usually initialized with h_0 being equal to the mean of the training labels for regression or the majority class for classification. γ_m is a factor that scales the contribution of a tree to the overall model. The model is then trained iteratively for h_m by selecting γ_m that minimizes the loss function $L(y, f_m)$. This is repeated until the model includes all M trees.

The two most popular gradiant boosting algorithms are

- Adaboost: Improves the learning process by focusing on the instances that yield the largest errors.
- XGBboost: Flexible algorithm in which each new tree is only focused on the minimization of the training sample loss.

4 XGBoost

Tree boosting is highly effective and widely used machine learning method. XGBoost stands for "Extreme Gradient Boosting", where the term "Gradient Boosting" originates from the paper Greedy Function Approximation: A Gradient Boosting Machine, by Friedman.

XGBoost, which is used widely by data scientists to achieve state-of-the-art results on many machine learning challenges. XGBoost initially started as a research project by Tianqi Chen as part of the Distributed (Deep) Machine Learning Community (DMLC) group. Initially, it began as a terminal application which could be configured using a libsvm configuration file.

XGBoost was designed to be used with large, complicated datasets and is one of the most popular machine learning algorithm to deal with structured data. It is an advance version of gradient boosting method that is designed to focus on computational speed and model efficiency. XGBoost is preferred over other tree based model as it supports

- Parallelization
- Distributed computing methods
- Out-of-core computing
- Cache optimization

The algorithm is highly customizable and accurate in its predictions as it seeks to minimise the objective function

$$Obj^{(t)} = \underbrace{\sum_{i=1}^{n} l\left(y_i, \tilde{y}_i^{(t)}\right)}_{\text{training loss}} + \underbrace{\sum_{i=1}^{t} \Omega(f_i)}_{\text{regularisation term}}$$

$$Obj^{(t)} = \sum_{i=1}^{n} l\left(y_i, \tilde{y}_i^{(t-1)} + f_t(x_i)\right) + \sum_{i=1}^{t} \Omega(f_i)$$

The first term (over all instances) measures the distance between the true label and the output from the model while the second term (over all trees) penalises models that are too complex. Optimising loss tend to create more complex models while optimizing regularisation tends to generalise simplier models.

In XGBoost, we define the complexity as

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda {w_j}^2$$

where,

- T is the number of terminal nodes, or leaves in a tree
- γ and λ are regularisation terms which is a user definable penalty for pruning $>$ `gamma`: minimum loss reduction required to make a further partition on a leaf node of the tree. Large gamma will lead to more conservative algorithm. $>$ `lambda`: L2 regularization term on weights. Increasing this value will make model more conservative. Normalised to number of training examples.
- ${w_j}^2$ is the score of each leaf

While metrics for regression is straight forward, it can be complex for classification (logistic loss) where we use Taylor expansion of the loss function up to the second order. After removing all the constants, the specific objective (function) at step t becomes

$$\sum_{i=1}^{n}[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)] + \Omega(f_t)$$

where the g_i and h_i are defined as

$$g_i = \partial_{\tilde{y_i}^{(t-1)}} l(y_i, \tilde{y_i}^{(t-1)}) h_i = \partial^2_{\tilde{y_i}^{(t-1)}} l(y_i, \tilde{y_i}^{(t-1)})$$

Refer here for further details.

Manipulating the above formulation for classification, the leaf output is given as

$$\frac{\sum R_i}{\sum[P_i * (1 - P_i)] + \lambda}$$

where, R is the residual; P_i is the previous probability; λ is the regularization parameter.

5 Implementation

We will use XGBClassifier from XGBoost library to predict future prices of the security by creating a custom list of features from the raw price series.

Import Libraries

[ ]:
```
# Ignore warnings
import warnings
warnings.filterwarnings('ignore')
warnings.simplefilter(action='ignore', category=FutureWarning)

# Data manipulation
import pandas as pd
import numpy as np

# Plotting
import matplotlib.pyplot as plt
import seaborn as sns
# plt.rcParams['figure.figsize'] = (20,10)
# plt.style.use('fivethirtyeight')

# Preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import (
                                    train_test_split,
                                    RandomizedSearchCV,
                                    TimeSeriesSplit,
                                    cross_val_score
                                    )
# Classifier
from xgboost import XGBClassifier, plot_importance, to_graphviz

# metrics
from sklearn.metrics import (precision_recall_curve,
                             roc_curve,
                             RocCurveDisplay,
                             ConfusionMatrixDisplay
                            )

from sklearn.metrics import (accuracy_score,
                             f1_score,
                             recall_score,
                             precision_score,
                             roc_auc_score,
                             auc
                            )
```

```
from sklearn.metrics import (classification_report,
                             confusion_matrix
                            )
```

5.1 Retrieve Data

We will retrieve the adjusted closed price of SPY from locally stored data.

[ ]:
```
# Load file
# file_path = '/Users/roshan/Library/CloudStorage/GoogleDrive-<email>/My Drive/
 ↪2023/cqf/data/spy.csv'
df = pd.read_csv('data/spy.csv', index_col=0, parse_dates=True)[['Adj Close']]

# Calculate returns
df['Returns'] = np.log(df).diff()
df['Ret_1M'] = df['Returns'].rolling(20).sum()

# Output first five values
df.head()
```

5.2 EDA of Original dataset

[ ]:
```
# Descriptive statistics
df.describe()
```

5.3 Cleaning & Imputation

Data is already cleaned. However, NaN values from derived features should be treated.

[ ]:
```
# Check for missing values
df.isnull().sum()
```

5.4 Feature Specification

Features or Predictors are also known as an independent variable which are used to determine the value of the target variable. We will use raw price to derive list of custom features.

[ ]:
```
# Create features (predictors) list
features_list = []
for r in range(10, 65, 5):
    df['Ret_'+str(r)] = df.Returns.rolling(r).sum()
    df['Std_'+str(r)] = df.Returns.rolling(r).std()
    features_list.append('Ret_'+str(r))
    features_list.append('Std_'+str(r))

# Drop NaN values
df.dropna(inplace=True)
```

5.5 Target or Label Definition

Label or the target variable is also known as the dependent variable. Here, the target variable is whether SPY price will close up or down on the next trading day. If the tomorrow's closing price is greater than today's closing price, then we will buy the SPY, else we will sell SPY.

We assign a value of +1 for the buy signal and 0 otherwise to target variable. The target can be described as :

$$y_t = \begin{cases} 1, & \text{if } p_{t+1} > 0.995 * p_t \\ 0, & \text{if } p_{t+1} \text{Otherwise} \end{cases}$$

whre, p_t is the current closing price of SPY and p_{t+1} is the 1-day forward closing price of SPY.

[ ]:
```
# Define Target
df['Target'] = np.where(df['Adj Close'].shift(-1)>0.995 * df['Adj Close'],1,0)
# df = df[:-1]

# Check output
df
```

[ ]:
```
# Convert to NumPy
X = df.drop(['Adj Close', 'Returns', 'Ret_1M', 'Target'],axis=1)
X.values
```

[ ]:
```
# Define label or target
y = df['Target']
y
```

Split Data

[ ]:
```
# Splitting the datasets into training and testing data.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42, shuffle=False)

# Output the train and test data size
print(f"Train and Test Size {len(X_train)}, {len(X_test)}")
```

6 Base Model

We will now create and train the XGBoost (Regessor). All the feature and label should have a similar scale and the scaling can be accomplished by using the `StandardScaler` transformer.

6.1 Fit Model

[ ]:
```
# Scale and fit the classifier model
model =  XGBClassifier(verbosity = 0, silent=True, random_state=42)
model.fit(X_train, y_train)
```

6.2 Predict Model

[ ]:
```
# Predicting the test dataset
y_pred = model.predict(X_test)

# Predict Probabilities
y_proba = model.predict_proba(X_test)
```

[ ]:
```
acc_train = accuracy_score(y_train, model.predict(X_train))
acc_test = accuracy_score(y_test, y_pred)

print(f'Train Accuracy: {acc_train:0.4}, Test Accuracy: {acc_test:0.4}')
```

[ ]:
```
# Display confussion matrix
disp = ConfusionMatrixDisplay.from_estimator(
        model,
        X_test,
        y_test,
        display_labels=model.classes_,
        cmap=plt.cm.Blues
    )
disp.ax_.set_title('Confusion matrix')
plt.show()
```

[ ]:
```
# Classification Report
print(classification_report(y_test, y_pred))
```

[ ]:
```
# Display ROCCurve
disp_roc = RocCurveDisplay.from_estimator(
            model,
            X_test,
            y_test,
            name='XGBoost')
disp_roc.ax_.set_title('ROC Curve')
plt.plot([0,1], [0,1], linestyle='--')
plt.show()
```

7 Hyper-parameter Tuning

Hyper-parameters are parameters that are not directly learnt within estimators. They are passed as arguments to the constructor of the estimator classes (Classifier in this case). It is possible and recommended to search the hyper-parameter space for the best cross validation score. Any parameter provided when constructing an estimator may be optimized in this manner.

We will tune the hyperparameters to select the best score by TimeSeriesSplit cross-validation. This is a variation of KFold. In the kth split, it returns first k folds as train set and the (k+1)th fold as test set. Unlike standard cross-validation methods, successive training sets are supersets of those that come before them.

7.1 XGBoost's hyper-parameter

XGBoost has plethora of tuning parameters includeing the regulatization parameters and some of the most common hyperparameters are:

- `learning rate`: step size shrinkage used in update to prevents overfitting. Range is [0,1].
- `max_depth`: maximum depth of a tree.
- `colsample_bytree`: percentage of features used per tree. High value can lead to overfitting.
- `min_child_weight`: minimum sum of instance weight needed in a child.
- `gamma`: minimum loss reduction required to make a further partition on a leaf node of the tree. Large gamma will lead to more conservative algorithm.
- `lambda`: L2 regularization term on weights. Increasing this value will make model more conservative. Normalised to number of training examples. [parameter for linear booster].

Refer here for complete list of tuning parameters.

7.2 RandomizedSearch

The RandomizedSearchCV implements a "fit" and a "score" method and perform randomized search on hyper parameters. The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings. In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions.

[ ]:
```
# Timeseries CV 3-split
# tscv = TimeSeriesSplit(n_splits=4, gap=1) # sklearn 1.0
# for train, test in tscv.split(X):
#     print(train, test)
```

[ ]:
```
# Cross-validation
tscv = TimeSeriesSplit(n_splits=5, gap=1)
```

[ ]:
```
# Get params list
model.get_params()
```

[ ]:
```
# Hyper parameter optimization
param_grid = {'learning_rate': [0.05, 0.10, 0.15, 0.20, 0.25, 0.30],
              'max_depth': [3, 4, 5, 6, 8, 10, 12, 15],
              'min_child_weight': [1, 3, 5, 7],
              'gamma': [0.0, 0.1, 0.2 , 0.3, 0.4],
              'colsample_bytree': [0.3, 0.4, 0.5 , 0.7]}
```

[ ]:
```
# perform random search
rs = RandomizedSearchCV(model, param_grid, n_iter=100, scoring='f1', cv=tscv,␣
 ↪verbose=0)
rs.fit(X_train, y_train, verbose=0)
```

[ ]:
```
# best parameters
rs.best_params_
```

[ ]:
```
# best score
rs.best_score_
```

8 Tuned Model

Let's now train and predict the model with the best search parameter## Refit Model

[ ]:
```
# Refit the XGB Classifier with the best params
cls = XGBClassifier(**rs.best_params_)

cls.fit(X_train, y_train,
        eval_set=[(X_train, y_train), (X_test, y_test)],
        # eval_metric='logloss',
        verbose=True)
```

[ ]:
```
# Return the evaluation results
# evals_result = cls.evals_result()
# evals_result
```

[ ]:
```
# Cross validation score
score = cross_val_score(cls,X_train,y_train,cv=tscv)
print(f'Mean CV Score : {score.mean():0.4}')
```

8.1 Feature Importance

Feature Importance refers to techniques that calculate a score for all the input features for a given model where the scores represent the "importance" of each feature. It is calculated as the decrease in node impurity weighted by the probability of reaching that node. The node probability can be calculated by the number of samples that reach the node, divided by the total number of samples. The higher the value the more important the feature.

[ ]:
```
# Plot feature importance
fig, ax = plt.subplots(figsize=(10,8))
feature_imp = pd.DataFrame({'Importance Score': cls.
 ↪feature_importances_,'Features': X.columns}).sort_values(by='Importance␣
 ↪Score', ascending=False)

sns.barplot(x=feature_imp['Importance Score'], y=feature_imp['Features'])
ax.set_title('Features Importance');
```

Importance type can be either of the following:

- weight: the number of times a feature is used to split the data across all trees.
- gain: the average gain across all splits the feature is used in.
- cover: the average coverage across all splits the feature is used in.
- total_gain: the total gain across all splits the feature is used in.
- total_cover: the total coverage across all splits the feature is used in.

By default, `feature_importances_` rank features based on the average gain across all splits. This can be changed using the `plot_importance` method.

[ ]:
```
# The Gain is the most relevant attribute to interpret the relative importance␣
↪of each feature.
plot_importance?
```

[ ]:
```
# feature importance_type = 'gain'
plot_importance(cls, importance_type='weight', show_values=False);
```

8.2 Shap Values

SHAP (SHapley Additive exPlanations) is a game theoretic approach to explain the output of any machine learning model. Shapley values are a widely used approach from cooperative game theory that come with desirable properties and is the average marginal contribution of a feature value across all possible coalitions.

[ ]:
```
import shap
explainer = shap.TreeExplainer(cls)
shap_values = explainer.shap_values(X_test)
```

Visualization of feature importance

[ ]:
```
# future importance summary
shap.summary_plot(shap_values, X_test, plot_type="bar")
```

[ ]:
```
# interpretation plot
shap.summary_plot(shap_values, X_test)
```

8.3 Plot Tree

[ ]:
```
## Tree Visualization
# change tree number to see the corresponding plot
to_graphviz(cls, num_trees=10, rankdir='UT')
```

For a classification tree with two classes, the value of the leaf node represent the raw score. It can be converted to a probability score by using the logistic (sigmoid) function.

8.4 Predict Model

[ ]:
```
# Predicting the test dataset
y_pred = cls.predict(X_test)

# Measure Accuracy
acc_train = accuracy_score(y_train, cls.predict(X_train))
acc_test = accuracy_score(y_test, y_pred)
# Print Accuracy
print(f'\n Training Accuracy \t: {acc_train :0.4} \n Test Accuracy \t\t:␣
↪{acc_test :0.4}')
```

[ ]:
```
# Display confussion matrix
disp = ConfusionMatrixDisplay.from_estimator(
        cls,
        X_test,
        y_test,
        display_labels=model.classes_,
        cmap=plt.cm.Blues
    )
disp.ax_.set_title('Confusion matrix')
plt.show()
```

[ ]:
```
# Classification Report
print(classification_report(y_test, y_pred))
```

[ ]:
```
# Display ROCCurve
disp_roc = RocCurveDisplay.from_estimator(
            cls,
            X_test,
            y_test,
            name='Tuned XGBoost')
disp_roc.ax_.set_title('ROC Curve')
plt.plot([0,1], [0,1], linestyle='--')
plt.show()
```

9 References

- Cross-validation
- TimeSeriesSplit
- RandomizedSearchCV
- Hyperparameters Tuning
- Metrics & Scoring
- Introduction to Boosted Trees
- XGBoost documentation
- XGBoost (clone) documentation

June 2023, Certificate in Quantitative Finance.