QBUS6850: Tutorial 9 – Extreme Gradient Boosting

Objectives

1. Get familiar with Gradient Boosting algorithm for regression and classification
2. Learn how to use XGBoost package to do the classification and regression

1. Gradient Boosting Classification

In this task, let's review the example in Lecture 08. We will build a Gradient Boosting Decision Tree to do the classification on Iris dataset. This dataset contains 150 data point, and each data is described with 4 features (namely sepal length & width(cm) and petal length & width (cm))
Please refer to QBUS6850_Lecture08.pdf for the detailed explanation.

Step 0: Import libs and data

Read the code below, and try to answer the following questions:
1. What is the `T` in the following code? (in line 22)
2. What is the purpose of `get_dummies`? (in line 26)

```
import numpy as np
import pandas as pd
from random import shuffle
from sklearn import datasets
from sklearn.tree import DecisionTreeRegressor
np.random.seed(5)
# load the dataset
iris = datasets.load_iris()
X = iris.data
t = iris.target
# randomly shuffle the dataset
N = len(t)
index = list(range(N))
shuffle(index)
# select the first 12 data to train the model
m = 12
T = 3
X1 = X[index[:m]]
t1 = t[index[:m]]
t1 = pd.get_dummies(t1)
F = np.full((m,3),0.0)
rho = 1
```

Step 1: Initial Models and Initial Probability (Form F to P)

Classification is also a regression problem with a specially defined loss function, such as the cross entropy over the data.

Before you start reading the following code, please quickly review what is mathematical definition of cross entropy

```
# define a new function for cal the probability
def FtoP(F):
    expF = np.exp(F)
    a = expF.sum(axis=1)
    P = expF / a[:,None]
    return P
```

Step 2: Calculate the negative gradient

```
P = FtoP(F)
NegG = t1.values - P
print(NegG)
```

Step 3: Modeling from X to g(1) g(2) and g(3)

Build a decision tree `h1(x)` of depth 1 from `X` to `-g1`, `-g2`, and `-g3`.

For simplicity, we define `max_depth =2` for each tree.

```
baseH0 = []
baseH1 = []
baseH2 = []
# First Round
# for each t, build up a regressor
for t in range(T):
    regressor = DecisionTreeRegressor(random_state=0,
max_depth=2)
    h0 = regressor.fit(X1, NegG[:,0])
    baseH0.append(h0)

    regressor = DecisionTreeRegressor(random_state=0,
max_depth=2)
    h1 = regressor.fit(X1, NegG[:,1])
    baseH1.append(h1)

    regressor = DecisionTreeRegressor(random_state=0,
max_depth=2)
    h2 = regressor.fit(X1, NegG[:,2])
    baseH2.append(h2)
    #### not finished, need to update the F
```

Step 4: Updating F := F + ρh with ρ = 1 for simplicity

Please add the following statement in your for loop.

```
    F[:,0] = F[:,0] + rho*h0.predict(X1)
    F[:,1] = F[:,1] + rho*h1.predict(X1)
```

```
    F[:,2] = F[:,2] + rho*h2.predict(X1)
```

Step 5: F to P again

```
    #Next Round
    P = FtoP(F)
```

Step 6: Negative Gradient again

```
    NegG = t1.values - P
```

Step 7: Modelling Negative Gradients again, we have the second set of basic modeller h1(x), h2(x), and h3(x): Decision trees of depth 1

Note: the splitting points for this set of basic models are the same as the first set of basic models.

```
# Now the models are stored in BaseH0, BaseH1 and BaseH2
# Predict for a new case
x = X[index[148:150]]
F0 = 0.0
F1 = 0.0
F2 = 0.0

for t in range(T):
    F0 = F0 + baseH0[t].predict(x)
    F1 = F1 + baseH1[t].predict(x)
    F2 = F2 + baseH2[t].predict(x)
```

Step 8: Final models

```
F = np.vstack((F0,F1,F2))
F = F.T
```

Step 9: Prediction on new test data

Finally, you should able to see the probability outputs. We will select the largest probability as the class label index.

```
predictedP = FtoP(F)
print(predictedP)
```

Task 2: Installing the XGBoost package

Open the Command Termial:

If you are python3 user, then type in "`pip3 install xgboost`" to install the XGBoost.

If you are python2 user, then type in "`pip install xgboost`" to install the XGBoost.

For more information, please refer the official document at:
https://xgboost.readthedocs.io/en/latest/build.html#

Task 3: Predict Onset of Diabetes (Classification)

This tutorial is based on https://machinelearningmastery.com/develop-first-xgboost-model-python-scikit-learn/

In this task, we are going to use the Pima Indians onset of diabetes dataset. This dataset is comprised of 8 input variables that describe medical details of patients and one output variable to indicate whether the patient will have an onset of diabetes within 5 years.

You can learn more about this dataset on the UCI Machine Learning Repository website. This is a good dataset for a first XGBoost model because all of the input variables are numeric and the problem is a simple binary classification problem. It is not necessarily a good problem for the XGBoost algorithm because it is a relatively small dataset and an easy problem to model.

Download this dataset and place it into your current working directory with the file name "`pima-indians-diabetes.csv`"
https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv.

Step1: Load and Prepare Data

In this section we will load the data from file and prepare it for use for training and evaluating an XGBoost model.

We will start off by importing the classes and functions we intend to use in this tutorial.

```
from numpy import loadtxt
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Next, we can load the CSV file as a NumPy array using the NumPy function `loadtext()`.

```
# load data
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=",")
```

We must separate the columns (attributes or features) of the dataset into input patterns (X) and output patterns (Y). We can do this easily by specifying the column indices in the NumPy array format.

```
# split data into X and y
X = dataset[:,0:8]
Y = dataset[:,8]
```

Finally, we must split the X and Y data into a training and test dataset. The training set will be used to prepare the XGBoost model and the test set will be used to make new predictions, from which we can evaluate the performance of the model.
For this we will use `the train_test_split()` function from the scikit-learn library. We also specify a seed for the random number generator so that we always get the same split of data each time this example is executed.

```
# split data into train and test sets
seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, Y,
test_size=test_size, random_state=seed)
```

Step 2: Train the XGBoost Model

XGBoost provides a wrapper class to allow models to be treated like classifiers or regressors in the scikit-learn framework.

This means we can use the full scikit-learn library with XGBoost models.

The XGBoost model for classification is called `XGBClassifier`. We can create and and fit it to our training dataset. Models are fit using the scikit-learn API and the `model.fit()` function.

Parameters for training the model can be passed to the model in the constructor. Here, we use the sensible defaults.

```
# fit model no training data
model = XGBClassifier()
model.fit(X_train, y_train)
```

You can see the parameters used in a trained model by printing the model, for example:

```
print(model)
```

You can learn more about the defaults for the `XGBClassifier` and `XGBRegressor` classes in the XGBoost Python scikitlearn API.
https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.sklearn

You can learn more about the meaning of each parameter and how to configure them on the XGBoost parameters page.
https://xgboost.readthedocs.io/en/latest//parameter.html

We are now ready to use the trained model to make predictions.

Step 3: Make Predictions with XGBoost Model

We can make predictions using the fit model on the test dataset.

To make predictions we use the scikit-learn function `model.predict()`.

By default, the predictions made by XGBoost are probabilities. Because this is a binary classification problem, each prediction is the probability of the input pattern belonging to the first class. We can easily convert them to binary class values by rounding them to 0 or 1.

```
# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
```

Now that we have used the fit model to make predictions on new data, we can evaluate the performance of the predictions by comparing them to the expected values. For this we will use the built in `accuracy_score()` function in scikit-learn.

```
# evaluate predictions
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

Task 4: Boston Housing (regression)

In this task, we are going to use the Boston housing dataset.
This dataset contains information collected by the U.S Census Service concerning housing in the area of Boston Mass. It was obtained from the StatLib archive (http://lib.stat.cmu.edu/datasets/boston), and has been used extensively throughout the literature to benchmark algorithms. However, these comparisons were primarily done outside of Delve and are thus somewhat suspect.
The dataset is small in size with only 506 cases.

For more information of this dataset, please refer:
https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html

Step1: Load and Prepare Data

Similar to the previous task, we will load the data and prepare it for use for training and evaluating an XGBoost model.

```
import xgboost as xgb
import numpy as np
from sklearn.model_selection import KFold, train_test_split,
GridSearchCV
from sklearn.metrics import confusion_matrix,
mean_squared_error
from sklearn.datasets import load_boston
rng = np.random.RandomState(12345)
```

We could use `load_boston()` function to load the dataset.

```
print("Boston Housing: regression")
boston = load_boston()
```

Define the X and Y for your XGBoost model.

```
y = boston['target']
X = boston['data']
```

You could also print out the dataset information by calling the shape function.

```
print(X.shape)
```

Now you can see our dataset contains 506 datapoint, and each data point is described with 13 features.

Step 2: Applying k-fold Cross Validation

Cross-validation is a statistical method used to estimate the skill of machine learning models.

It is commonly used in applied machine learning to compare and select a model for a given predictive modelling problem in order to flag problems like overfitting and to give an insight on how the model will generalize to an independent dataset (i.e., an unknown dataset, for instance from a real problem)

In this section, we will use k-fold cross-validation. In k-fold cross-validation, the original sample is randomly partitioned into k equal sized subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining k − 1 subsamples are used as training data. The cross-validation process is then repeated k times, with each of the k subsamples used exactly once as the validation data. The k results can then be averaged to produce a single estimation. The advantage of this method over repeated random sub-sampling (see below) is that all observations are used for both training and validation, and each observation is used for validation exactly once. 10-fold cross-validation is commonly used, but in general k remains an unfixed parameter.

```
kf = KFold(n_splits=2, shuffle=True, random_state=rng)
```

For more information on K-Fold Cross Validation, please refer to:
https://en.wikipedia.org/wiki/Crossvalidation_(statistics)#k-fold_cross-validation

You may also refer to the following link for the detailed information of each parameter and how to configure them by using scikit libs:
http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

Step 3: Train and Make Prediction with XGBRegressor()

We then call `split()` function to generate indices to split data into training and test set. For each training and test set, we define a new `XGBRegressor()` model.

```
for train_index, test_index in kf.split(X):
    xgb_model = xgb.XGBRegressor().fit(X[train_index],
y[train_index])
    predictions = xgb_model.predict(X[test_index])
    actuals = y[test_index]
    print(mean_squared_error(actuals, predictions))
```

Step 4: Parameter Optimization

Now we are able to use `XGBRegressor()` function to make prediction.

One remaining issue is about the value selection of the hyperparameters, i.e. learning rate `alpha`, number of classifiers (`n_estimators`), and maximun depth of each tree model (`max_depth`).

By default, this function defines `learning_rate=0.1, n_estimators=100`, and `max_depth=3`. However, this setting may not suitable for our case. In order to find out what hyperparameter settings are suitable for our case, we use `GridSearchCV()` function to test different value of these hyperparameters. Note that, in this experiment, we only test different `max_depth` and `n_estimators`.

```
xgb_model = xgb.XGBRegressor()
clf = GridSearchCV(xgb_model, {'max_depth': [2,4,6],
'n_estimators': [50,100,200]}, verbose=1)

clf.fit(X,y)
print(clf.best_score_)
print(clf.best_params_)
```