Programming for Data Science and Artificial Intelligence

Classification - Gradient Boosting

Readings:

- [GERON] Ch7
- [VANDER] Ch5
- [HASTIE] Ch16
- https://scikit-learn.org/stable/modules/ensemble.html

In [1]:
```
Name = "Muhammad Omer Farooq Bhatti"
Id = "122498"
```

In [52]:
```
import numpy as np
from sklearn.datasets import load_boston, load_breast_cancer, load_digits    # <---- Import dataset
from sklearn.model_selection import train_test_split
from sklearn.dummy import DummyRegressor, DummyClassifier   # <---- Dummy Regressor to use as base model
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier   # <---- Decision Tree Regressor & C
```

Gradient Boosting

Another popular one is Gradient Boosting. Similar to AdaBoost, Gradient Boosting works by adding sequential predictors. However, instead of adding **weights**, this method tries to fit the new predictor to the **residual errors** made by the previous predictor. The hypothesis function of gradient boosting is as follows:

$$ H(x) = h_0(x) + \alpha_1h_1(x) + \cdots + \alpha_sh_s(x) $$
Although they look similar, notice that no alpha is applied to the first predictor. In addition, each alpha is the same, as opposed to voting power in AdaBoost. Typically, similar to AdaBoost, decision trees are used for each $h_i(x)$ but are not limited to stump. In practice, min_leaves are set to around 8 to 32.

Since gradient boosting actually originate from additive linear regression, we shall first talk about **gradient boosting for regression**. Also assume that we are using **regression trees** for our regressors.

Gradient Boosting for Regression

Firstly, let's look at the following equation where $h_0(x)$ is our first predictor and we would like to minimize the residual as follows:

$$h_0(x) + residual_0 = y $$$$ residual_0 = y - h_0(x) $$
That is, we would y to be as close as $h_0(x)$ such that residual is 0

$$ y = h_0(x) $$
The question is that is it possible to add the second predictor $h_1(x)$ such that the residual is further reduced

$$ y = h_0(x) + h_1(x) $$
This equation can be written as:

$$h_1(x) = y - h_0(x) $$
This equation informs us that if we can find a subsequent predictor that can best fit the "residual" (i.e. $y - h_0(x)$), then we can improve the accuracy.

How is this related to gradient descent?

Well, firstly, here is our loss function for regression:

$$J = \frac{1}{2}(y - h(x))^2$$
And here, we want to minimize J by gradient of the loss function in respect to by adjusting h_x. We can thus treat h_x as parameters and take derivatives:

$$\frac{\partial J}{\partial h_(x)} = h(x) - y $$
Thus, we can interpret residuals as negative gradients:

$$ \begin{aligned} y & = h_0(x) + h_1(x)\\ & = h_0(x) + (y - h_0(x)) \\ & = h_0(x) - (h_0(x) - y) \\ & = h_0(x) - \frac{\partial J}{\partial h_0(x)} \end{aligned} $$
So in fact, we are using "gradient" descent in "gradient" boosting to find the new model, written as:

$$h_1(x) = - \frac{\partial J}{\partial h_0(x)} = y - h_0(x)$$
or more generally

$$h_s(x) = - \frac{\partial J}{\partial h_{s-1}(x)} = y - h_{s-1}(x)$$
where s is the index of predictor

So residuals or gradients?

Although they are equivalent in the mse loss function, it is more useful to use negative gradients as it is more general, and can apply to other loss functions as well, e.g., **cross-entropy** in the case of classification.

In cross entropy, the loss function is

$$J= y log h(x) + (1 - y) lg (1-h(x))$$

If you look at our previous lecture on logistic regression, the derivative of this **in respect to h(x)** will be:

$$\frac{\partial J}{\partial h_(x)} = h(x) - y$$

This may look the same as mse, but note that our $h(x)$ (i.e., regression tree) outputs continuous values. In order to transform $h(x)$ into discrete class, we shall transform using sigmoid function g as follows:

$$g(h(x)) = g(z) = \frac{1}{1+e^{-z}}$$

For multiclass classification, g is defined as the softmax function:

$$g(h(x)) = g(z) = \frac{e^z_c}{\Sigma_{i=1}^{k} e^z_k}$$

Also remind that to use softmax function, we need to first one-hot encode our y. And during prediction, we need to perform `np.argmax` along the axis=1

Adding learning rate

To make sure adding the subsequent predictor would not overfit our model, we shall add an learning rate α in front of this, which shall be the same across all predictors (different from AdaBoost where alpha is different across all predictors)

$$h_s(x) = - \alpha \frac{\partial J}{\partial h_{s-1}(x)}$$

What about next predictor

We can stop if we are happy, either using some predefined iterations, or whether the residual does not decrease further using some validation set.

In this case, it is obvious that 2 predictors are simply not enough. Thus, we first need to calculate the residuals which are

$$ residual_1 = y - (h_0(x) + \alpha h_1(x))$$

then we define $h_2(x)$ as

$$h_2(x) = \alpha(y - (h_0(x) + \alpha h_1(x)))$$

And then repeat

The final prediction shall use the following hypothesis function $H(x)$:

$$ H(x) = h_0(x) + \alpha_1h_1(x) + \cdots + \alpha_sh_s(x) $$

Summary of steps

1. Initialize the model as simply mean or some constant
2. Predict and calculate the residual
3. Let the next model fit the residual
4. Predict using the combined models and calculate the residual
5. Let the next model fit this residual
6. Simply repeat 4-5 until stopping criteria is reached

When to use Boosting

Let's summarize some useful info about Gradient Boosting:

Advantages:

1. Extremely powerful - especially useful for heterogeneous data (e.g., house price, number of bedrooms).

Disadvantages:

1. They cannot be parallelized. Obvious since they are sequential predictors.
2. They can easily overfit, thus require careful choice of estimators or the use of regularization such as max_depth.
3. When we talk about homogeneous data such as images, videos, audio, text, or huge amount of data, deep learning works better.

===Task===

Modify the above scratch code such that:

- Notice that we are still using max_depth = 1. Attempt to tweak min_samples_split, max_depth for the regression and see whether we can achieve better mse on our boston data
- Notice that we only write scratch code for gradient boosting for regression, add some code so that it also works for binary classification. Load the breast cancer data from sklearn and see that it works.
- Further change the code so that it works for multiclass classification. Load the digits data from sklearn and see that it works
- Put everything into class

In [45]:
```
class gradientBoosting:
    def __init__(self, n_estimators=200, mode='regression',
                 learning_rate=0.1, max_depth=3, min_samples_split=5 ):

        #Parameters
        self.n_estimators = n_estimators              # <-- No. of predictors
        self.mode=mode                                # <-- 'regression'|'classification'
        self.learning_rate=learning_rate              # <-- uniform value of alpha defined for each predicto
        self.max_depth=max_depth                      # <-- max_depth of decision trees used as predictors
        self.min_samples_split=min_samples_split      # <-- min. samples required to split the node of a dec

        self.trained_models=[]                        # <-- List of predictors to be used


    def fit(self, X, y):
        models=[]
        tree_parameters = {'max_depth': self.max_depth, 'min_samples_split': self.min_samples_split}

        for i in range(0, self.n_estimators-1):    # <-- After including h0, we need (n_estimators-1) Decisi
            models.append(DecisionTreeRegressor(**tree_parameters)) # h1(x) and onwards -> DecisionTreeRegre

        #print("y :", y)

        #Defining first model h0 as dummy regressor
        #h1(x) and onwards are DecisionTreeRegressors
        h0 = DummyRegressor(strategy='mean')  #DummyRegressor will just predict the mean of training data
        h0.fit(X, y) # fit the dummy model
        self.trained_models.append(h0)  # First trained model h0 appended

        # For every model in list, first train using all previously trained models
        # then appending to the trained model list
        for model in models:

            y_pred = self.predict(X, final=False)   #First loop will just have h0 model, second will have h0
                                                    #third will have h0,h1,h2 for prediction and so on

            #print("y_pred: ", y_pred[:3])

            #residual will be the total errors made by trained_models
            #First loop will have y-h0, second will have y-(h0+a1*h1), third will have y-(h0+a1*h1+a2*h2) an
            #As models are added to the trained_models list after fitting on the residuals, we will see decr
            #value of residuals in the next iterations.
            residual = self.grad(y, y_pred)        #returns y - y_pred -> error OR -ive of Gradient

            model.fit(X, residual)              #Using fit method of DecisionTreeRegressor class to fit the
                                                #predict the residual value. This way we can predict the res
                                                #add it to our y_pred value to reduce error. Each subsequent
                                                #trained to predict a residual value which is decreasing wit
                                                #addition of a new predictor. This way the residual --> erro
                                                #reduced to a very small value.

                                                #Hyperparameters for the above operation may be alpha=learni
                                                #n_estimators = no. of predictors, and max_depth of the Deci

            self.trained_models.append(model)      #Add to trained models list to be used in next iteration
                                                   #train the next model
    def predict(self, X, final=True):

        #We first get a value using our base predictor which is a DummyRegressor
        #because it is not multiplied by alpha=learning_rate
        base_predictor = self.trained_models[0].predict(X)

        #Initialize the sum of residuals to be zero
        boost_predictors = 0

        #Predict residual error using DecisionTreeRegressors
        for model in self.trained_models[1:]:
            boost_predictors += self.learning_rate * model.predict(X)
        if self.mode == 'regression':
            #Return y_predicted = h0 + residuals
            return base_predictor + boost_predictors
        elif self.mode == 'classification' and final==False:
            #Return y_predicted = g( h0 + residuals ) where g(z) is softmax()
            return self.softmax(base_predictor + boost_predictors)
        elif self.mode == 'classification' and final==True:
            #Return y_predicted = argmax( g( h0 + residuals ) )  where g(z) is softmax()
            return self.softmax(base_predictor + boost_predictors).argmax(axis=1)
        else:
            raise ValueError("Please enter valid mode: 'regression' or 'classification'")

    def grad(self, y, y_pred):
        return y - y_pred     # -ive gradient w.r.t h(x) --> also referred to as residual (+ive gradient is
                              # We need gradient w.r.t h(x) as we are updating the h(x) function
                              # Previously we used gradient=X.T*(y_pred-y) which was w.r.t theta since we up
                              # the value of theta using gradient

    def softmax(self, z):  # <--- Softmax function works for both binary and multinomial classification
        return np.exp(z)/( np.sum( np.exp(z), axis=1, keepdims=True ) )   # <---- Softmax function
```

In [6]:
```
#Loading Data ----> Regression
X, y = load_boston(return_X_y = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=150)
```

In [7]:
```
#Gradient Boosting Regression
parameters = {'n_estimators': 200, 'mode': 'regression',
              'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 5} #Good results achieved for max_d
model = gradientBoosting(**parameters)
model.fit(X_train, y_train)
yhat = model.predict(X_test)

print("MSE: ", np.sum(((y_test-yhat)**2)/y_test.shape[0]))
```

```
MSE:  6.54283706815806
```

In [46]:
```
#Loading Data ----> Binary Classification
X, y = load_breast_cancer(return_X_y = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=150)
```

In [42]:
```
def encode_y(y):
    n_class = len(np.unique(y))
    y_encoded = np.zeros((y.shape[0], n_class))
    for _class in range(0, n_class):
        y_encoded[np.where(y==_class),_class] = 1 #for indexes where y=class --> y_encoded[selected_idx, cla
    return y_encoded
```

In [47]:
```
#Gradient Boosting Binary Classification
parameters = {'n_estimators': 200, 'mode': 'classification',
              'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 5}
y_train=encode_y(y_train)
print("y_encoded :", y_train[:5])
model = gradientBoosting(**parameters)
model.fit(X_train, y_train)
yhat = model.predict(X_test)
#print(yhat)
print("Accuracy : ", np.sum(yhat==y_test)/y_test.shape[0])
```

```
y_encoded : [[0. 1.]
 [0. 1.]
 [1. 0.]
 [1. 0.]
 [0. 1.]]
Accuracy :  0.9532163742690059
```

In [49]:
```
#Loading Data ----> Multinomial Classification
X, y = load_digits( return_X_y=True )
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=150)
```

In [50]:
```
#Gradient Boosting Multinomial Classification
parameters = {'n_estimators': 200, 'mode': 'classification',
              'learning_rate': 0.1, 'max_depth': 3, 'min_samples_split': 5}
y_train=encode_y(y_train)
print("y_encoded :", y_train[:5])
model = gradientBoosting(**parameters)
model.fit(X_train, y_train)
yhat = model.predict(X_test)
#print(yhat)
print("Accuracy : ", np.sum(yhat==y_test)/y_test.shape[0])
```

```
y_encoded : [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]
Accuracy :  0.9296296296296296
```

In [51]:
```
for max_depth in [1, 3, 5, 7, 9]:
    for alpha in [i/10 for i in range(1,10)]:
        parameters = {'n_estimators': 200, 'mode': 'classification',
              'learning_rate': alpha, 'max_depth': max_depth, 'min_samples_split': 5}
        model = gradientBoosting(**parameters)
        model.fit(X_train, y_train)
        yhat = model.predict(X_test)
        #print(yhat)
        print(f"For max_depth: {max_depth}, alpha: {alpha}, Accuracy : ", np.sum(yhat==y_test)/y_test.shape[

    #BEST RESULTS for max_depth = 5 and learning_rate = 0.5  ---> Accuracy :  0.9666666666666667
```

```
For max_depth: 1, alpha: 0.1, Accuracy :  0.7833333333333333
For max_depth: 1, alpha: 0.2, Accuracy :  0.8629629629629629
For max_depth: 1, alpha: 0.3, Accuracy :  0.8777777777777778
For max_depth: 1, alpha: 0.4, Accuracy :  0.8944444444444445
For max_depth: 1, alpha: 0.5, Accuracy :  0.9111111111111111
For max_depth: 1, alpha: 0.6, Accuracy :  0.9148148148148149
For max_depth: 1, alpha: 0.7, Accuracy :  0.912962962962963
For max_depth: 1, alpha: 0.8, Accuracy :  0.9148148148148149
For max_depth: 1, alpha: 0.9, Accuracy :  0.9166666666666666
For max_depth: 3, alpha: 0.1, Accuracy :  0.9296296296296296
For max_depth: 3, alpha: 0.2, Accuracy :  0.9518518518518518
For max_depth: 3, alpha: 0.3, Accuracy :  0.9555555555555556
For max_depth: 3, alpha: 0.4, Accuracy :  0.9555555555555556
For max_depth: 3, alpha: 0.5, Accuracy :  0.9592592592592593
For max_depth: 3, alpha: 0.6, Accuracy :  0.9592592592592593
For max_depth: 3, alpha: 0.7, Accuracy :  0.9537037037037037
For max_depth: 3, alpha: 0.8, Accuracy :  0.9592592592592593
For max_depth: 3, alpha: 0.9, Accuracy :  0.9629629629629629
For max_depth: 5, alpha: 0.1, Accuracy :  0.9518518518518518
For max_depth: 5, alpha: 0.2, Accuracy :  0.9592592592592593
For max_depth: 5, alpha: 0.3, Accuracy :  0.9629629629629629
For max_depth: 5, alpha: 0.4, Accuracy :  0.9648148148148148
For max_depth: 5, alpha: 0.5, Accuracy :  0.9666666666666667
For max_depth: 5, alpha: 0.6, Accuracy :  0.9629629629629629
For max_depth: 5, alpha: 0.7, Accuracy :  0.9666666666666667
For max_depth: 5, alpha: 0.8, Accuracy :  0.9648148148148148
For max_depth: 5, alpha: 0.9, Accuracy :  0.9648148148148148
For max_depth: 7, alpha: 0.1, Accuracy :  0.9462962962962963
For max_depth: 7, alpha: 0.2, Accuracy :  0.9481481481481482
For max_depth: 7, alpha: 0.3, Accuracy :  0.9481481481481482
For max_depth: 7, alpha: 0.4, Accuracy :  0.95
For max_depth: 7, alpha: 0.5, Accuracy :  0.9555555555555556
For max_depth: 7, alpha: 0.6, Accuracy :  0.9518518518518518
For max_depth: 7, alpha: 0.7, Accuracy :  0.9462962962962963
For max_depth: 7, alpha: 0.8, Accuracy :  0.9481481481481482
For max_depth: 7, alpha: 0.9, Accuracy :  0.95
For max_depth: 9, alpha: 0.1, Accuracy :  0.9166666666666666
For max_depth: 9, alpha: 0.2, Accuracy :  0.9185185185185185
For max_depth: 9, alpha: 0.3, Accuracy :  0.9111111111111111
For max_depth: 9, alpha: 0.4, Accuracy :  0.8962962962962963
For max_depth: 9, alpha: 0.5, Accuracy :  0.9037037037037037
For max_depth: 9, alpha: 0.6, Accuracy :  0.9074074074074074
For max_depth: 9, alpha: 0.7, Accuracy :  0.9074074074074074
For max_depth: 9, alpha: 0.8, Accuracy :  0.9185185185185185
For max_depth: 9, alpha: 0.9, Accuracy :  0.9074074074074074
```