09.6 - Supervised Learning - Classification - Boosting, AdaBoost, GradientBoosting Scratch

October 12, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 9.6 - Classification - Boosting, AdaBoost, GradientBoosting Scratch

1.1.1 Readings:

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

[1]:
```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
import numpy as np
import matplotlib.pyplot as plt

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

1.2 Boosting

Boosting is a general strategy for learning classifiers by combining simpler ones. The idea of boosting is to take a "weak classifier" — that is, any classifier that will do at least slightly better than chance — and use it to build a much better classifier, thereby boosting the performance of the weak classification algorithm. This boosting is done by averaging the outputs of a collection of weak classifiers. The common form of hypothesis function for boosting is as follows:

$$\begin{aligned} H(x) &= \alpha_1 h_1(x) + \alpha_2 h_2(x) + \cdots + \alpha_s h_s(x)) \\ &= \Sigma_{s=1}^{S} \alpha_s h_s(x) \end{aligned}$$

where S = number of classifiers and α is the weight associated with each classifier

The among the first, and therefore popular boosting algorithm is **AdaBoost**, so-called because it is *adaptive.*

AdaBoost is extremely simple to use and implement (far simpler than SVMs), and often gives very effective results. There is tremendous flexibility in the choice of weak classifier as well. Anyhow, Decision Tree with max_depth=1 and max_leaf_nodes=2 are often used (also known as **stump**)

Suppose we are given training data $(\mathbf{x_i}, y_i)$, where $\mathbf{x_i} \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$. And suppose we are given a (potentially large) number (denoted S) of weak classifiers, denoted $h_s(x) \in \{-1, 1\}$ where $s = 1, 2, \cdots, S$, and for each classifier, we define α_s as the *voting power* of the classifier $h_s(x)$. Then, the hypothesis function is based on a linear combination of the weak classifier and is written as:

$$\begin{aligned} H(x) &= \text{sign}\big(\alpha_1 h_1(x) + \alpha_2 h_2(x) + \cdots + \alpha_s h_s(x))\big) \\ &= \text{sign}\big(\Sigma_{s=1}^{S} \alpha_s h_s(x)\big) \end{aligned}$$

Our job is to find the optimal α_s, so we can know which classifier we should give more weightage (i.e., believe more) in our hypothesis function since their accuracy is relatively better compared to other classifiers. To get this alpha, we should define what is "good" classifier. This is simple, since good classifier should simply has the maximum number of accurate classified samples as:

$$max\big(\Sigma_{i=1}^{m} I(h_s(x_i) = y_i)\big)$$

or can be written as minimization function as

$$min\big(\Sigma_{i=1}^{m} I(h_s(x_i) \neq y_i)\big)$$

Aside from "weighted" wisdom of crowd, AdaBoost has one more capability, and that is that each subsequent classifier will try to correct the errors made by previous predictor. In other words, whatever samples the previous classifier misclassified, it should be prioritized in the subsequent classifier. To realize this mechanism, the concept is to increase the penalty if those previously misclassified sample are wrong. To do so, we first initialize the weight for each sample, which shall be applied to the first predictor $h_1(x)$ to be

$$w_i^{(s)} = \frac{1}{m}; s = 1; i = 1, 2, \cdots, m$$

Then, after the first classifier was fitted, we readjust this weight by increasing weight for those misclassified sample, and decreasing weight for those correctly classified sample. To make sure classifier will be chosen on the basis of these weighted errors, we shall revise our definition of "good" classifiers as follows:

$$min\big(\frac{\Sigma_{i=1}^{m} w_i^s I(h_s(x_i) \neq y_i)}{\Sigma_{i=1}^{m} w_i^s}\big)$$

Note that the lower term is simply so that all weights sum to 1.

Thus, the subsequent classifier will be chosen based on the one that can create the least weighted errors.

Let's put everything into the AdaBoost algorithm as follows:

define S

for i from 1 to m {

$$w_i^{(1)} = \frac{1}{m}$$

make sure $y \in \{-1, 1\}$

}

for $s = 1$ to S {

Looping through all features and threshold, identify the best stump, whose value has the minimum of this objective function:

$$\epsilon_s = \frac{\Sigma_{i=1}^m w_i^s I(h_s(x_i) \neq y_i)}{\Sigma_{i=1}^m w_i^s}$$

where I is indicator function $I(h_s(x_i) \neq y_i) = 1$ if $h_s(x_i) \neq y_i$ and 0 otherwise

Then calculate the voting power of the weak classifier, denoted α_s and can be calculated as:

$$\alpha_s = \frac{1}{2} ln \frac{1 - \epsilon_s}{\epsilon_s}$$

Before fitting the next stump, we need to make sure to exaggerate the weights of *incorrectly* classified samples so our next stump will be chosen based on the new weighted objective function:

Then **for** all i {

$$w_i^{(s+1)} = w_i^{(s)} e^{\alpha_s I(h_s(\mathbf{x_i} \neq y_i)}$$

}

}

To predict, we simply take the weighted sum of all predictors and take the sign of them. Recall that S is number of stumps/predictors you have

$$H(x) = \text{sign}\big(\Sigma_{s=1}^S \alpha_s h_s(x)\big)$$

Stopping criteria of AdaBoost is important to impose or else we can get some overfitting with AdaBoost by adding too many classifiers. We can either specify the number of iterations, or when we reach a certain level of accuracy, or perform early stopping by using a validation set to detect the iteration when overfit starts to happen.

1.3 AdaBoost

1.3.1 Scratch

[2]:
```
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=500, random_state=1)
y = np.where(y==0,-1,1)  #change our y to be -1 if it is 0, otherwise 1

X_train, X_test, y_train, y_test = train_test_split(
```

```
    X, y, test_size=0.3, random_state=42)
```

[3]:
```
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

m = X_train.shape[0]
S = 20
stump_params = {'max_depth': 1, 'max_leaf_nodes': 2}
models = [DecisionTreeClassifier(**stump_params) for _ in range(S)]

#initially, we set our weight to 1/m
W = np.full(m, 1/m)

#keep collection of a_j
a_js = np.zeros(S)

for j, model in enumerate(models):

    #train weak learner
    model.fit(X_train, y_train, sample_weight = W)

    #compute the errors
    yhat = model.predict(X_train)
    err = W[(yhat != y_train)].sum()

    #compute the predictor weight a_j
    #if predictor is doing well, a_j will be big
    a_j = np.log ((1 - err) / err) / 2
    a_js[j] = a_j

    #update sample weight; divide sum of W to normalize
    W = (W * np.exp(-a_j * y_train * yhat))
    W = W / sum (W)


#make weighted predictions
Hx = 0
for i, model in enumerate(models):
    yhat = model.predict(X_test)
    Hx += a_js[i] * yhat

yhat = np.sign(Hx)

print(classification_report(y_test, yhat))
```

```
              precision    recall  f1-score   support
```

```
          -1       0.96      0.97      0.97        79
           1       0.97      0.96      0.96        71

    accuracy                           0.97       150
   macro avg       0.97      0.97      0.97       150
weighted avg       0.97      0.97      0.97       150
```

1.3.2 Sklearn

Sklearn implements AdaBoost using SAMME which stands for Stagewise Additive Modeling using a Multiclass Exponential Loss Function.

The following code trains an AdaBoost classifier based on 200 Decision stumps. A Decision stump is basically a Decision Tree with max_depth=1. This is the default base estimator of AdaBoostClassifier class:

[4]:
```
from sklearn.ensemble import AdaBoostClassifier

#SAMME.R - a variant of SAMME which relies on class probabilities
#rather than predictions and generally performs better
ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)
y_pred = ada_clf.predict(X_test)
print("Ada score: ", accuracy_score(y_test, y_pred))
```

```
Ada score:  0.9666666666666667
```

1.3.3 ===Task===

Your work: Let's modify the above scratch code:

Notice that if err = 0, then α will be undefined, thus attempt to fix this by adding some very small value to the lower term

Notice that sklearn version of AdaBoost has a parameter learning_rate. This is in fact the $\frac{1}{2}$ in front of the α calculation. Attempt to change this $\frac{1}{2}$ into a parameter called eta, and try different values of it and see whether accuracy is improved. Note that sklearn default this value to 1.

Observe that we are actually using sklearn DecisionTreeClassifier. If we take a look at it closely, it is actually using weighted gini index, instead of weighted errors that we learn above. Attempt to write your own class of class Stump that actually uses weighted errors, instead of weighted gini index

```
<li>Put everything into a class</li>
</ol>
```

1.4 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:

$$H(x) = h_0(x) + \alpha_1 h_1(x) + \cdots + \alpha_s h_s(x)$$

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.

1.4.1 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:

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

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

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

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

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

This may look the same as mse, but here, h(x) is

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

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 $H(x)$:

$$H(x) = h_0(x) + \alpha_1 h_1(x) + \cdots + \alpha_s h_s(x)$$

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

1.4.2 Gradient Boosting for Classification

What we have discussed is gradient boosting for regression. However, there are not much difference for classification.

Recall the cost function of classification is **cross entropy** where its derivative is simply:

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

Although this may look similar, $h(x)$ carries a function that convert real value to probabilities.

For binary classification, $h(x)$ is defined as the sigmoid function:

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

For multiclass/binary classification, $h(x)$ is defined as the softmax function:

$$\frac{e_c^x}{\Sigma_{i=1}^k e_k^x}$$

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

1.4.3 Scratch

[5]:
```
from scipy.special import expit
from sklearn.tree import DecisionTreeRegressor
from sklearn.dummy import DummyRegressor

def grad(y, f):
    return y - f

def fit(X, y, models):

    models_trained = []

    #using DummyRegressor is a good technique for starting model
    first_model = DummyRegressor(strategy='mean')
    first_model.fit(X, y)
    models_trained.append(first_model)

    #fit the estimators
    for i, model in enumerate(models):
        #predict using all the weak learners we trained up to
        #this point
        y_pred = predict(X, models_trained)

        #errors will be the total errors maded by models_trained
        residual = grad(y, y_pred)

        #fit the next model with residual
        model.fit(X, residual)

        models_trained.append(model)

    return models_trained

def predict(X, models):
```

```
    learning_rate = 0.1  ##hard code for now
    f0 = models[0].predict(X)  #first use the dummy model
    boosting = sum(learning_rate * model.predict(X) for model in models[1:])
    return f0 + boosting
```

[6]:
```
# Regression

from sklearn.datasets import load_boston
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import GradientBoostingRegressor

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=42)

n_estimators = 200
tree_params = {'max_depth': 1}
models = [DecisionTreeRegressor(**tree_params) for _ in range(n_estimators)]

#fit the models
models = fit(X_train, y_train, models)

#predict
y_pred = predict(X_test, models)

#print metrics
print("Our MSE: ", mean_squared_error(y_test, y_pred))
```

```
Our MSE:  12.945557601580582
```

1.4.4 Sklearn

sklearn has implemented GradientBoosting under the API of GradientBoostingClassifier for classification and GradientBoostingRegressor for regression.

[7]:
```
#Compare to sklearn: ls is the same as our mse
sklearn_model = GradientBoostingRegressor(
    n_estimators=n_estimators,
    learning_rate = 0.1,
    max_depth=1,
    loss='ls'
)

y_pred_sk = sklearn_model.fit(X_train, y_train).predict(X_test)

#print metrics
```

```
print("Sklearn MSE: ", mean_squared_error(y_test, y_pred_sk))
```

```
Sklearn MSE:  12.945557601580584
```

XGBoost XGBoost is an optimized distributed gradient boosting, designed to be more efficient, flexible, and portable (Chen and Guestrin 2016). In fact, XGBoost is often an important component of the winning entries in ML competitions (e.g., Kaggle). XGBoost also offers several nice features, such as automatically taking care of early stopping: XGBoost's API is quite similar to Scikit-Learn's:

[8]:
```
#make sure to pip install xgboost
#for mac guys, do "brew install libomp" which installs openMP library
#required for XGBoost

import xgboost

xgb_reg = xgboost.XGBRegressor()

#not improved after 2 iterations
xgb_reg.fit(X_train, y_train,
                eval_set=[(X_test, y_test)], early_stopping_rounds=2)
y_pred = xgb_reg.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("MSE:", mse)  #notice we are using mse while xgb uses root mse
```

```
[0]     validation_0-rmse:16.15458
Will train until validation_0-rmse hasn't improved in 2 rounds.
[1]     validation_0-rmse:11.84377
[2]     validation_0-rmse:8.79602
[3]     validation_0-rmse:6.72584
[4]     validation_0-rmse:5.46526
[5]     validation_0-rmse:4.65454
[6]     validation_0-rmse:4.08462
[7]     validation_0-rmse:3.76129
[8]     validation_0-rmse:3.54313
[9]     validation_0-rmse:3.37742
[10]    validation_0-rmse:3.24836
[11]    validation_0-rmse:3.18872
[12]    validation_0-rmse:3.10860
[13]    validation_0-rmse:3.09993
[14]    validation_0-rmse:3.08393
[15]    validation_0-rmse:3.08760
[16]    validation_0-rmse:3.06310
[17]    validation_0-rmse:3.05292
[18]    validation_0-rmse:3.05715
[19]    validation_0-rmse:3.05827
Stopping. Best iteration:
[17]    validation_0-rmse:3.05292
```

```
MSE: 9.320308418219375
```

Let's look at time

[9]:
```
%timeit xgboost.XGBRegressor().fit(X_train, y_train)
%timeit GradientBoostingRegressor().fit(X_train, y_train)
```

```
30.1 ms ± 886 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
79.6 ms ± 329 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

1.4.5 ===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

1.4.6 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.

[ ]:
```

```