Ensemble Methods: Gradient Boosting

Another very popular Boosting algorithm is **Gradient Boosting**. Just like AdaBoost, Gradient Boosting works by **sequentially adding predictors** to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors (or more general gradient) made by the previous predictor.

A Simple Example: boosting from the perspective of fitting to residual error

You are given a training data $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ and the task is to fit a model $g_1 = g_0 + h_1$ from g_0 to minimize the residual errors. The simple solution is:

$$
\begin{aligned}
&g_0(\mathbf{x}_1) + h_1(\mathbf{x}_1) = y_1 \\
&g_0(\mathbf{x}_2) + h_1(\mathbf{x}_2) = y_2 \\
&\cdots \\
&g_0(\mathbf{x}_n) + h_1(\mathbf{x}_n) = y_n
\end{aligned}
$$

Or, equivalently, you wish

$$
\begin{aligned}
&h_1(\mathbf{x}_1) = y_1 - g_0(\mathbf{x}_1) \\
&h_1(\mathbf{x}_2) = y_2 - g_0(\mathbf{x}_2) \\
&\cdots \\
&h_1(\mathbf{x}_n) = y_n - g_0(\mathbf{x}_n)
\end{aligned}
$$

We may not find a regression tree to fit these residuals perfectly but we can certainly achieve the goal approximately. **How?**

A: We can fit a regression tree h_1 to the data $\mathcal{D} = \{(\mathbf{x}_i, y_i - g_0(\mathbf{x}_i))\}_{i=1}^n$.

$g_1 = g_0 + h_1$

Fun Time: In general, will g_1 perform better than g_0? (1) Yes (2) No

Remark: The $y_i - g_0(\mathbf{x}_i)$ are called residuals. These are the parts that existing model g_0 cannot do well. The role of h_1 is to compensate the shortcoming of existing model g_0. If the new model $g_1 = g_0 + h_1$ is still not satisfactory, we can add another regression tree.

Formally, the idea is to start with a simple model (weak learner) g_0 for the training data $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ and then to improve or "**boost**" this learner to a learner $g_1 := g_0 + h_1$. Here, the function (or hypothesis) h_1 is found by minimizing the training loss for $g_0 + h_1$ over all functions (or hypotheses) h in some class of functions (or hypothesis set) $\mathcal{H}$. For example, $\mathcal{H}$ could be the set of functions that can be obtained via a decision tree of maximal depth 2. Given a loss function $L(\cdot)$, the function h_1 is thus obtained as the solution to the optimization problem:

$$h_1 = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(y_i, g_0(\mathbf{x}_i) + h(\mathbf{x}_i))$$

This process can be repeated for g_1 to obtain $g_2 = g_1 + h_2$, and so on, yielding the boosted prediction function

$$g_B = g_0 + \sum_{b=1}^{B} h_b$$

Or

$$g_B = g_0 + \gamma \sum_{b=1}^{B} h_b$$

$0 < \gamma < 1$ to **reduce overfitting**.

A Simple Example (Take Two): boosting from the perspective of gradient decent

Let us again consider a very simple regression setting using the half of the square-error loss $L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ in which y is the ground truth and $\hat{y}$ is the predicted value. We can now explicitly write down the loss function for h_1:

$$L = \sum_{i=1}^{n} \frac{1}{2}(y_i - (g_0(\mathbf{x}_i) + h(\mathbf{x}_i))^2 = \sum_{i=1}^{n} \frac{1}{2}(h(\mathbf{x}_i) - (y_i - (g_0(\mathbf{x}_i)))^2$$

By minimizing the loss, we look to tune the fitting parameters of h so that:

$$h_1(\mathbf{x}_i) \approx (y_i - g_0(\mathbf{x}_i))$$

Remark: As we have observed, the function h_b is just simply to approximate our original output y_i minus the contribution of the previous model g_{b-1}. This quantity is called residual error or residual:

$$e_i^b := y_i - g_{b-1}(\mathbf{x}_i).$$

And the dataset for h_b to fit is $\mathcal{D}_b = \{(\mathbf{x}_i, e_i^b)\}_{i=1}^n$.

Now back to the link with gradient: we note that the negative gradient of the half of square error loss at $(b-1)$ boosting is:

$$-\frac{\partial L}{\partial z}\Big|_{z=g_{b-1}(\mathbf{x}_i)} = -\frac{\partial}{\partial z}\left(\frac{1}{2}(y_i - z)^2\right)\Big|_{z=g_{b-1}(\mathbf{x}_i)} = (y_i - g_{b-1}(\mathbf{x}_i))$$

This is the residual e_i^b. We thus conclude for regression with the half of the square loss:

residual ⇔ negative gradient
fit h_b to residual ⇔ fit h_b to negative gradient
update g_{b-1} based on residual ⇔ update g_{b-1} based on negative gradient

So we are actually updating our model using gradient descent! In fact, one of the major advances in gradient boosting was the recognition that one can use a similar gradient descent method (more on this in the later Chapter) for any differentiable loss function. The resulting algorithm is called **gradient boosting**.

Remark: The benefit of formulating this algorithm using gradients is that it allows us to consider other loss functions and derive the corresponding algorithms in the same way.

Python Example: we now compare performance of decision tree, random forest and gradient boosting for a regression problem. We use the R^2 metric (coefficient of determination) for comparison. In regression, the R^2 is a statistical measure of how well the regression predictions approximate the real data points. An R^2 of 1 indicates that the regression predictions perfectly fit the data.

Let us first generate a few data with related by polynomial and sine transforms with `make_friedman1` from `sklearn`.

```
# create regression problem
n_points = 1000 # points
x, y =  make_friedman1(n_samples=n_points, n_features=15,
                       noise=1.0, random_state=100)

# split to train/test set
x_train, x_test, y_train, y_test = \
        train_test_split(x, y, test_size=0.33, random_state=100)
```

We will first use a deep decision tree (with nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples):

```
# decision tree
from sklearn.tree import DecisionTreeRegressor

# training
regTree = DecisionTreeRegressor(random_state=100)
regTree.fit(x_train,y_train)

# test
yhatdt = regTree.predict(x_test)
print("Decision Tree R^2 score = ", r2_score(y_test, yhatdt))
```

```
Decision Tree R^2 score =  0.5777939315921408
```

We continue with the random forest with B = 500 trees and a subset feature size $m = 8$:

```
# random forest
from sklearn.ensemble import RandomForestRegressor

# training
rf = RandomForestRegressor(n_estimators=500, max_features=8,
random_state=100)
rf.fit(x_train,y_train)

# test
yhatrf = rf.predict(x_test)
print("Random Forest R^2 score = ", r2_score(y_test, yhatrf))
```

```
Random Forest R^2 score =  0.8106675872067525
```

Remark (The Optimal Number of Subset Features m): Recall in random forests, m features is chosen as split candidates from the full set of p features. The default values for m are $p/3$ and $\sqrt{p}$ for regression and classification setting, respectively. However, the standard practice is to treat m as a hyperparameter that requires tuning, depending on the specific problem at hand.

Finally, let us use the gradient boosting estimator implemented in `sklearn`. We use $\gamma = 0.1$ and perform B = 100 boosting rounds. As a prediction function h_b for b = 1, . . . , B we use small decision

trees of depth at most 3. Note that such individual trees do not usually give good performance; that is, they are weak prediction functions.

```
# boosting sklearn
from sklearn.ensemble import GradientBoostingRegressor

# training
breg = GradientBoostingRegressor(learning_rate=0.1,
            n_estimators=100, max_depth =3, random_state=100)
breg.fit(x_train,y_train)

# test
yhatb = breg.predict(x_test)
print("Gradient Boosting R^2 score = ",r2_score(y_test, yhatb))
```

```
Gradient Boosting R^2 score =  0.8992706169055638
```

We can see that the resulting boosting prediction function gives the R^2 score equal to 0.899, which is better than R^2 scores of simple decision tree (0.5754), and the random forest (0.8106).

You can download the above python code `Ensemble_compare_01.ipynb` from the course website.

Remarks:

1. Gradient boosting is also known as gradient tree boosting, stochastic gradient boosting (an extension), and gradient boosting machines, or GBM for short.
2. Ensembles for gradient boosting are often constructed from shallow decision tree models. Trees are added one at a time to the ensemble and fit to correct the prediction errors made by prior models.
3. Models are fit using any arbitrary differentiable loss function and gradient descent optimization algorithm. This gives the technique its name, "gradient boosting," as the loss gradient is minimized as the model is fit, much like a neural network.
4. Gradient boosting is an effective machine learning algorithm and is often the main, or one of the main, algorithms used to win machine learning competitions (like Kaggle) on tabular and similar structured datasets.
5. A popular implementation of gradient boosting is the version provided with the scikit-learn library. Additional third-party libraries are available that provide computationally efficient alternate implementations of the algorithm that **often achieve better results in practice**. Examples include the **XGBoost** (eXtreme Gradient Boosting) library, the **LightGBM** (Light Gradient Boosting Machine) library, and the **CatBoost** (Categorical Features+Gradient Boosting) library.
6. To include the third-party library into your Anaconda environment is very simple. For example, to include the XGBoost library, you simply do the following via the command window:

```
conda install -c conda-forge xgboost
```