

Ensemble Methods: Gradient Boosting

AdaBoost is a foundational boosting model that trains a new weak learner to fix the misclassifications of the previous weak learner. It does this by maintaining and adaptively updating weights on training examples. These weights reflect the extent of misclassification and indicate priority training examples to the base-learning algorithm.

We now look at an alternative to convey misclassification information to a base-learning algorithm for boosting: **loss function gradients**. Recall that we use loss functions to measure how well a model fits each training example in the data set. For example, the half of the square-error loss function is $\mathcal{L}(y, \hat{y}) = \frac{1}{2} (y - \hat{y})^2$ in which y is the ground truth and \hat{y} is the predicted value.

The **gradient** of the loss function for a single example is called the **residual** and, as we'll see shortly, captures the **deviation** between true and predicted labels. This error, or residual, measures the amount of misclassification. In contrast to AdaBoost, which uses weights as a surrogate for residuals, **Gradient Boosting uses these residuals directly!** Thus, Gradient Boosting is another sequential ensemble method that aims to train weak learners over residuals (i.e., gradients).

Remark: The Gradient Boosting framework **can be applied to any loss function**, meaning that any classification, regression, or ranking problem can be “boosted” using weak learners. This flexibility has been a key reason for the emergence and ubiquity of Gradient Boosting as a state-of-the-art ensemble approach. Several powerful packages and implementations of Gradient Boosting are available (LightGBM, CatBoost, XGBoost) and provide the ability to train models on big data efficiently via parallel computing and GPUs.

1. Gradient Boosting = Gradient Descent + Boosting

In Gradient Boosting, we aim to train a sequence of weak learners that approximate the gradient at each iteration. Gradient Boosting and its successor, Newton Boosting, are currently considered state-of-the-art ensemble methods and are widely implemented and deployed for several tasks in diverse application areas.

We'll first look at the intuition of Gradient Boosting and contrast it with another familiar boosting method: AdaBoost. Armed with this intuition, we'll implement our version of Gradient Boosting to understand what is going on under the hood.

2. Intuition: Learning with Residuals

Gradient Boosting uses residuals or errors (between the true and predicted labels) to tell the base-learning algorithm which training examples it should focus on in the next iteration. What exactly is a residual? For a training example, it's simply the error between the true label and the corresponding prediction. Intuitively, a correctly classified example must have a small residual, and a misclassified example must have a large residual.

More concretely, if a classifier h makes a prediction $h(\mathbf{x})$ on a training example \mathbf{x} , a naïve way of computing the residual would be to directly measure the difference between them:

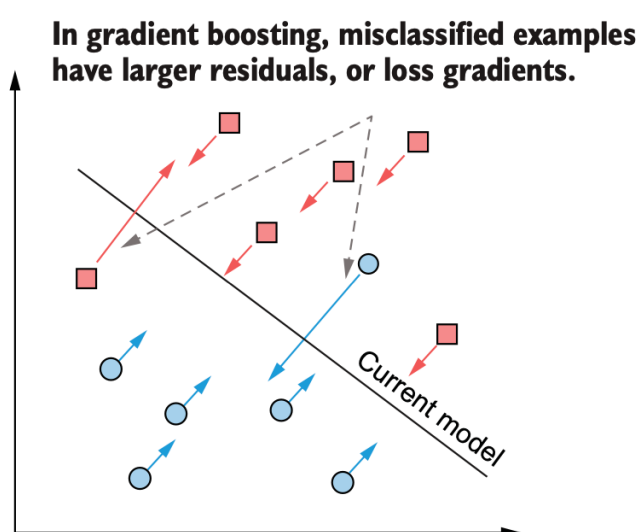
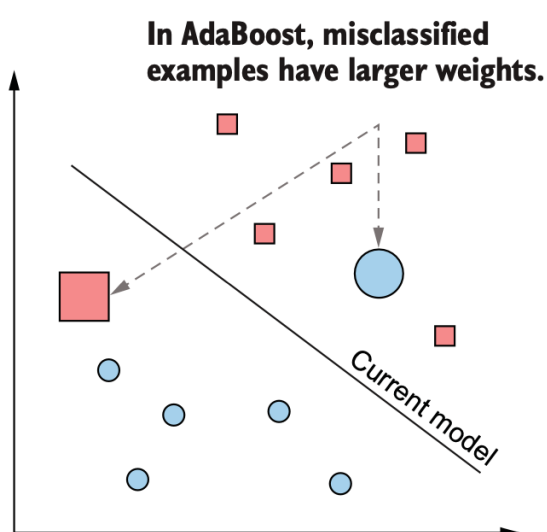
$$\text{residual} = y - h(\mathbf{x})$$

Consider the half of the square-error loss $\mathcal{L} = \frac{1}{2}(y - h(\mathbf{x}))^2$, the gradient for our model h is

$$\frac{\partial \mathcal{L}}{\partial h} = -(y - h(\mathbf{x}))$$

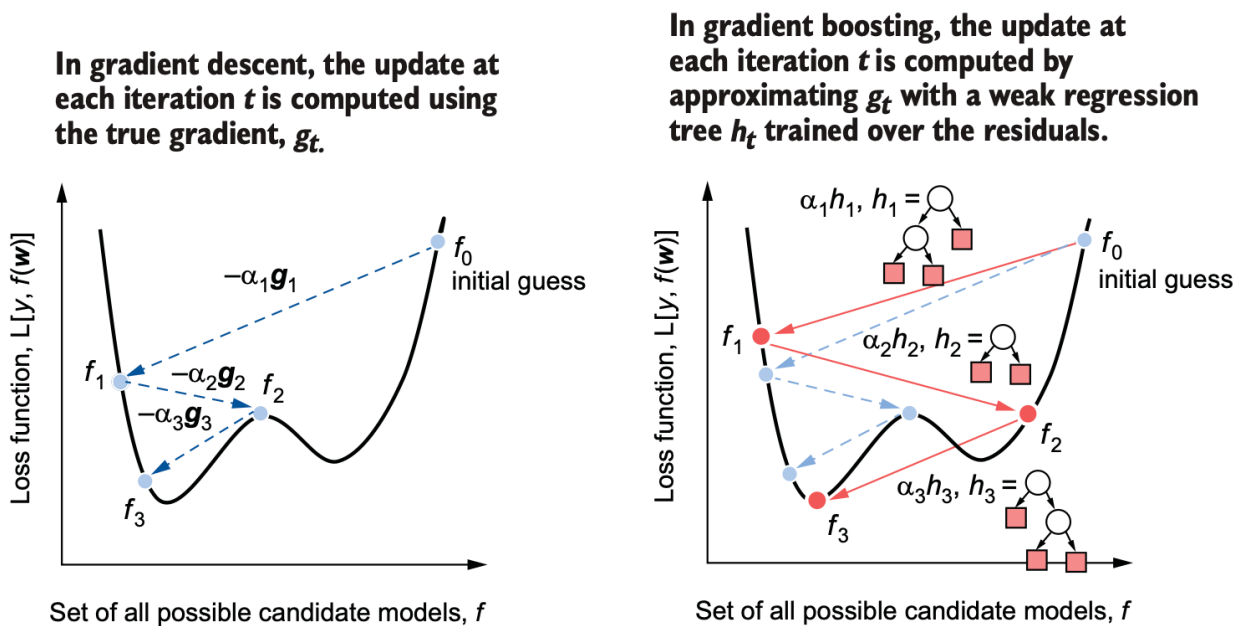
The negative gradient of the squared loss is exactly the same as our residual!

Training examples that are badly misclassified will have large gradients (residuals) as the gap between the true and predicted labels will be large. Training examples that are correctly classified will have small gradients.



In Gradient Boosting, we have example-wised residuals (or negative loss gradients), r_i , and a residual-augmented data set (\mathbf{x}_i, r_i) . Each training example now has an associated residual, which can be viewed as a real-valued label. Thus, training a weak learner in Gradient Boosting is a regression problem that requires a base-learning algorithm such as **decision-tree regression**.

When trained, weak estimators in Gradient Boosting can be viewed as approximate gradients. The figure below illustrates how Gradient Descent differs from Gradient Boosting and how Gradient Boosting is conceptually similar to Gradient Descent. The key difference between the two is that Gradient Descent directly uses the negative gradient, while Gradient Boosting trains a weak regressor to approximate the negative gradient.




Remarks:

- At each boosting iteration, AdaBoost and Gradient Boosting both update the current model using an additive expression of the following form:

$$F_{t+1}(\mathbf{x}) = F_t(\mathbf{x}) + \alpha_t h_t(\mathbf{x})$$

- The key difference between AdaBoost and Gradient Boosting is in how we compute the updates h_t and the hypothesis weights (also known as step lengths) α_t . An additional benefit of Gradient Boosting is that the algorithm is not limited to the exponential loss function.

Algorithm 	Loss function	Base-learning algorithm	$h_t(x)$	Step length α_t
AdaBoost for Classification	Exponential	Classification with weighted examples	Weak classifier	Computed in closed form
Gradient Boosting	User-specified	Regression with examples and residuals	Weak regressor	Line search

3. Implement Gradient Boosting

We now have all the ingredients to formalize the algorithmic steps of Gradient Boosting. The basic algorithm can be outlined with the following pseudocode:

```

initialize:  $F = f_0$ , some constant value
for  $t = 1$  to  $T$ :
1. compute the negative residuals for each example,  $r_i^t = -\frac{\partial L}{\partial F}(x_i)$ 
2. fit a weak decision tree regressor  $h_t(\mathbf{x})$  using the training set  $(x_i, r_i)_{i=1}^n$ 
3. compute the step length ( $\alpha_t$ ) using line search
4. update the model:  $F_t = F + \alpha_t \cdot h_t(\mathbf{x})$ 

```

The following listing implements this pseudocode specifically for the half of the squared loss. It uses a type of line search called golden section search to find the best step length.

```

def fit_gradient_boosting(X, y, n_estimators=10):
    # Initialize
    n_samples, n_features = X.shape
    estimators = []
    F = np.full((n_samples, ), 0.0)
    # Predictions of each training example using the ensemble

    for t in range(n_estimators):
        # Fit a weak learner to the residuals, which are computed as
        gradient(Loss(y, F))
        residuals = y - F
        h = DecisionTreeRegressor(max_depth=1)
        h.fit(X, residuals)

        # Compute a step length that produces the best improvement in
        the loss
        hreg = h.predict(X)
        loss = lambda a: 0.5*np.linalg.norm(y - (F + a * hreg))**2
        step = minimize_scalar(loss, method='golden')
        a = step.x

        # Update the ensemble predictions

```

```

F += a * hreg

    # Update the ensemble
    estimators.append((a, h))

return estimators

```

Once the model is trained, we can make ensemble predictions. Note that this model returns predictions of $-1/1$ rather than $0/1$.

```

def predict_gradient_boosting(X, estimators):
    pred = np.zeros((X.shape[0], )) #initialization

    for a, h in estimators:
        pred += a * h.predict(X)

    y = np.sign(pred)

    return y

```

We can test drive this implementation on a simple two-moons classification example. Note that we convert the training labels from $0/1$ to $-1/1$ to ensure that we learn and predict correctly:

```

X, y = make_moons(n_samples=200, noise=0.15, random_state=13)
y = 2 * y - 1
Xtrn, Xtst, ytrn, ytst = train_test_split(X, y, test_size=0.25,
random_state=11)

estimators = fit_gradient_boosting(Xtrn, ytrn)
ypred = predict_gradient_boosting(Xtst, estimators)

from sklearn.metrics import accuracy_score
tst_err = 1 - accuracy_score(ytst, ypred)
tst_err

```

The error rate of this model is 6%, which is pretty good.

You can download the above Python code `Gradient_Boosting.ipynb` from the course website.

4. Scikit-Learn 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 = 0.5668634601419928

We continue with the random forest with 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 = 0.8097663209351416

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` with the following considerations:

- We can control the complexity of the base tree estimators directly with `max_depth` and `max_leaf_nodes`. 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.
- `n_estimators` caps the number of weak learners that will be trained sequentially by `GradientBoostingRegressor` and is essentially the number of algorithm iterations.
- In practice, Gradient Boosting trains weak learners (h_t in iteration t) sequentially and constructs an ensemble incrementally and additively in a less aggressive manner:

$$F_{t+1}(\mathbf{x}) = F_t(\mathbf{x}) + \eta \alpha_t h_t(\mathbf{x})$$

where η is the learning rate. The learning rate is a user-defined learning parameter that lies in the range $0 < \eta \leq 1$. A slower learning rate means that it will often take more iterations to train an ensemble. It may be necessary to opt for slower learning rates to make successive weak learners more robust to outliers and noise. Learning rate is controlled by the `learning_rate` parameter.

```
# 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 the simple decision tree (0.5669), and the random forest (0.8098).

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. It 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
```