Gradient Boosting

Building and Optimizing Models with Gradient Boosting

This tutorial explains how to build and optimize models with **gradient boosting**, a method that dominates many **Kaggle** competitions and achieves `state-of-the-art` results across various datasets.

Introduction

For much of this course, you have made predictions with the random forest method, which achieves better performance than a single decision tree simply by averaging the predictions of many decision trees.

We refer to the random forest method as an **ensemble method**. By definition, **ensemble methods** combine the predictions of several models (e.g., several trees, in the case of random forests).

Next, we'll learn about another ensemble method called gradient boosting.

Gradient Boosting

Gradient boosting is a method that goes through cycles to iteratively add models into an ensemble.

It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. (Even if its predictions are wildly inaccurate, subsequent additions to the ensemble will address those errors.

How Gradient Boosting Works

At each iteration, a new weak learner (typically a small decision tree) is fit to the current errors, also called residuals or negative gradients.

1. Start with a baseline prediction, often the average of the target for regression.

2. Compute the gradient of the loss with respect to current predictions.
3. Fit a weak learner to this gradient signal.
4. Add the learner to the ensemble with a small weight controlled by the learning rate.
5. Repeat for a fixed number of iterations or until validation performance stops improving.

Formally, the model builds:

$$F_M(x) = F_0(x) + \sum_{m=1}^{M} \eta \, h_m(x)$$

where η is the learning rate and h_m are weak learners.

Residuals (negative gradients) at step m capture how predictions must change to reduce the loss:

$$r_i^{(m)} = - \left. \frac{\partial \, \mathcal{L}\big(y_i, F(x_i)\big)}{\partial F(x_i)} \right|_{F = F_{m-1}}$$

Fit the weak learner h_m to these residuals, then pick a step size via line search:

$$\gamma_m = \arg\min_{\gamma} \sum_i \mathcal{L}\Big(y_i, \; F_{m-1}(x_i) + \gamma \, h_m(x_i)\Big)$$

Update the model:

$$F_m(x) = F_{m-1}(x) + \eta \, \gamma_m \, h_m(x)$$

Then update:

Key Hyperparameters

- Learning rate (η)
    - Smaller values make each step conservative and usually require more trees but improve generalization.

- Number of estimators (M)
    - Total boosting rounds. Higher M with lower η is a common recipe.
- Tree depth / leaves
    - Shallow trees (depth 3–8) work best in boosting to capture simple interactions.
- Subsample / column sample
    - Stochasticity that reduces variance and overfitting. Typical values: 0.6–0.9.
- Regularization
    - L1/L2 penalties, min child weight, min samples per leaf, and shrinkage all help prevent overfitting.

Choosing a Loss

- Regression: squared error, Huber, quantile (for quantile regression), MAE.
- Classification: logistic loss for binary, multinomial deviance for multi-class.
- Ranking: pairwise or listwise losses (framework dependent).

Practical Training Workflow

- Split data into train and validation sets. Use early stopping to pick M automatically.
- Start with a modest learning rate ($\eta = 0.05$–0.1) and tune M via early stopping.
- Then explore depth and subsampling.
- Calibrate class weights or scale_pos_weight for imbalanced classification.
- Use feature importance and SHAP values to interpret model behavior.

Popular Implementations

- XGBoost: highly optimized, supports sparse matrices, robust regularization.
- LightGBM: histogram-based, leaf-wise growth, very fast on large and high-dimensional data.

- CatBoost: strong on categorical variables with ordered target statistics and minimal tuning.

Minimal Examples

```
# XGBoost (binary classification)
from xgboost import XGBClassifier

model = XGBClassifier(
    n_estimators=2000,        # large upper bound, rely on early stopping
    learning_rate=0.05,
    max_depth=6,
    subsample=0.8,
    colsample_bytree=0.8,
    reg_lambda=1.0,
    random_state=42,
    tree_method="hist"
)
model.fit(X_train, y_train,
          eval_set=[(X_valid, y_valid)],
          eval_metric="logloss",
          early_stopping_rounds=100,
          verbose=False)

print("Best iteration:", model.best_iteration)
```

```
# LightGBM (regression)
import lightgbm as lgb

params = dict(
    objective="rmse",
    learning_rate=0.05,
    num_leaves=63,
    feature_fraction=0.8,
    bagging_fraction=0.8,
    bagging_freq=1,
    lambda_l2=1.0,
    random_state=42
)
train_ds = lgb.Dataset(X_train, label=y_train)
valid_ds = lgb.Dataset(X_valid, label=y_valid)

model = lgb.train(
    params,
    train_set=train_ds,
    num_boost_round=5000,
    valid_sets=[valid_ds],
    valid_names=["valid"],
    early_stopping_rounds=200,
    verbose_eval=False
)
```

Troubleshooting and Tips

- If training AUC is high but validation AUC is low, reduce depth, increase regularization, or lower learning rate and increase estimators with early stopping.
- If underfitting, slightly increase depth or num_leaves and reduce regularization.
- Handle leakage carefully. Use time-aware splits for temporal data.
- Standardize evaluation with cross-validation and fixed random seeds.
- Carefully process categorical variables. Prefer CatBoost for high-cardinality categories or use target encoding with leakage-safe schemes.

When to Prefer Gradient Boosting

- Tabular data with mixed types and non-linear interactions.
- Medium to large datasets where linear models underperform and deep learning is not clearly superior.

Reason	XGBoost	LightGBM/CatBoost
Ease of Learning	High (great for beginners)	Lower (more advanced)
Use Case	General-purpose	Specialized (speed/categorical)
Preprocessing	Requires encoding for categories	Handles categories natively
Performance	Balanced	Optimized for specific scenarios