Random Forests, Gradient Boosting, and Beyond

M2 Stat SD 1

This tutorial explores two of the most powerful ensemble learning methods in contemporary statistical machine learning: random forests and gradient boosting machines. Both approaches construct predictive models by aggregating multiple decision trees, yet they do so through fundamentally distinct mechanisms. Random forests employ bootstrap aggregation (bagging) combined with feature randomisation to reduce prediction variance, whilst gradient boosting adopts a sequential learning strategy, iteratively fitting trees to residuals in order to reduce prediction bias.

The practical exercises herein will guide you through the implementation and calibration of these methods using real-world data, whilst the theoretical component will illuminate the elegant mathematical framework underlying gradient boosting—namely, its interpretation as functional gradient descent in an infinite-dimensional function space.

Learning Objectives

By the conclusion of this tutorial, you should be able to:

- Fit random forest models and calibrate the principal hyperparameter (mtry or max_features) using out-of-bag error estimation, thereby avoiding the computational expense of cross-validation
- Implement extreme gradient boosting (XGBoost) and calibrate multiple hyperparameters simultaneously through Bayesian optimisation, a principled approach that substantially outperforms naïve grid search
- Articulate the theoretical foundations of gradient boosting for both regression and classification tasks, demonstrating rigorously how the algorithm constitutes gradient descent in function space
- Compare and contrast the statistical properties of random forests and gradient boosting machines, particularly with respect to the bias-variance decomposition
- Make informed judgments regarding when each method is most appropriately deployed in practice

1 Preliminaries

1.1 Environment setup

Before commencing the practical exercises, it is essential to establish a proper computational environment. We shall create an isolated conda (or venv) environment to ensure reproducibility and avoid potential conflicts between package dependencies. Conda, as a package and environment management system, allows us to maintain separate Python installations with specific versions of libraries, thereby guaranteeing that our analyses remain reproducible across different systems and over time.

Create a new conda or venv environment and install the requisite libraries (`pandas`, `matplotlib`, `seaborn`, `numpy`, `scipy`), the `xgboost` package, the jupyter packages if required, and the `skranger` package via pip within the environment.

Implementation guidance: From your terminal, execute the following sequence of commands.

Using Conda (recommended): Create a new conda environment (replacing `env_name` with a descriptive name of your choosing, such as `ml_ensemble`):

```
conda create -n env_name python=3.10
conda activate env_name
```

Subsequently, install the core scientific computing libraries and machine learning packages:

```
conda install pandas matplotlib seaborn numpy scipy scikit-learn xgboost jupyter
```

Note that `scikit-learn` is explicitly included, as it provides the fundamental machine learning infrastructure we shall employ throughout these exercises. Finally, install the `skranger` package, which provides Python bindings to the highly efficient `ranger` random forest implementation originally developed in R:

```
pip install skranger
```

The rationale for using `pip` rather than `conda` for this particular package is that `skranger` is not available through the standard conda repositories. This hybrid approach—leveraging conda for the primary scientific stack whilst supplementing with pip for specialized packages—represents current best practice in Python environment management.

Using venv (alternative): Should you prefer Python's native virtual environment system, you may employ `venv` instead. This approach is particularly suitable when working on systems where conda is not installed or when a lighter-weight solution is desired. Navigate to your project directory and execute:

```
python3 -m venv env_name
source env_name/bin/activate  # On Unix/macOS
# OR
env_name\Scripts\activate  # On Windows
```

Once the virtual environment is activated, install all required packages via `pip`:

```
pip install pandas matplotlib seaborn numpy scipy scikit-learn xgboost jupyter skranger
```

The principal distinction between these two approaches lies in their scope and capabilities: conda manages both Python packages and system-level dependencies, rendering it particularly robust for scientific computing where binary dependencies are common, whereas `venv` provides a lightweight, Python-only isolation mechanism. For the purposes of these exercises, either approach is entirely satisfactory, though conda may prove more reliable when dealing with packages that have complex binary dependencies, such as certain scientific computing libraries.

1.2 Dataset

We shall employ the **California housing dataset** from the `sklearn` library for this practical work. This dataset, derived from the 1990 U.S. Census, comprises 20,640 observations corresponding to census block groups in California. Each block group represents the smallest geographical unit for which the U.S. Census Bureau publishes sample data, typically containing between 600 and 3,000 inhabitants.

The dataset contains **eight features**:

- median income (in tens of thousands of dollars), housing median age (in years),
- average number of rooms per dwelling,
- average number of bedrooms per dwelling,
- block group population,
- average household occupancy, and
- geographical coordinates (latitude and longitude).

The **response variable** is the median house value for households within each block group, measured in hundreds of thousands of dollars. It should be noted that the response has been capped at \$500,000, introducing a degree of censoring in the upper tail of the distribution.

This dataset presents an interesting regression problem, as the relationship between predictors and response exhibits both linear and non-linear characteristics, making it particularly suitable for comparing the performance of linear models with that of ensemble methods such as random forests and gradient boosting machines.

Implementation guidance: Load the dataset using `fetch_california_housing()` from `sklearn.datasets`. This function returns a Bunch object containing `data` and `target` attributes which must be extracted separately.

1.3 Data preparation and linear regression benchmark

Construct training and test sets using the `train_test_split` function from `sklearn.model_selection`, allocating 25% of the observations to the test set. Set a fixed `random_state` parameter to ensure reproducibility of your results.

Fit a linear regression model on the training set and evaluate its performance on the test set using the coefficient of determination R^2. You may employ the `LinearRegression` class from `sklearn.linear_model` and invoke its `score()` method to compute this metric. Consider whether ridge or lasso regularization would be warranted in this context.

2 Exercise 1: Fitting random forest models

2.1 First part

1. Fit a random forest model to the California housing dataset using the default implementation provided by `RandomForestRegressor` in `scikit-learn`, with $B = 500$ trees and default hyperparameter values.

   Implementation guidance: Import `RandomForestRegressor` from `sklearn.ensemble`. Specify `n_estimators=500` and set `oob_score=True` to enable out-of-bag error estimation. The OOB score may be accessed via the `oob_score_` attribute following model fitting.

2. Determine the number of trees required before the out-of-bag error stabilises.

   Implementation guidance: To monitor the evolution of the out-of-bag error as a function of the number of trees, one may employ the `warm_start=True` parameter in conjunction with iterative fitting, or alternatively examine the staged predictions. Consider constructing a plot of the out-of-bag error against the number of trees to visualise the convergence behavior.

3. Fit a random forest model to the California housing dataset using the `ranger` implementation with hyperparameter values identical to those employed above.

   Implementation guidance: Import `RangerForestRegressor` from `skranger.ensemble`. The syntax mirrors that of scikit-learn, with `n_estimators` and `oob_score` parameters.

4. Compare the computational time required to fit both models.

   Implementation guidance: One may employ Python's `time` module or the `%%time` magic command in Jupyter notebooks to measure execution time.

5. What are the admissible values for the `mtry` (or `max_features`) hyperparameter? Employ the out-of-bag error to calibrate this hyperparameter through an exhaustive grid search strategy.

   Implementation guidance: For regression tasks, conventional values to consider are $\sqrt{p}$, $p/3$, and $p/2$, where p denotes the number of predictor variables. In scikit-learn, the `max_features` parameter accepts either integers or floats (the latter interpreted as proportions). Fit multiple models across this grid of

candidate values and select the configuration yielding the lowest out-of-bag error. This process may be automated using either a simple iterative loop or `GridSearchCV` from `sklearn.model_selection`, though it should be noted that the latter employs cross-validation rather than out-of-bag error estimation.

6. Compare the performance of both random forest models (with default and calibrated hyperparameters) as well as the linear regression model on the test set, using the coefficient of determination R^2 as the evaluation metric.

7. Examine the variable importance plot. Which variables emerge as the most influential predictors?

   Implementation guidance: Access the `feature_importances_` attribute of the fitted model object. These importances may be visualised using a bar chart constructed with `matplotlib` or `seaborn`. Consider ordering the features by their importance values to enhance interpretability.

2.2 Second part

1. What effect does augmenting the feature set with $\widehat{Y}_{\text{LM}}$ have on model performance, where $\widehat{Y}_{\text{LM}}$ denotes the linear combination of existing features returned by the linear regression model to predict the target variable?

2. What consequences arise from introducing 8 features comprised entirely of noise (i.e., randomly generated)? What if we introduce 30 such noise features?

3. What are the implications of adding 8 features that exhibit strong correlation with the existing predictors (e.g., correlation coefficient of 0.9)? How does the model behave when 30 highly correlated features are introduced?

3 Exercise 2: Fitting XGBoost models

1. Fit an XGBoost model to the California housing dataset using the `XGBRegressor` class from the `xgboost` library with default hyperparameter values.

   Implementation guidance: Import `XGBRegressor` from `xgboost`. Fit the model using the `.fit()` method and evaluate its performance using the `.score()` method on the test set.

2. Employ Bayesian optimisation to calibrate the following hyperparameters: `n_estimators`, `max_depth`, `learning_rate`, `subsample`, `colsample_bytree`, `reg_alpha`, and `reg_lambda`. You may utilise the `BayesianOptimization` class from the `bayes_opt` library. Evaluate the performance of each hyperparameter configuration using 5-fold cross-validation.

   What is Bayesian optimisation? Bayesian optimisation represents an efficient strategy for hyperparameter tuning that frames the optimisation problem as a sequential decision-making task. In contrast to grid search or random search—methods which sample the hyperparameter space without learning from previous evaluations—Bayesian optimisation constructs a probabilistic surrogate model (typically a Gaussian process) of the objective function and employs an acquisition function to determine which hyperparameter configuration to evaluate next. This approach judiciously balances exploration of uncertain regions of the parameter space with exploitation of promising areas, rendering it particularly valuable when objective function evaluations are computationally expensive, as is frequently the case with cross-validated model performance assessments.

   Implementation guidance: First, install the `bayes_opt` library via pip. Import `BayesianOptimization` from `bayes_opt`. You must define an objective function that accepts hyperparameters as arguments, fits an XGBoost model with the specified hyperparameters, and returns the cross-validation score. Employ `cross_val_score` from `sklearn.model_selection` with `cv=5` to perform 5-fold cross-validation.

   Define the search space for each hyperparameter as a dictionary of bounds (e.g., `{'n_estimators': (50, 500), 'max_depth': (3, 10), ...}`). Note that `n_estimators` and `max_depth` must be integers, whilst

`learning_rate`, `subsample`, and `colsample_bytree` are continuous parameters constrained to the interval $(0, 1]$. The regularization parameters `reg_alpha` and `reg_lambda` are non-negative real-valued quantities.

Initialise the `BayesianOptimization` object with your objective function and parameter bounds, then invoke its `.maximize()` method with appropriate `n_iter` (number of iterations) and `init_points` (number of initial random explorations) arguments. The optimal hyperparameters may be retrieved from the `.max` attribute of the fitted optimisation object.

3. Compare the performance of both XGBoost models (with default and calibrated hyperparameters), the optimal random forest model, and the linear regression model on the test set, using the coefficient of determination R^2 as the comparative metric.

4 Exercise 3: Understanding Gradient Boosting as Gradient Descent in Function Space

This theoretical exercise aims to elucidate the **fundamental principle underlying gradient boosting algorithms**: the interpretation of boosting as gradient descent optimisation performed in function space rather than in parameter space. This perspective, first articulated by Friedman (2001), provides profound insight into why boosting algorithms converge to optimal predictors and how they may be extended to arbitrary differentiable loss functions.

4.1 Context and Motivation

In classical statistical learning, we seek to estimate a function $\hat{f}(x)$ that minimises a risk-an expected loss:

$$R(\hat{f}) = \mathbb{E}[\ell(Y, \hat{f}(X))]$$

where ℓ denotes a loss function (e.g., squared error for regression, cross-entropy for classification). In practice, we minimise the empirical risk over our training data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$:

$$\widehat{R}_n(\hat{f}) = \sum_{i=1}^{n} \ell(y_i, \hat{f}(x_i))$$

Gradient boosting constructs $\hat{f}$ as an additive expansion of base learners (typically shallow decision trees), and the optimisation proceeds via functional gradient descent. Rather than updating parameters in a finite-dimensional space, we iteratively refine the entire function $\hat{f}$ itself.

4.2 Part A: Gradient Boosting for Regression

4.2.1 Question 1: The Regression Boosting Algorithm

Consider the regression boosting algorithm presented in the slides. At iteration b, we have constructed a current estimate $\widehat{f}^{[b-1]}(x)$. The algorithm proceeds by:

1. Computing residuals: $r_i = y_i - \widehat{f}^{[b-1]}(x_i)$ for all $i = 1, \dots, n$
2. Fitting a new tree $\widehat{h}^{(b)}$ to predict these residuals from the features x_i
3. Updating the model: $\widehat{f}^{[b]}(x) = \widehat{f}^{[b-1]}(x) + \lambda \widehat{h}^{(b)}(x)$

where $\lambda \in (0, 1]$ denotes the learning rate (or shrinkage parameter).

Task: Demonstrate rigorously that this algorithm corresponds to gradient descent with step size λ for minimizing the squared error loss $\ell(y, f(x)) = \frac{1}{2}(y - f(x))^2$.

Guidance:

- Compute the gradient of $R(\widehat{f}) = \sum_{i=1}^{n} \frac{1}{2}(y_i - \widehat{f}(x_i))^2$ with respect to the function values $\widehat{f}(x_i)$
- Show that the residuals r_i equal the negative gradient
- Explain why fitting a tree to the residuals approximates the negative gradient direction
- Relate the learning rate λ to the step size in gradient descent

4.2.2 Question 2: Generalization to Arbitrary Loss Functions

The power of the gradient boosting framework lies in its applicability to any differentiable loss function $\ell(y, f(x))$.

Task: Consider an arbitrary differentiable loss function ℓ. Define the pseudo-residuals at iteration b as:

$$\tilde{r}_i^{(b)} = -\left[\frac{\partial \ell(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=\widehat{f}^{[b-1]}}$$

Explain why the gradient boosting algorithm should fit a tree $\widehat{h}^{(b)}$ to these pseudo-residuals rather than to the actual residuals. What is the interpretation of this procedure in terms of functional gradient descent?

Guidance:

- Recall that in gradient descent, we move in the direction of the negative gradient
- Explain the role of the tree as an approximation to the gradient direction
- Consider why this generalizes the squared error case from Question 1

4.3 Part B: Gradient Boosting for Binary Classification

4.3.1 Question 3: Classification with Logistic Loss

For binary classification with $y \in \{0, 1\}$, the logistic (cross-entropy) loss is given by:

$$\ell(y, f(x)) = -[y \log \sigma(f(x)) + (1 - y) \log(1 - \sigma(f(x)))]$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ denotes the logistic function, and $f(x)$ represents the log-odds (not the probability directly).

Task:

(a) Compute the gradient of this loss function:

$$\frac{\partial \ell(y, f(x))}{\partial f(x)} = \ ?$$

(b) Show that the pseudo-residuals for logistic regression boosting take the form:

$$\tilde{r}_i = y_i - \sigma(\widehat{f}(x_i))$$

(c) Interpret this expression: why do the pseudo-residuals represent the discrepancy between the observed class label and the current predicted probability?

Guidance:

- Use the chain rule carefully when differentiating the logarithms
- Recall that $\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$
- Note that $\sigma(\widehat{f}(x_i))$ represents the current estimated probability that $y_i = 1$

4.3.2 Question 4: Second-Order Information

Unlike simple regression, the second derivative of the logistic loss is not constant. Compute:

$$\frac{\partial^2 \ell(y, f(x))}{\partial f(x)^2} = ?$$

Task: Explain why more sophisticated implementations of gradient boosting for classification (such as XGBoost) utilise this second-order information. How might this lead to improved convergence properties?

Guidance:

- The second derivative provides information about the curvature of the loss function
- Consider Newton's method as an extension of gradient descent
- Recall that the second derivative $\sigma(f)(1-\sigma(f))$ represents the variance of a Bernoulli random variable

4.4 Part C: The Role of the Learning Rate

4.4.1 Question 5: Bias-Variance Trade-off in Boosting

The learning rate λ controls how aggressively the algorithm learns from each iteration.

Task:

(a) Explain qualitatively why a smaller learning rate (e.g., $\lambda = 0.01$) combined with a larger number of iterations B might yield superior test performance compared to a larger learning rate (e.g., $\lambda = 0.5$) with fewer iterations.

(b) Relate this phenomenon to the bias-variance trade-off. Does boosting primarily reduce bias or variance? How does this differ from bagging and random forests?

Guidance:

- Consider the concept of "slow learning" and its regularization effect
- Recall that boosting builds upon residuals, progressively reducing bias
- Contrast with bagging, which averages independent predictions to reduce variance

4.5 Part D: Practical Implications

4.5.1 Question 6: Why Shallow Trees?

In boosting algorithms, one typically employs shallow trees (often with depth $d = 1$ to $d = 6$) rather than deep, fully-grown trees as in random forests.

Task: Provide a theoretical justification for this practice. Consider:

- The sequential nature of boosting
- The role of individual trees as weak learners
- The curse of overfitting

Guidance:

- Recall that boosting is designed to combine many weak learners
- Deep trees have low bias but high variance
- The additive structure allows complexity to accumulate across iterations