■ **Course:** CS109A: Introduction to Data Science

■ **Lecture:** Lecture 23: Gradient Boosting

■ **Instructors:** Pavlos Protopapas, Kevin Rader, Chris Gumb

■ **Objective:** Understand boosting as an approach to reduce bias, learn how gradient boosting works, and connect it to gradient descent

## Contents

# 1 The Big Picture: A Different Approach to Ensembles

**Key Summary**

This lecture introduces **Boosting**, a fundamentally different ensemble approach from Bagging and Random Forests:

- **Random Forest**: Start with complex trees (low bias, high variance) → reduce **variance** by averaging
- **Boosting**: Start with simple trees (high bias, low variance) → reduce **bias** by iterative correction

The key insight: **Learn from your mistakes**. Each new tree corrects the errors of all previous trees.

## 1.1 Where We Are in the Course

Let's recap our journey through tree-based methods:

1. **Single Decision Trees**: Simple, interpretable, but prone to overfitting or underfitting
2. **Bagging**: Reduce variance by averaging many deep trees trained on bootstrap samples
3. **Random Forest**: Improve bagging by decorrelating trees (random feature selection)
4. **Boosting**: Take a completely different approach—reduce bias instead of variance

## 1.2 Why Boosting Matters

**Key Information**

**The Practical Importance of Boosting:**

For **tabular data** (spreadsheets, databases—not images or text), tree-based ensemble methods often outperform deep learning:

- On Kaggle competitions with structured data, boosting methods (especially XGBoost, LightGBM, CatBoost) win **almost every time**
- Random Forests and Boosting are the "go-to" methods for tabular problems
- Neural networks excel at images, text, and audio—but for tables, trees reign supreme

This is why understanding boosting is essential for any data scientist!

# 2 The Intuition Behind Boosting

## 2.1 The Opposite of Random Forest

Random Forest and Boosting take opposite approaches to the bias-variance tradeoff:

**Table 1:** *Random Forest vs. Boosting*

| Aspect | Random Forest | Boosting |
|---|---|---|
| Starting point | Deep trees (complex) | Shallow trees (simple) |
| Initial problem | High variance | High bias |
| Solution | Average many trees | Add trees sequentially |
| Goal | Reduce variance | Reduce bias |
| Training | Parallel (independent) | Sequential (dependent) |

## 2.2 The Exam Analogy

**Example: Passing a Hard Exam**

Imagine you need to pass a very difficult exam (say, you need an A to pass). You could:

**Option 1 (Bad):** Steal a time machine, go back to 1996, befriend the inventors of boosting, study with them for a decade, return to present. (Not practical!)

**Option 2 (Good):** Gather simple rules of thumb from different people:

1. **Last year's student**: "Never choose option D" → You get 60%
2. **Teaching Assistant**: "If 'overfitting' is an option, it's usually correct" → Focus on questions you got wrong, now at 65%
3. **Professor**: "Cross-validation is always a good answer" → Focus on remaining mistakes, now at 70%

**Combine the rules with appropriate weights:**

- Give more weight to more reliable rules
- Each rule is a "weak learner" (not great on its own)
- Combined, they form a "strong learner" (90%+ accuracy!)

**This is boosting!** Simple rules, learned sequentially, each fixing the mistakes of previous ones.

## 2.3 Key Concepts

**Definition: Weak Learner**

A **weak learner** is a model that performs only slightly better than random guessing. In the context of trees, this typically means a **stump**—a tree with only one split (depth 1).

The remarkable insight: combining many weak learners can create a **strong learner**!

**Definition: Boosting**

**Boosting** is an ensemble method that:

1. Uses **simple models** (weak learners) as building blocks
2. Combines them **additively**: $T_{final}(x) = \sum_h \lambda_h T_h(x)$
3. Builds models **sequentially**, with each new model correcting previous mistakes

The three keywords to remember:

1. **Simple**: Use weak learners (stumps)
2. **Additive**: Add models together
3. **Sequential**: Each model learns from previous mistakes

# 3 How Gradient Boosting Works

**Key Summary**

The core idea of gradient boosting is beautifully simple:

**Each new tree predicts the residuals (errors) of the current ensemble.**

If tree 1 predicts "too low by 5," tree 2 learns to predict "+5" to correct it.

## 3.1 The Algorithm Step by Step

**Definition: Gradient Boosting Algorithm (Regression)**

**Input**: Training data $(x_i, y_i)$, learning rate $\lambda$, number of iterations $M$

**Initialize**: $T_0(x) =$ simple model (e.g., predicting the mean of $y$)

**For** $m = 1, 2, \ldots, M$**:**

1. **Compute residuals**: $r_i^{(m-1)} = y_i - T_{m-1}(x_i)$
   (These are the "mistakes" of the current model)
2. **Fit new tree to residuals**: Train a simple tree $t_m$ to predict $r^{(m-1)}$
3. **Update model**: $T_m(x) = T_{m-1}(x) + \lambda \cdot t_m(x)$

**Output**: $T_M(x) = T_0(x) + \lambda t_1(x) + \lambda t_2(x) + \cdots + \lambda t_M(x)$

## 3.2 A Concrete Example

**Example: Gradient Boosting on Simple Data**

**Data**: 6 points with input $x$ and target $y$

**Step 1: Fit initial model $T_0$**

- Use a stump (one split)
- Find the split that minimizes MSE
- Say we split at $x = 6.5$:
  - Left region ($x < 6.5$): Predict mean $\approx 0$
  - Right region ($x \geq 6.5$): Predict mean $\approx 7.5$

**Step 2: Compute residuals**

- $r_i = y_i - T_0(x_i)$
- These show where our model is wrong

**Step 3: Fit $t_1$ to residuals**

- Train a stump to predict the residuals
- Say we split at $x = 3.5$:
  - Left region: Predict mean residual $\approx 2$
  - Right region: Predict mean residual $\approx -1.5$

**Step 4: Update model**

- Set $\lambda = 0.5$ (learning rate)

- $T_1(x) = T_0(x) + 0.5 \cdot t_1(x)$

**Step 5: Repeat**

- Compute new residuals from $T_1$
- Fit $t_2$ to these residuals
- $T_2(x) = T_1(x) + 0.5 \cdot t_2(x)$
- Continue until stopping criterion

### 3.3 Why the Learning Rate ($\lambda$)?

A natural question: Why not just add the full residual prediction? Why multiply by $\lambda < 1$?

**Important: The Learning Rate**

The learning rate $\lambda$ controls **how much we trust each new tree**.

**Problem without $\lambda$:**

- We're fitting residuals with a **weak learner** (stump)
- The stump's prediction of residuals is **imperfect**
- If we add the full prediction, we might **overcorrect**
- This is like playing whack-a-mole: fix one error, create another

**Solution with $\lambda$:**

- Take a **small step** in the right direction
- Don't fully trust any single weak learner
- Gradually converge to the correct answer
- Typical values: $\lambda = 0.01$ to $0.1$

Think of it like this: if you're not sure of the direction, take small steps rather than giant leaps!

### 3.4 Important Implementation Note

**Warning**

**We don't actually combine trees into one big tree!**

In practice:

- We store each tree separately: $T_0, t_1, t_2, \ldots, t_M$
- For prediction, we evaluate each tree and sum the results:

$$\hat{y} = T_0(x) + \lambda \cdot t_1(x) + \lambda \cdot t_2(x) + \cdots$$

- This is just numerical addition—no tree merging!

# 4 Why Is It Called "Gradient" Boosting?

This is the mathematical heart of the lecture. Understanding this connection reveals why boosting works so well.

## 4.1 Review: Gradient Descent

**Definition: Gradient Descent**

**Gradient descent** is an iterative optimization algorithm:
**Goal**: Find parameters $w$ that minimize a loss function $L(w)$
**Update rule**:

$$w_{new} = w_{old} - \lambda \cdot \nabla L(w)$$

where:

- $\nabla L(w)$ is the gradient (direction of steepest increase)
- $\lambda$ is the learning rate (step size)
- We move in the **opposite** direction of the gradient (toward minimum)

**Example: Gradient Descent Intuition**

Imagine you're blindfolded on a mountain and want to reach the valley:

1. **Feel the slope**: Figure out which direction is "up" (the gradient)
2. **Step downhill**: Move in the opposite direction
3. **Choose step size**: If the slope is steep, take a bigger step; if gentle, take a smaller step
4. **Repeat**: Keep feeling the slope and stepping until you reach the bottom

The learning rate controls your step size:

- Too large: You might overstep and end up on the other side
- Too small: You'll get there eventually, but very slowly

## 4.2 The Key Connection: Residuals ARE Gradients

Here's the magical connection that justifies the name "gradient boosting."

Consider the MSE loss function:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

Take the derivative with respect to the prediction $\hat{y}$:

$$\frac{\partial L}{\partial \hat{y}} = -(y - \hat{y})$$

**Important: The Key Insight**

The **negative gradient** of MSE with respect to predictions is exactly the **residual**:

$$-\frac{\partial L}{\partial \hat{y}} = y - \hat{y} = \text{residual}$$

Therefore:

- When we fit a tree to **residuals**, we're fitting it to the **negative gradient**
- Adding $\lambda \times$ (residual prediction) is exactly gradient descent!
- We're doing gradient descent in **function space** rather than parameter space

### 4.3 Why This Matters

Understanding the gradient descent connection gives us:

1. **Theoretical guarantee**: If the loss is convex and learning rate is small enough, gradient descent converges to the minimum. So our boosting algorithm will converge!
2. **Extension to other losses**: We can use ANY differentiable loss function:
   - MSE for regression
   - Log-loss for classification (this gives AdaBoost-like behavior)
   - Huber loss for robust regression
   - Quantile loss for quantile regression

   Just compute the negative gradient and fit trees to that!
3. **Understanding the learning rate**: The learning rate in boosting plays the same role as in gradient descent—controlling step size to ensure convergence.

### 4.4 Optimization in Function Space

**Example: Parameter Space vs. Function Space**

**Traditional ML (Linear/Logistic Regression):**

- We have parameters $\beta_0, \beta_1, \ldots$
- We optimize by adjusting these parameters
- Gradient descent moves in **parameter space**

**Gradient Boosting:**

- Trees don't have fixed parameters (depth can vary)
- We optimize the **predictions** directly
- Each tree moves us toward better predictions
- Gradient descent moves in **function/prediction space**

This is a profound shift! Instead of adjusting coefficients, we're directly adjusting predictions by adding corrective models.

# 5 Hyperparameters and Tuning

Gradient boosting has several important hyperparameters to tune.

## 5.1 Learning Rate ($\lambda$)

**Table 2:** *Effect of Learning Rate*

| Learning Rate | Pros | Cons |
|---|---|---|
| Small (0.01) | More robust, better generalization | Needs many trees, slow |
| Medium (0.1) | Good balance | Standard choice |
| Large (0.3+) | Faster training | May overshoot, overfit |

**Rule of thumb**: Use a small learning rate with many trees for best performance.

## 5.2 Number of Trees (Iterations)

Unlike Random Forest, boosting CAN overfit with too many trees:

- **Too few**: Underfitting (high bias)
- **Too many**: Overfitting (memorizing training data)
- **Solution**: Use early stopping with validation data

## 5.3 Tree Complexity

Each weak learner can have its own complexity:

- **Depth 1 (stump)**: Very weak, needs many trees
- **Depth 3-5**: Common choice, captures interactions
- **Depth 10+**: May overfit, defeats purpose of "weak" learners

## 5.4 Practical Tuning Strategy

**Key Information**

**Recommended Tuning Approach:**

1. Start with:
   - Learning rate: 0.1
   - Max depth: 3
   - N estimators: 100
2. Watch the learning curve (training vs. validation error)
3. If overfitting: Decrease learning rate, add regularization
4. If underfitting: Increase depth or number of trees
5. Final model: Use small learning rate (0.01-0.05) with early stopping

# 6 Gradient Boosting in Python

## 6.1 Basic Implementation with sklearn

```python
from sklearn.ensemble import GradientBoostingRegressor, GradientBoostingClassifier
from sklearn.model_selection import train_test_split

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Create gradient boosting regressor
gb_reg = GradientBoostingRegressor(
    n_estimators=100,       # Number of trees
    learning_rate=0.1,      # Shrinkage parameter
    max_depth=3,            # Depth of each tree (weak learner)
    min_samples_split=2,    # Minimum samples to split
    min_samples_leaf=1,     # Minimum samples per leaf
    random_state=42
)

# Train
gb_reg.fit(X_train, y_train)

# Evaluate
train_score = gb_reg.score(X_train, y_train)
test_score = gb_reg.score(X_test, y_test)
print(f"Train R^2: {train_score:.4f}")
print(f"Test R^2: {test_score:.4f}")
```

## 6.2 Monitoring Training with Staged Predict

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

# Get staged predictions
train_errors = []
test_errors = []

for i, y_pred_train in enumerate(gb_reg.staged_predict(X_train)):
    train_errors.append(mean_squared_error(y_train, y_pred_train))

for i, y_pred_test in enumerate(gb_reg.staged_predict(X_test)):
    test_errors.append(mean_squared_error(y_test, y_pred_test))

# Plot learning curve
plt.figure(figsize=(10, 6))
plt.plot(train_errors, label='Training Error')
```

```
plt.plot(test_errors, label='Test Error')
plt.xlabel('Number of Trees')
plt.ylabel('MSE')
plt.title('Gradient Boosting Learning Curve')
plt.legend()
plt.show()

# Find optimal number of trees
best_n = np.argmin(test_errors)
print(f"Optimal number of trees: {best_n}")
```

### 6.3 Early Stopping

```
from sklearn.ensemble import GradientBoostingRegressor

# Use validation_fraction and n_iter_no_change for early stopping
gb_early = GradientBoostingRegressor(
    n_estimators=500,           # Maximum trees (won't necessarily use all)
    learning_rate=0.1,
    max_depth=3,
    validation_fraction=0.1,    # Use 10% for validation
    n_iter_no_change=10,        # Stop if no improvement for 10 rounds
    tol=1e-4,                   # Tolerance for improvement
    random_state=42
)

gb_early.fit(X_train, y_train)
print(f"Trees used: {gb_early.n_estimators_}")
```

### 6.4 Feature Importance

```
import pandas as pd

# Get feature importances (based on improvement in criterion)
importances = gb_reg.feature_importances_

# Display sorted importances
feature_importance_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
}).sort_values('Importance', ascending=False)

print(feature_importance_df)

# Plot
plt.figure(figsize=(10, 6))
plt.barh(range(len(importances)), importances[importances.argsort()])
plt.yticks(range(len(importances)),
```

```
            [feature_names[i] for i in importances.argsort()])
plt.xlabel('Feature Importance')
plt.title('Gradient Boosting Feature Importance')
plt.show()
```

# 7 Comparing Ensemble Methods

**Table 3:** *Comprehensive Comparison of Tree Ensemble Methods*

| **Aspect** | **Bagging** | **Random Forest** | **Gradient Boosting** |
|---|---|---|---|
| Primary goal | Reduce variance | Reduce variance | Reduce bias |
| Tree type | Deep, complex | Deep, complex | Shallow, simple |
| Training | Parallel | Parallel | Sequential |
| Speed | Fast | Fast | Slower |
| Overfitting risk | Low | Low | Higher |
| Tuning difficulty | Easy | Easy | Harder |
| Feature randomization | No | Yes (at each split) | No |
| Tree correlation | High | Low | N/A (sequential) |
| Interpretability | Low | Low (some via importance) | Low (some via importance) |
| Performance | Good | Better | Often best |

**Key Information**

**When to Use Which:**

- **Random Forest**: When you want good results with minimal tuning. Great default choice.
- **Gradient Boosting**: When you need maximum performance and are willing to tune carefully. Especially for competitions.
- **Both**: Always try both! The best method depends on your specific data.

# 8 Key Takeaways

**Key Summary**

**Boosting Philosophy:**

- Combine many **weak learners** into one **strong learner**
- Reduce **bias** (opposite of Random Forest which reduces variance)
- **Learn from mistakes**: Each new model corrects previous errors

**Gradient Boosting Mechanics:**

- Start with a simple prediction
- Compute residuals (errors)
- Fit new tree to predict residuals
- Add new tree with learning rate: $T_{new} = T_{old} + \lambda \cdot t_{new}$
- Repeat

**The Gradient Connection:**

- Residuals ARE the negative gradient of MSE loss
- Fitting trees to residuals = gradient descent in function space
- Learning rate = step size in gradient descent
- This is why it's called "gradient" boosting!

**Hyperparameters:**

- Learning rate ($\lambda$): Small values (0.01-0.1) are safer
- Number of trees: Use early stopping to prevent overfitting
- Tree depth: Keep trees shallow (depth 3-5)

**Practical Advice:**

- Gradient Boosting often outperforms Random Forest (with tuning)
- For tabular data, tree methods often beat deep learning
- XGBoost, LightGBM, CatBoost are optimized implementations

# 9 Practice Questions

1. **Conceptual**: Explain the fundamental difference between how Random Forest and Gradient Boosting reduce prediction error. Which component of error does each target?
2. **Algorithm**: In gradient boosting, what is the "target" that each new tree is trained to predict? Why is this different from the original target $y$?
3. **Mathematical**: Show that for MSE loss $L = \frac{1}{2}(y - \hat{y})^2$, the negative gradient with respect to $\hat{y}$ equals the residual.
4. **Learning Rate**: Why do we use a learning rate $\lambda < 1$ instead of adding the full residual prediction? What would happen if $\lambda$ were too large?
5. **Overfitting**: Unlike Random Forest, Gradient Boosting can overfit with too many trees. Why is this the case? How does early stopping help?
6. **Comparison**: You have a dataset and find that:
   - Random Forest: Train accuracy 95%, Test accuracy 88%
   - Gradient Boosting: Train accuracy 99%, Test accuracy 85%

   What does this suggest about your Gradient Boosting model? What would you adjust?
7. **Code**: Write Python code using sklearn to:
   - Train a GradientBoostingClassifier
   - Use early stopping based on validation performance
   - Plot the learning curve showing train vs. validation error
8. **Extension**: Gradient boosting works with any differentiable loss function. How would the algorithm change if we used absolute error loss $L = |y - \hat{y}|$ instead of MSE? What would we fit trees to?