Gradient Boosting on the Court: XGBoost for NBA Performance Prediction

Joseph Mastromonica, Akaash Srikakulam

April 2025

Contents

1 Introduction/Cover

This project is intended to utilize machine learning, in particular a machine learning model known as XGBoost, to predict a single player statistic such as points in an upcoming NBA game. The script in its entirety takes in a user input of a current NBA player and the desired statistic to predict, updates the data of the desired player to the most current data to finally utilize the XGBoost Regression algorithm to make a final prediction regarding the desired statistic for the desired player.

2 The Script(Brief Overview)

The user provides the name of a player currently in the NBA. The script then scrapes NBA game data, which will be any statistics relating to the player such as points, assists, rebounds, etc. The data is scraped using the *nba_api* python library. The scraped data is then loaded into the XGBoost Regression model and can then be used to predict a variety of statistics for the chosen player in a upcoming game. The code for this script was developed in collaboration with the authors of this paper, along with strong guidance from AI systems, in particular ClaudeAI and DeepSeek were used to help create this script.

2.1 Lines 1-356

This section of code is used to scrape NBA data. It scrapes player data as well as team data using the NBA API and then stores it into a cache.

Figure 1: Section of Scraping Code

The data can be retrieved into a CSV file for viewing purposes.

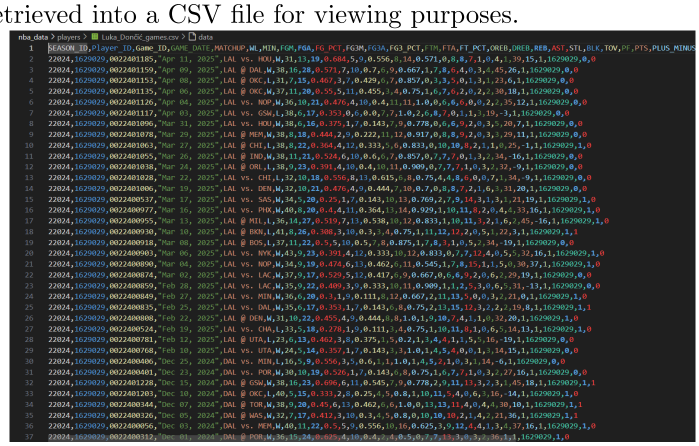

Figure 2: Example Player Data: Luka Dončić

2.2 Lines 357-600

This section of code is used for visualizations and verification of data. We have here the capability to visualize and compare either two players or teams for one statistic at a time. We also have the capability to spot check the data that has been scraped.

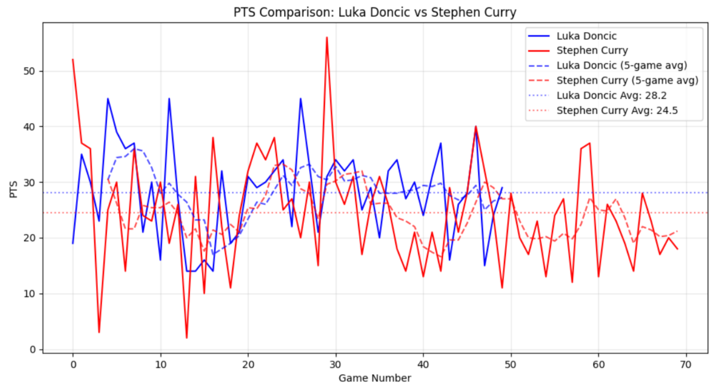

Figure 3: Luka Doncic vs. Steph Curry: Points

2.3 Lines 601-718

This section of code contains the entirety of the model. The features of the model are the different statistics that we scrape. (i.e. PTS = Points, AST = Assists, etc.)

```
def predict_next_game_points(self, player_name, visualize=True):
    """
    Simple XGBoost example to predict a player's next game points.

    Args:
        player_name (str): Name of the player to predict for
        visualize (bool): Whether to show feature importance plot

    Returns:
        dict: Prediction results and model metrics
    """
    try:
        # Get player data
        player_id = self.get_player_id_by_name(player_name)
        if not player_id:
            return {"error": f"Player {player_name} not found"}

        df = self.get_player_game_log(player_id, save=False)
        if df.empty:
            return {"error": f"No data found for {player_name}"}

        # Sort by date and prepare data
        df = df.sort_values('GAME_DATE')

        # Create features (using simple rolling averages)
        features = [
            'PTS', 'REB', 'AST', 'FG_PCT', 'MIN',
            'FGA', 'FG3A', 'FTA', 'FGM', 'FG3M', 'FTM',
            'STL', 'PLUS_MINUS'
        ]

        # Create lagged features (previous game stats)
        for feature in features:
            df[f'prev_{feature}'] = df[feature].shift(1)
```

Figure 4: Machine Learning Model Code

3 Connections to the Textbook

In the textbook *Foundations of Data Science* by Avrim Blum, John Hopcroft, and Ravindran Kannan, the boosting algorithm is defined to start by, given a sample $S \in \mathbb{R}^{n \times d}$, where d is the number of features of n labeled examples $\mathbf{x}_1, \ldots, \mathbf{x}_n$, initializing each example $\mathbf{x}_i \in \mathbb{R}^d$ to have a weight $w_i = 1$. Then, letting $\mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$ and for $t = 1, 2, \ldots, t_0$

- Call the weak learner on the weighted sample $(S, \mathbf{w})$, receiving hypothesis $h_t : \mathbb{R}^d \mapsto \{+1, -1\}$.
- Multiply the weight of each example that was misclassified by h_t by $\alpha = \frac{\frac{1}{2}+\gamma}{\frac{1}{2}-\gamma}$, where $0 < \gamma \leq \frac{1}{2}$ is the error rate of the weak learner. This error rate is a property of the weak learner (assumed, not computed) and quantifies how much better the weak learner is compared to a random guess. Leave the other weights as they are.

Output the classifier $\mathrm{MAJ}(h_1, \ldots, h_{t_0})$ which takes the majority vote of the hypotheses returned by the weak learner. Assume t_0 is odd so there is no tie.
While XGBoost uses several concepts foundational to boosting, its actual implementation differs from the mathematical definition we see above. There are four key differences in the way XGBoost performs boosting compared to the textbook definition:

(1) The weak learner,

(2) reweighting methods,

(3) the method of combining ensemble results,

(4) loss/objective function.

To further highlight the significance of these differences, we will go through them one at a time.

3.1 The Weak Learner

The textbooks nearby definition of a *weak learner* reads: "an algorithm that does just a little bit better than random guessing." It also specifies that a weak learner is only required to get a learning rate less than or equal to $\frac{1}{2} - \gamma$. However, XGBoost's weak learners are fixed-depth trees (controlled by the `max_depth` parameter). XGBoost mathematically constructs trees (deciding how the branches split) to maximize $Gain \in \mathbb{R}$. That is

$$Gain = \frac{(\sum_{i \in L} \nabla_{\hat{y}_i} L)^2}{\sum_{i \in \ L} \nabla^2_{\hat{y}_i} L + \lambda} + \frac{(\sum_{i \in R} \nabla_{\hat{y}_i} L)^2}{\sum_{i \in \ R} \nabla^2_{\hat{y}_i} L + \lambda} - \frac{(\sum_{i \in P} \nabla_{\hat{y}_i} L)^2}{\sum_{i \in \ P} \nabla^2_{\hat{y}_i} L + \lambda},$$

$$\nabla L, \nabla^2 L \in \mathbb{R}^n$$

where P is the set of all data points in the current node before a split, L is the subset of points sent to the left child after a split and R the right child. λ is the regularization term.

3.2 Reweighting Methods

In the textbook, weighting was done explicitly, where the misclassified examples received higher weights depending on their error rate. This can be thought of practically as "paying more attention to the problems you are getting wrong." This was done mathematically by multiplying the weight of each misclassified example by

$$\alpha = \frac{\frac{1}{2}+\gamma}{\frac{1}{2}-\gamma} \in \mathbb{R}$$

It is important to note that it is theoretically possible to get an undefined α when $\gamma = \frac{1}{2}$, but that implies a flawless weak learner.
In XGBoost, however, weak learners are trees instead of binary classifiers and reweighting is done through using the loss function gradient and hessian to identify the harder examples. In using gradient descent, optimal leaf weight is calculated as,

$$w^* = -\frac{\sum_{i\in\text{node}} \nabla_{\hat{y}_i} L}{\sum_{i\in\text{node}} \nabla^2_{\hat{y}_i} L + \lambda}$$

$$\nabla L \in \mathbb{R}^n$$

for which λ is the regularization term and a higher gradient value indicates a steeper rise in error, indicating a higher effective weight (scalar).

3.3 Combining Ensemble Results

In the textbook, the classifier takes the majority opinion of hypotheses after the tree has been boosted. In the case of the textbook, a hypothesis is either a 1 or a -1 since the algorithm is a simple, classifying one. To combine results, for hypotheses $h_1, \ldots, h_{t_0}$, our algorithm outputs

$$\text{MAJ}(h_1, \ldots, h_{t_0})$$

On the other hand, XGBoost uses an additive model, which is just the sum of all tree predictions and is mathematically expressed as

$$\hat{y}_i = \sum_{t=1}^{T} f_t(x_i), f_t \in \mathcal{F}$$

where T is the number of trees, and f_t is a function in the functional space $\mathcal{F}$. That is, since $f_t : \mathbb{R}^d \mapsto \mathbb{R}$, we know $\hat{y}_i \in \mathbb{R}$. It follows that for n predictions, $\hat{\mathbf{y}} \in \mathbb{R}^n$.

3.4 Loss/Objective Function

The textbook underscores that the aforementioned algorithm is for classification whereas we are using XGBoost for regression. This causes a discrepancy in the nature of the chosen objective/loss functions. As mentioned before, the algorithm in the textbook is using binary classification, meaning there are two mutually exclusive classes. XGBoost, as we know, is using MSE as its objective which we know to be the loss function plus some regulatory term

$$L_{MSE}(\theta) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \in \mathbb{R}$$

This is done when initializing the `XGBRegressor` class. By default,

```
XGBRegressor(objective='reg:squarederror')  # Default loss
```

However, this objective can be changed to other functions, like the square log error (`reg:squaredlogerror`), which looks like

$$L_{SLE}(\theta) = \frac{1}{2}\sum_{i=1}^{n}\left[log(y_i+1) - log(\hat{y}_i+1)\right]^2 \in \mathbb{R}$$

4 Connections to Classwork

XGBoost uses the MSE Loss Function as follows

$$L_{MSE}(\theta) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \in \mathbb{R}$$

We noticed pretty quickly that this was identical to the Loss Function featured in *L6w*, minus the assumption linear data. This connection gave rise to important differences in the two Loss Functions. Recall that through expanding the norm and optimization techniques, the function in *L6w* has a solution of closed-form for linear regression

$$\begin{aligned} \sum_{i=1}^{n} (y_i - \theta X)^T (y_i - \theta X) &= -\theta^T X^T X\theta + 2\theta^T X^T y - y^T y \\ \implies \frac{d\theta}{dt} &= -2X^T X\theta + 2X^T y \overset{!}{=} 0 \\ \implies \theta &= \boxed{(X^T X)^{-1} X^T y} \end{aligned}$$

$$X \in \mathbb{R}^{n\times d}, y \in \mathbb{R}^n, \theta \in \mathbb{R}^R$$

In this section, we will attempt to draw parallels between XGBoost and classwork, specifically *L6w*. We will acknowledge that both methods aim to minimize prediction error, however XGBoost's gradient-boosted trees generalize *L6w*'s linear approach by:

(1) Replacing the linear subspace with a functional space of trees,

(2) Swapping closed-form solutions for gradient-driven iterative fitting,

(3) Introducing adaptive weighting via gradients and Hessians.

4.1 Replacing Spaces

In the perspective of *L6w*, the solution space is a linear subspace spanned by the columns of X (i.e., all possible $\theta^T X$). We also have that the best-fit line is found by orthogonally projecting y onto this subspace (via normal equations, yielding

$$X^T X\theta = X^T y$$

XGBoost generalizes this process by using a functional space of trees $\mathcal{F} = \{f_t(X)\}$, where each f_t is a decision tree. These trees partition the input space into non-linear regions (analogous to basis functions in *L6w*, but adaptive to data).

Overall, In *L6w*, the basis vectors $[1, x_i]$ define a plane in $\mathbb{R}^n$, while each tree f_t in XGBoost adds a new direction in the functional space $\mathcal{F}$, refining the prediction iteratively. This is an important distinction because it shows that trees capture interactions and non-nonlinearities that the simple linear model in *L6w* cannot.

4.2 Closed-Form v.s. Gradient-Driven Iterative Fitting

As we saw in *L6w*, assuming linearity and invertibility of $X^T X$, the closed-form solution

$$\theta = (X^T X)^{-1} X^T y$$

is derived from solving the normal equations (exact projection). However, no closed-form solution exists for trees, so XGBoost uses gradient descent in a functional space. That is, starting with an initial guess $\hat{y}_i$, at each iteration t, fit a tree f_t to the negative gradient (see section 2.1)

$$-\nabla L = y_i - \hat{y}_i^{(t-1)} \qquad \text{(for MSE)}$$

Predictions are then updated with some learning rate, η, as follows (see section 2.3)

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta f_t(X_i)$$

The gradient ∇L points in the direction of steepest error reduction, analogous to how *L6w*'s residuals $y - X\theta$ could guide updates in prediction values if prediction was done iteratively. Iterative fitting generalizes *L6w*'s "one-step" projection to a sequence of corrective steps. This emphasizes that XGBoost handles nonlinearities and large-scale data where closed-form solutions are infeasible, and prediction are instead scalable via gradient boosting.

4.3 Adaptive Weighting v.s. Gradients/Hessians

We saw that in problem 2 of *L6w*, our objective function became

$$\sum_{i=1}^{n} w(x_i)(y_i - \hat{y}_i)^2$$

after introducing a positive valued function, $w(x_i)$, that weights according to the x value. After solving the normal equations, we obtain

$$X^T W X\theta = X^T W y$$

which assumes predefined weights W. One can see how regressing and then tweaking your weighting manually each step at a time can become tedious fast. Luckily, XGBoost uses weights that are dynamically adapted using gradients and hessians. Letting

$$g_i = \nabla_{\hat{y}_i} L \text{ and } h_i = \nabla^2_{\hat{y}_i} L$$

XGBoost constructs trees using the following formula for a metric they call *Gain* (see section 2.2)

$$Gain = \frac{(\sum_{i \in L} g_i)^2}{\sum_{i \in\ L} h_i + \lambda} + \frac{(\sum_{i \in R} g_i)^2}{\sum_{i \in\ R} h_i + \lambda} - \frac{(\sum_{i \in P} g_i)^2}{\sum_{i \in\ P} h_i + \lambda}$$

It became clear to us that this is analogous to weighted least squares, but weights are learned from data (via gradients) rather than pre-specified and or manually tweaked. This is an important distinction as it shows hard examples (large gradients) receive more attention, mimicking the hypothetical weighted regression in problem 2 of *L6w*. Lastly, it shows that XGBoost uses second-order optimization (with the hessians) as it accelerates convergence vs. gradient-only methods.

5 Prediction Example

This document demonstrates a highly simplified version of the XGBoost algorithm to predict Luka Dončić's points using only his minutes played and points scored from the immediately preceding game as our features(d). This example is for illustrative purposes to show the core mechanics of gradient boosting in a way that can be followed by hand.

5.1 Our Goal

Predict Luka Dončić's points in an upcoming game using only his **Minutes Played (MIN)** and **Points Scored (PTS)** from his *immediately preceding* game.

5.2 Simplified XGBoost: Predicting Luka Dončić's Points (Real Data Hand Calculation)

This illustrative example breaks down a highly simplified XGBoost prediction for Luka Dončić's points. We'll use only two features and a tiny subset of the true data to demonstrate the core concepts of gradient boosting, making the calculations traceable by hand. This example mirrors the fundamental logic discussed previously but reduces complexity for clarity.

5.2.1 Our Goal in This Example

Predict Luka Dončić's points for an upcoming game using only his **Minutes Played (MIN)** and **Points Scored (PTS)** from his *immediately preceding* game. These will be denoted as `prev_MIN` and `prev_PTS`.

5.2.2 Step 1: Establishing Our Miniature "Training" Dataset

We use a sequence of Luka Dončić's actual game data to form our training instances. The full data provided was:

- Game on 4/4: MIN 36, PTS 35
- Game on 4/6: MIN 37, PTS 30
- Game on 4/8: MIN 31, PTS 23
- Game on 4/9: MIN 38, PTS 45
- Game on 4/11: MIN 31, PTS 39 (This final game's points will serve as a comparison for our prediction)

From this, we construct 3 training sequences (X_i, y_i), where $X_i = (\texttt{prev\_MIN}_i, \texttt{prev\_PTS}_i)$ and y_i is the actual points in the subsequent game:

Sequence	`prev_MIN`	`prev_PTS`	Actual Next Game PTS (y_i)
1 (4/4 → 4/6)	36	35	30
2 (4/6 → 4/8)	37	30	23
3 (4/8 → 4/9)	31	23	45

So, our training instances are:

- $X_1 = (36, 35)$, $y_1 = 30$
- $X_2 = (37, 30)$, $y_2 = 23$
- $X_3 = (31, 23)$, $y_3 = 45$

5.2.3 Step 2: Feature Scaling – Calculating Mean & StDev from Our Training Data

To mimic the 'StandardScaler' used in the Python script (though on a vastly smaller dataset), we calculate the mean (μ) and population standard deviation (σ) for each feature *directly from our 3 training instances.*

- For `prev_MIN` values [36, 37, 31]:

$$\mu_{\text{MIN}} = \frac{36 + 37 + 31}{3} = \frac{104}{3} \approx 34.67$$

$$\sigma_{\text{MIN}} = \sqrt{\frac{(36 - 34.67)^2 + (37 - 34.67)^2 + (31 - 34.67)^2}{3}} = \sqrt{\frac{1.33^2 + 2.33^2 + (-3.67)^2}{3}}$$
$$\approx \sqrt{\frac{1.77 + 5.43 + 13.47}{3}} = \sqrt{6.89} \approx 2.62$$

- For `prev_PTS` values [35, 30, 23]:

$$\mu_{\text{PTS}} = \frac{35 + 30 + 23}{3} = \frac{88}{3} \approx 29.33$$

$$\sigma_{\text{PTS}} = \sqrt{\frac{(35 - 29.33)^2 + (30 - 29.33)^2 + (23 - 29.33)^2}{3}} = \sqrt{\frac{5.67^2 + 0.67^2 + (-6.33)^2}{3}}$$
$$\approx \sqrt{\frac{32.15 + 0.45 + 40.07}{3}} = \sqrt{24.22} \approx 4.92$$

We'll use rounded values for easier presentation: $\mu_{\text{MIN}} \approx 34.7$, $\sigma_{\text{MIN}} \approx 2.6$; and $\mu_{\text{PTS}} \approx 29.3$, $\sigma_{\text{PTS}} \approx 4.9$.

Scaled Feature Value = (Raw Value - μ) / σ. Our scaled training features, $X_{i,\text{scaled}} = (\texttt{scaled\_prev\_MIN}_i, \texttt{scaled\_prev\_PTS}_i)$:

- $X_{1,\text{scaled}} = \left(\frac{36-34.7}{2.6}, \frac{35-29.3}{4.9}\right) \approx (0.50, 1.16)$
- $X_{2,\text{scaled}} = \left(\frac{37-34.7}{2.6}, \frac{30-29.3}{4.9}\right) \approx (0.88, 0.14)$
- $X_{3,\text{scaled}} = \left(\frac{31-34.7}{2.6}, \frac{23-29.3}{4.9}\right) \approx (-1.42, -1.29)$

5.2.4 Step 3: XGBoost - Initial Prediction (F_0)

The ensemble starts with an initial prediction, typically the mean of the target variable (y_i) from the training data. Let F_i be our predictions at the i^{th} step. That is, $F_i = \hat{y}_i$

$$F_0 = \text{Average}(y_1, y_2, y_3) = \text{Average}(30, 23, 45) = \frac{98}{3} \approx 32.67$$

5.2.5 Step 4: Calculate Errors (Residuals r_1) - The Role of Gradient Descent

We calculate the difference between the actual points and our initial prediction F_0. These residuals, $r_{1,i} = y_i - F_0$, are the negative gradients of the squared error loss function $\frac{1}{2}(y_i - F_0)^2$ with respect to F_0. Our first tree will try to predict these gradients (residuals). This is where the "Gradient" in **Gradient Boosting** comes from: we are fitting a model to the (negative) gradient of our loss function.

- $r_{1,1} = 30 - 32.67 = -2.67$
- $r_{1,2} = 23 - 32.67 = -9.67$
- $r_{1,3} = 45 - 32.67 = 12.33$

5.2.6 Step 5: Build Tree 1 (f_1) - A Weak Learner The Concept of Gain

We build a simple decision tree (a "stump," which is a weak learner) to predict the residuals r_1 using our scaled features. Let's choose to split on `scaled_prev_PTS`. Scaled `prev_PTS` values for X_1, X_2, X_3 are $[1.16, 0.14, -1.29]$. Corresponding r_1 values are $[-2.67, -9.67, 12.33]$.

How a split is chosen (Gain): In a full XGBoost implementation, the algorithm would test many possible split points for each feature. For each potential split, it calculates a score called **Gain**, which measures how much the split improves the model's ability to predict the residuals (typically by reducing the sum of squared errors of the residuals within each resulting leaf). The feature and split point yielding the highest Gain is chosen. **Our simplification:** For this demo, we "eyeball" a split. Let's try: Is `scaled_prev_PTS` ≤ 0?

- If YES (Instance 3: `scaled_prev_PTS` ≈ -1.29): Residual is 12.33. Average = 12.33. This is the leaf's output value.
- If NO (Instance 1: ≈ 1.16; Instance 2: ≈ 0.14): Residuals are $-2.67, -9.67$. Average $= \frac{-2.67-9.67}{2} = \frac{-12.34}{2} = -6.17$. This is the leaf's output value.

So, Tree 1 (f_1) is: If `scaled_prev_PTS` ≤ 0, output 12.33; else output -6.17.

5.2.7 Step 6: Update Predictions (F_1) using Tree 1 - The Boosting Step

We update our predictions by adding the (scaled by learning rate) output of Tree 1. This is the **Boosting** step: combining a new weak learner to improve the ensemble. Let learning rate $\eta = 0.5$. This value for η was chosen arbitrarily and mainly for the sake of computation. The formula is $F_1(X_i) = F_0 + \eta \times f_1(X_{i,\text{scaled}})$.

- For $X_{1,\text{scaled}}$ (`scaled_prev_PTS` $\approx 1.16 > 0$): $F_1(X_1) = 32.67 + 0.5 \times (-6.17) = 32.67 - 3.085 = 29.585$
- For $X_{2,\text{scaled}}$ (`scaled_prev_PTS` $\approx 0.14 > 0$): $F_1(X_2) = 32.67 + 0.5 \times (-6.17) = 32.67 - 3.085 = 29.585$
- For $X_{3,\text{scaled}}$ (`scaled_prev_PTS` $\approx -1.29 \leq 0$): $F_1(X_3) = 32.67 + 0.5 \times (12.33) = 32.67 + 6.165 = 38.835$

5.2.8 Step 7: Calculate New Errors (Residuals r_2)

Now, we find the errors of these new predictions $F_1(X_i)$. These are $r_{2,i} = y_i - F_1(X_i)$.

- $r_{2,1} = 30 - 29.585 = 0.415$
- $r_{2,2} = 23 - 29.585 = -6.585$
- $r_{2,3} = 45 - 38.835 = 6.165$

5.2.9 Step 8: Build Tree 2 (f_2) - Another Weak Learner

We train another stump to predict the new residuals r_2. Let's use `scaled_prev_MIN`. Scaled `prev_MIN` values for X_1, X_2, X_3 are $[0.50, 0.88, -1.42]$. Corresponding r_2 values are $[0.415, -6.585, 6.165]$. **Simplified Split (No Gain Calculation):** Is `scaled_prev_MIN` ≤ 0?

- If YES (Instance 3: `scaled_prev_MIN` ≈ -1.42): Residual is 6.165. Average = 6.165.
- If NO (Instance 1: ≈ 0.50; Instance 2: ≈ 0.88): Residuals are $0.415, -6.585$. Average = $\frac{0.415-6.585}{2} = \frac{-6.17}{2} = -3.085$.

Tree 2 (f_2): If `scaled_prev_MIN` ≤ 0, output 6.165; else output -3.085.

5.2.10 Step 9: Making a Final Prediction for Luka's 4/11 Game

Luka's game on 4/9 (used to predict 4/11) had: **MIN = 38, PTS = 45**. Let this be $X_{\text{new}} = (38, 45)$.

First, scale X_{new} using the μ and σ from our training data (Step 2):

- $\texttt{scaled\_prev\_MIN}_{\text{new}} = (38 - 34.7)/2.6 = 3.3/2.6 \approx 1.27$
- $\texttt{scaled\_prev\_PTS}_{\text{new}} = (45 - 29.3)/4.9 = 15.7/4.9 \approx 3.20$

So, $X_{\text{new, scaled}} \approx (1.27, 3.20)$.

Now, pass $X_{\text{new, scaled}}$ through our 2-tree model:

1. Initial Prediction: $F_0 = 32.67$.
2. Tree 1 (f_1) Contribution: Input is $X_{\text{new, scaled}}$. Tree 1 splits on `scaled_prev_PTS`. $\texttt{scaled\_prev\_PTS}_{\text{new}} \approx 3.20$. Is $3.20 \leq 0$? No. So, $f_1(X_{\text{new, scaled}})$ outputs -6.17.
3. Tree 2 (f_2) Contribution: Input is $X_{\text{new, scaled}}$. Tree 2 splits on `scaled_prev_MIN`. $\texttt{scaled\_prev\_MIN}_{\text{new}} \approx 1.27$. Is $1.27 \leq 0$? No. So, $f_2(X_{\text{new, scaled}})$ outputs -3.085.

The final prediction $F_2(X_{\text{new, scaled}})$ is the sum of the initial prediction and the weighted contributions of all trees: $F_M(X) = F_0(X) + \sum_{m=1}^{M} \eta \cdot f_m(X)$ (General formula for M trees) For our $M = 2$ trees:

$$F_2(X_{\text{new, scaled}}) = F_0 + \eta \times f_1(X_{\text{new, scaled}}) + \eta \times f_2(X_{\text{new, scaled}})$$

$$F_2(X_{\text{new, scaled}}) = 32.67 + 0.5 \times (-6.17) + 0.5 \times (-3.085)$$

$$F_2(X_{\text{new, scaled}}) = 32.67 - 3.085 - 1.5425$$

$$F_2(X_{\text{new, scaled}}) = 32.67 - 4.6275 = 28.0425$$

Our simplified XGBoost model predicts Luka will score approximately **28.04 points** in the 4/11 game. (Luka's actual points on 4/11 were 39. Our highly simplified model has a noticeable error, which is expected given the simplifications.)

6 References

References

[1] Starmer, Josh. “XGBoost Series.” YouTube, YouTube, 16 Dec. 2019, www.youtube.com/watch?v=OtD8wVaFm6E&t=191s.

[2] Blum, Avrim, et al. Foundations of Data Science. 4 Jan. 2018, www.cs.cornell.edu/jeh/book.pdf.

[3] “Introduction to Boosted Trees.” Introduction to Boosted Trees - Xgboost 3.0.1 Documentation, https://xgboost.readthedocs.io/en/release_3.0.0/tutorials/model.html. Accessed 14 Apr. 2025.

[4] Chen, Tianqi, and Tong He. Xgboost: eXtreme Gradient Boosting, 22 Apr. 2025, https://cran.ms.unimelb.edu.au/web/packages/xgboost/vignettes/xgboost.pdf.