



## Scrabble Player Rating Prediction

--

Predict players' ratings based on Woogles.io gameplay

Allison Liu , Dev Rao, Harsha Podapati, Praveen  
Kumar G, Nick Seeman

MSBA 6421  
Dr. Yicheng Song



## Table of Content

### **I. Business Context & Project Definition**

- 1.1 What Is the Problem?
- 1.2 What Is Project Goal?

### **II. Technical Specifications & Solution Overview**

- 2.1 Which Tools are Used?
- 2.2 Which Datasets are Used?

### **III. Feature Engineering**

- 3.1 Methodology
- 3.2 Data Preparation
  - 3.2.1 Joining Datasets
  - 3.2.2 Dealing With Outliers
- 3.3 Feature Selection & Engineering
  - 3.3.1 Board Metrics
  - 3.3.2 Game Metrics
  - 3.3.3 Historical Player Metrics
  - 3.3.4 Feature Selection

### **IV. Predictive Modeling**

- 4.1. Modeling
  - 4.1.1 Decision Tree
  - 4.1.2 Random Forest
  - 4.1.3 Neural Networks
  - 4.1.4 XGBoost
  - 4.1.5 LightGBM
  - 4.1.6 Stacking
- 4.2. Final Model Selection

### **V. Business Value & Future Improvements**

- 6.1 Business Value
- 6.2 Future Improvements

# I. Business Context & Project Definition

---

## 1.1 What Is the Problem?

Scrabble is a word game where players utilize letter tiles to create words on a board. The game can be played by 2, 3, or 4 players, but in this project, we are going to be looking at Scrabble being played by 2 players. Each player gets a set of 7 tiles which they can use each turn. The players take turns forming words from their set of tiles. Each letter tile has a point value, the goal of the game is to create words that maximize the total points. The game continues until all the tiles are used or players agree to end it. The winner is the player with the highest total score when the game concludes. Scrabble requires a combination of vocabulary and tile placement strategy. It is played by all age groups for recreational and competitive purposes. To ensure consistent play amongst games, the Scrabble associations has a comprehensive word list that is common among all the games.

## 1.2 What Is Project Goal?

In this project, based on Woogles.io datasets we are tasked with finding the rating of human players before a game is played. The dataset includes 72773 games played by three bots on Woogles.io BetterBot (beginner), STEEBot (intermediate), and HastyBot (advanced). The games are between the bots and human users. We will be using metadata about the games and specific information about the turns in each game to predict the rating of the human opponents. The rating provided from our model will be evaluated by RMSE.

## II. Technical Specifications & Solution Overview

---

### 2.1 Which Tools are Used?

The majority of this project is completed using **python**, including data cleaning, model building, training, testing, and validation. The packages used are as follow:

- Pandas
- Matplotlib
- Tensorflow
- Sklearn
- Keras
- RandomForest
- Lightgbm , XGboost

### 2.2 Which Datasets are Used?

There is a csv file for all the games played, which includes data about who was involved in the game, which player went first, the winner, game type, time taken and so on. The primary key of game id is used to connect to the turns.csv file. The turns.csv contains information about every single turn that was played for all games. Every turn has information about the player's rack, which tiles were played, what location the word was played at, and so on. There is detailed information which was enough to even recreate the game turn by turn. This became effective for us to go through the games and explore for more nuanced features.

1. games.csv – metadata about games (e.g., who went first, time controls)
2. turns.csv – all turns from start to finish of each game
3. train.csv – Final scores and ratings for each player in each game; ratings for each player are as of BEFORE the game was played
4. test.csv – final scores and ratings for each player in each game; ratings for each player are as of BEFORE the games were played.

# III. Feature Engineering

---

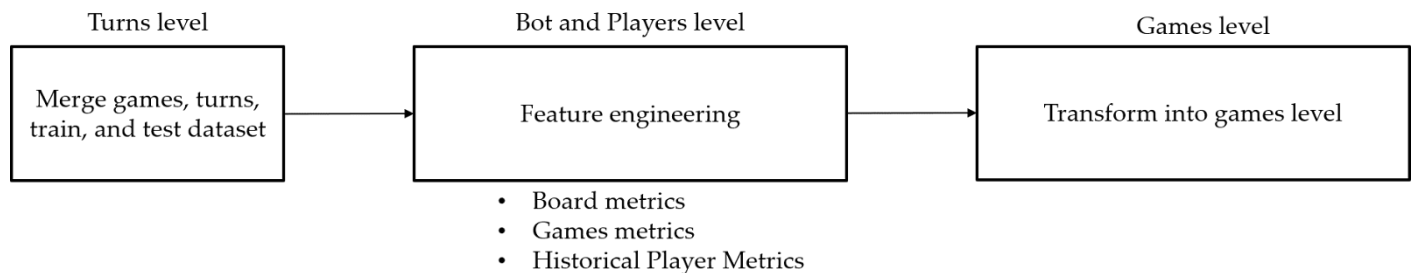
## 3.1 Methodology

As many of the group members were familiar with Scrabble, our collective experiences led us to look for additional features that were not present in the original dataset. First was to recreate the game and try to find any new features from it. We first recreated the board which allowed us to find that some spaces on the board have special purposes which higher rated players would frequently use. Second was to use a player's past performance and use them as indicators for their next games.

The given data was very well structured and had no null values (other than the word NULL used in a game!), but because we had a condition that the player's rating would only increase for rated games. Each game a player plays can be of two types, casual or rated. While exploring the data, some players played a disproportionate number of casual games.

Once the new features were formed, we consolidated all the necessary features into one file and used that as our new dataset.

Following is the workflow of merging dataset and feature engineering:



We first merged the raw datasets together to get a singular dataset to look at the data.

games_turns_merged_train											
	game_id	first	time_control_name	game_end_reason	winner	created_at	lexicon	initial_time_seconds	increment_seconds	rating_mode	max_ove
0	1	BetterBot	regular	STANDARD	1	2022-08-26 03:38:49	NWL20	1200	0	CASUAL	
1	1	BetterBot	regular	STANDARD	1	2022-08-26 03:38:49	NWL20	1200	0	CASUAL	
2	1	BetterBot	regular	STANDARD	1	2022-08-26 03:38:49	NWL20	1200	0	CASUAL	
3	1	BetterBot	regular	STANDARD	1	2022-08-26 03:38:49	NWL20	1200	0	CASUAL	
4	1	BetterBot	regular	STANDARD	1	2022-08-26 03:38:49	NWL20	1200	0	CASUAL	
...	...	...	...	...	...	...	...	...	...	...	...
2005493	72773	HastyBot	regular	STANDARD	1	2022-08-27 09:13:08	CSW21	1200	0	RATED	
2005494	72773	HastyBot	regular	STANDARD	1	2022-08-27 09:13:08	CSW21	1200	0	RATED	
2005495	72773	HastyBot	regular	STANDARD	1	2022-08-27 09:13:08	CSW21	1200	0	RATED	
2005496	72773	HastyBot	regular	STANDARD	1	2022-08-27 09:13:08	CSW21	1200	0	RATED	
2005497	72773	HastyBot	regular	STANDARD	1	2022-08-27 09:13:08	CSW21	1200	0	RATED	

2005498 rows × 22 columns

## 3.2 Data Preparation

### 3.2.1 Joining Datasets

The first step is to join 4 datasets together. The reason we merged train and test data is that it's important for us to do column transformation within training and testing data, aligning all processes before predicting. Once that is done, we have started feature engineering from different perspectives.

## 3.2.2 Dealing With Outliers

To address the challenges posed by the diverse nature of the training dataset in the competition, which includes around 70 thousand games, we recognized the need to mitigate issues related to sparse combinations and potential overfitting. The dataset exhibits variations in rules, time settings, opponents, and allowable words, influencing scoring capabilities and the performance of players in different game types.

A notable concern revolves around the distinction between ranked and casual games. Players are assigned a uniform 1500 rating upon registration. However, those exclusively engaging in casual games may not see their rating reflect their true skill level. This is exemplified by some players, like BB-8, who accumulated over 6000 casual games with a static 1500 rating, potentially masking their actual proficiency.

To tackle this issue, we strategically grouped sparse options in the training set, preventing the model from fixating on specific combinations and reducing the risk of overfitting. Given that the top 10 frequent players constitute a significant portion of the training data (20%), with BB-8 alone contributing a substantial number of casual games, there was a concern that the model might erroneously generalize based on the attributes of such players, leading to inaccurate ratings for similar playing styles.

In our approach, we sought to balance the benefits of having more data for training with the potential pitfalls associated with overemphasizing certain player profiles. By addressing outliers and grouping sparse options, we aimed to promote a more robust and generalized training set. Furthermore, in handling heavy players in the test set, we considered their past participation in rated games, allowing us to infer that their ratings may be more reflective of their true playing strength. This approach aimed to enhance the model's ability to provide accurate and meaningful ratings across a diverse range of gameplay scenarios.

## 3.3 Feature Selection & Engineering

In our modeling endeavors, we have incorporated feature engineering – a critical facet of machine learning that can enhance the predictive nature of our models. Feature engineering involves the transformation and creation of new features from the raw dataset. This allows us to extract valuable insights and patterns that may otherwise remain latent. By finding different features we aim to equip the models with a more nuanced understanding of the underlying data. We have approached the problem from three main sides, looking at information from the board, player and even the player's historical information from past games.

### 3.3.1 Board Metrics

We tried to use the games and turns dataset to simulate a working part of the entire game. To play the games, we decided to create a board and find if the spaces on the board can provide any extra information.

From the "turns" dataset, we have details of every move played between the player and the bot turn wise for a particular game. We built the simulation of this board play for every game to extract board level metrics. Let's look at how to construct the board from the turn's dataset.

Turns dataset has the column called move which contains of the set of letters the player uses for the play, it is the move they play. The information includes the tiles played, but also the position and orientation of the played word.

Scrabble is a 15 x 15 board, where the rows are indexed numbers and columns as letters. The orientation and location values denote whether it is filled across or vertically. Below are the steps that were carried out to build the board.

Step 1: Initialize the board:

This line creates a 2D list (board) initialized with empty strings. It represents a 15x15 crossword puzzle board.

```
board = [['' for _ in range(15)] for _ in range(15)]
board

[['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
 ['', '', '', '', '', '', '', '', '', '', '', '', '', '', '']]
```

*Step 2: This loop iterates over non null rows and extracts information about the move (word) and its location from the current row.*

```
for i, row in temp.dropna().iterrows():
    move = row['move']
    loc = row['location']

    col_index = ''.join([char for char in loc if char.isalpha()])
    row_index = ''.join([str(int(char)) for char in loc if char.isdigit()])

    if loc[0].isdigit():
        fill = "across"
    else:
        fill = "vertical"

    print(f"move: {move}, location: {loc}, col_index: {col_index}, row_index: {row_index}, fill : {fill}")
```

Col\_index, row\_index extracts the column index (letters) and row index (numbers) from the location and the way that location variable is arranged determines how the word is filled whether across (horizontal) or vertical.



```

move: DIG, location: 8G, col_index: G, row_index: 8, fill : across
move: HAP, location: 7H, col_index: H, row_index: 7, fill : across
move: LUTE, location: 6I, col_index: I, row_index: 6, fill : across
move: UM, location: 5K, col_index: K, row_index: 5, fill : across
move: ..DICATE, location: L5, col_index: L, row_index: 5, fill : vertical
move: OXO, location: M11, col_index: M, row_index: 11, fill : vertical
move: RAJAS, location: 5E, col_index: E, row_index: 5, fill : across
move: FAERIES, location: 9B, col_index: B, row_index: 9, fill : across
move: NEURON.L, location: C3, col_index: C, row_index: 3, fill : vertical
move: HO, location: 6F, col_index: F, row_index: 6, fill : across
move: QI, location: B2, col_index: B, row_index: 2, fill : vertical
move: KE., location: 8A, col_index: A, row_index: 8, fill : across
move: WEB, location: N13, col_index: N, row_index: 13, fill : vertical
move: rELIVED, location: O8, col_index: O, row_index: 8, fill : vertical
move: TIP, location: M3, col_index: M, row_index: 3, fill : vertical
move: CU., location: 15L, col_index: L, row_index: 15, fill : across
move: W..GLET, location: 3A, col_index: A, row_index: 3, fill : across
move: BAITER, location: 2F, col_index: F, row_index: 2, fill : across
move: MOSSY, location: N2, col_index: N, row_index: 2, fill : vertical
move: FAIN, location: 1G, col_index: G, row_index: 1, fill : across
move: NAY, location: O1, col_index: O, row_index: 1, fill : vertical
move: D.NNA, location: 7B, col_index: B, row_index: 7, fill : across
move: .ITZ, location: E9, col_index: E, row_index: 9, fill : vertical
move: RE., location: 12C, col_index: C, row_index: 12, fill : across
move: V.GO., location: 8K, col_index: K, row_index: 8, fill : across
move: LOR, location: K12, col_index: K, row_index: 12, fill : vertical

```

```
# Display the filled board in a grid
for row in board:
    print('\t'.join(row))
```

W	Q		G	L	B	F	A	I	N	R		T	M	N
	.				E	A	T					P	O	Y
		E		R	A	J		S	U	U	.		S	
K	D	.	N	N	H	O	A	L	P	T	D	G	S	.
	E	.	E	.	A	D	H	A		V	C	O	O	E
	F	.		I	I	E	I	G		L	A	X	W	I
		R	E	T						O	T	O	E	V
				.						R	C	U	.	D

We found that the skill ability of a player is highly related to the strategies players use to utilize special board squares. Some special board squares as double word, double letter, triple word, and triple letters and gives additional points to the player. We simulated the board game using ‘location’ information in the turns.csv dataset to build a board. Thus, we calculate how many special moves that players achieved to evaluate players’ level.

W	Q	.	G	L	B	F	A	I	N	R			M	N
	.	E			E	A	I	T	E			T	O	A
		U		R	A	J	A	S		U	.	I	S	Y
K	D	.	N	N	A	O	H	L	U	T	.	P	S	
	E	.			A		I	A	P		D	G	O	.
	F	L	E	.	I	E	S	G		V	C			E
				I							A			l
		R	E	T							T	O		I
				.						L	E	X	W	V
										O		O	E	E
										R	C	U	.	D

Bot\_Double\_Word 9 Bot\_Triple\_Word 2 Bot\_Double\_Letter 5 Bot\_Triple\_Letter 2 \
 0

Player\_Double\_Word 3 Player\_Triple\_Word 3 Player\_Double\_Letter 6 \
 0

Player\_Triple\_Letter 3
 0

### 3.3.2 Game Metrics

According to turns.csv dataset, we found that there is additional information that could be extracted between player turns. So, we wanted to calculate and add this other information as features into our final dataset.

#### Turns Count

We count how many times a player plays in each game; the number of players' turns has moderate correlation to player rating. A higher rated player tends to use fewer moves than lower rated players.

##### Count turns number by game\_id and nickname

```
turns_count = data[data['turn_type'] != 'End'].groupby(['game_id', 'nickname'])['turn_number'].count().reset_index()
turns_count.rename(columns={'turn_number': 'turns_count'}, inplace=True)
turns_count
```

	game_id	nickname	turns_count
0	1	BetterBot	13
1	1	stevy	13
2	2	BetterBot	12
3	2	Super	13
4	3	BetterBot	13
...	...	...	...
145541	72771	HastyBot	15
145542	72772	BetterBot	14
145543	72772	Gtowngrad	15
145544	72773	HastyBot	13
145545	72773	adola	12

145546 rows × 3 columns

#### Maximum and minimum points

We calculate players' maximum and minimum points in a game to state if he gets more points in each turn has an influence on rating score.

```
max_point = data.groupby(['game_id', 'nickname'])['points'].max().reset_index()
max_point.rename(columns={'points': 'max_point'}, inplace=True)
max_point
```

	game_id	nickname	max_point
0	1	BetterBot	68
1	1	stevy	98
2	2	BetterBot	85
3	2	Super	94
4	3	BetterBot	76
...	...	...	...
145541	72771	HastyBot	93
145542	72772	BetterBot	81
145543	72772	Gtowngrad	67
145544	72773	HastyBot	39
145545	72773	adola	90

145546 rows × 3 columns

```
min_point = data.groupby(['game_id', 'nickname'])['points'].min().reset_index()
min_point.rename(columns={'points': 'min_point'}, inplace=True)
min_point
```

	game_id	nickname	min_point
0	1	BetterBot	8
1	1	stevy	2
2	2	BetterBot	8
3	2	Super	0
4	3	BetterBot	0
...	...	...	...
145541	72771	HastyBot	0
145542	72772	BetterBot	9
145543	72772	Gtowngrad	4
145544	72773	HastyBot	4
145545	72773	adola	5

145546 rows × 3 columns

## Bingo and word length

Bingo is crucial for a player to get a high score and rating, so we calculate average bingo in each game. The length of a letter is also important for players to arrange how to play and get high scores.

```
data['bingo'] = data['word_len'].apply(lambda x: 1 if x == 7 else 0)
```

```
avg_bingo = data.groupby(['game_id', 'nickname'])['bingo'].mean().reset_index()
avg_bingo.rename(columns={'bingo': 'avg_bingo'}, inplace=True)
avg_bingo
```

	game_id	nickname	avg_bingo
0	1	BetterBot	0.076923
1	1	stevy	0.142857
2	2	BetterBot	0.250000
3	2	Super	0.214286
4	3	BetterBot	0.076923
...	...	...	...
145541	72771	HastyBot	0.133333
145542	72772	BetterBot	0.142857
145543	72772	Gtowngrad	0.125000
145544	72773	HastyBot	0.000000
145545	72773	adola	0.166667

145546 rows × 3 columns

```
avg_word_len = data.groupby(['game_id', 'nickname'])['word_len'].mean().reset_index()
avg_word_len
```

	game_id	nickname	word_len
0	1	BetterBot	4.000000
1	1	stevy	3.357143
2	2	BetterBot	4.250000
3	2	Super	3.428571
4	3	BetterBot	3.538462
...	...	...	...
145541	72771	HastyBot	3.333333
145542	72772	BetterBot	3.785714
145543	72772	Gtowngrad	2.812500
145544	72773	HastyBot	3.571429
145545	72773	adola	4.000000

145546 rows × 3 columns

### Count special letter – JQXZ

Special letters contain J and X – 8 points, Q and Z – 10 points. We calculate how many special players used in a game.

```
letters_to_count = ['J', 'Q', 'X', 'Z']
for letter in letters_to_count:
    data[f'{letter}_count'] = data[data['turn_type'] == 'Play']['move'].apply(lambda x: str(x).count(letter) if pd.notna(x) else 0)
```

```
data['total_count'] = data.apply(lambda row: sum(row[f'{letter}_count'] for letter in letters_to_count), axis=1)
```

```
special_letter = data.groupby(['game_id', 'nickname'])['total_count'].sum().reset_index()
special_letter.rename(columns={'total_count': 'count_letters_JQXZ'}, inplace=True)
special_letter
```

	game_id	nickname	count_letters_JQXZ
0	1	BetterBot	3.0
1	1	stevy	1.0
2	2	BetterBot	3.0
3	2	Super	1.0
4	3	BetterBot	0.0
...	...	...	...
145541	72771	HastyBot	1.0
145542	72772	BetterBot	3.0
145543	72772	Gtowngrad	1.0
145544	72773	HastyBot	2.0
145545	72773	adola	2.0

145546 rows × 3 columns

### Rack used rate

Rack used rate is also important to evaluate how a player appropriately used their tiles until the game ends. Thus, we divide length of move by length of rack to differentiate level of players.

```
data['rack_used_rate'] = data['word_len'] / data['rack_len']
data
```

```
rack_used = data.groupby(['game_id', 'nickname'])['rack_used_rate'].mean().reset_index()
rack_used
```

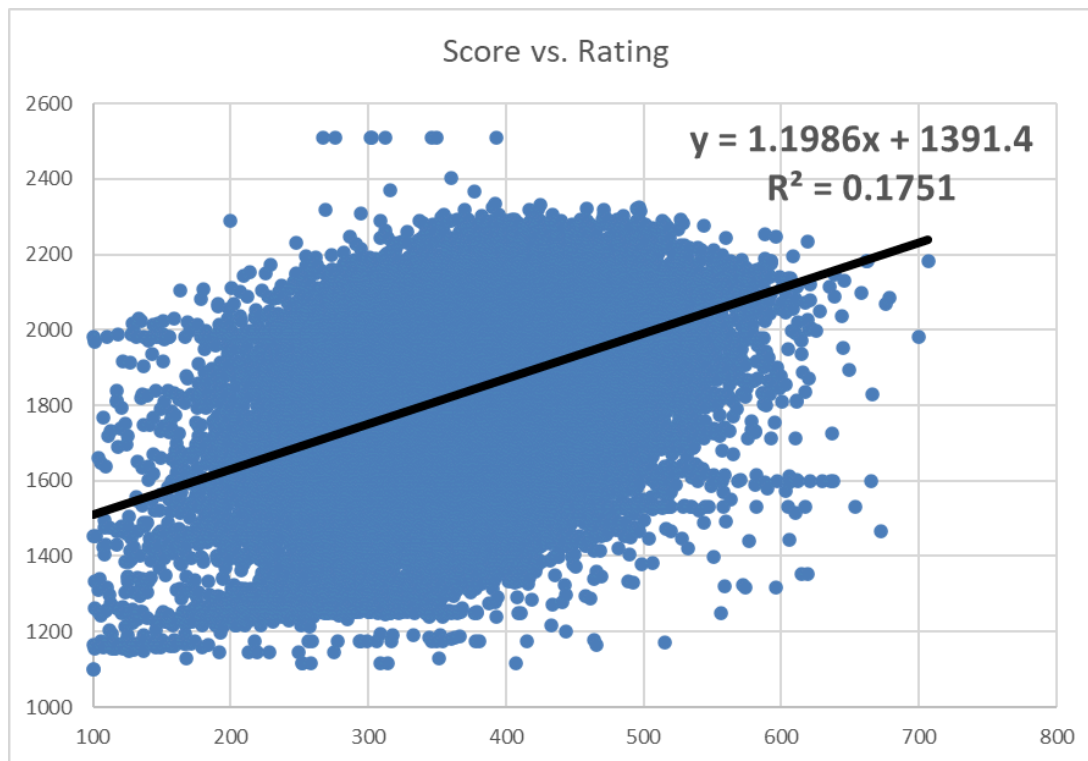
	game_id	nickname	rack_used_rate
0	1	BetterBot	0.596154
1	1	stevy	0.569231
2	2	BetterBot	0.617063
3	2	Super	0.623377
4	3	BetterBot	0.547619
...	...	...	...
145541	72771	HastyBot	0.517857
145542	72772	BetterBot	0.540816
145543	72772	Gtowngrad	0.485714
145544	72773	HastyBot	0.598901
145545	72773	adola	0.585714

145546 rows × 3 columns

### 3.3.3 Historical Player Metrics

In our feature engineering efforts, we focused on leveraging the intrinsic patterns within the players' past performance data to derive meaningful insights for predicting their next ratings. Specifically, we adopted a meticulous approach centered around the unique dynamics of a game, where the outcome of each match contributes to the player's overall skill level. Our methodology involved considering the last 10 games played by each participant, employing a tailored algorithm that considers the previous played opponents' rating. By applying the formula  $(\text{average opponent rating}) + (\text{wins} - \text{losses}) / 10$ , we derived a predictive measure for the player's next rating.

A very direct attempt to solve our problem here begins with a simple regression of player rating vs. player score. The data makes quite a bit of noise, but the overall trend is obvious. Roughly, each additional point a player scores on average correlates to 1.2 extra rating points.



One key to this is that the strength of the opponent has an impact on the player's ability to score points as well. Stronger opponents are able to capitalize on high-scoring squares for themselves and block those squares for their counterparts. Directly measuring the blocking is difficult, as it requires deep in-game strategic analysis, so we focused merely on the aspects within the player's control during their turn.

The in-game metrics that we calculated, such as word length, bingos, etc., were averaged for all games. This helps reduce the variance of these metrics for a player, as some of these events do not happen frequently within each game. Getting a broader sense of the players' skill was most important for this task, therefore by using more information (outside of the individual game we are estimating) about the player we found better model performance.

To implement this feature engineering strategy, we utilized the provided code. The `historical_average_rating` function takes a sorted dataframe as input, iterates through the games, and calculates a historical average rating for each player. The algorithm considers the last 10 games, adjusts for scenarios where there are fewer than 10 games played, and updates the rating based on win-loss differentials. The resulting historical player scores are stored in a dataframe, incorporating game-specific details. By employing this code, we aim to enhance the accuracy and relevance of our model in capturing the intricate nuances of player progression within the context of the game.

```
In [6]: # Creating historical average rating
import queue
import statistics

def historical_average_rating(df_sorted):
    base_rating = 1700

    score_df = pd.DataFrame(columns=['hist_player_score', 'game_id'])
    #Iterating through the games
    counter = 0
    group_count = 0
    games = 10
    for name, group_df in df_sorted:
        counter += 1

        # Initializing variables
        win = 0
        loss = 0
        games_played = 0

        # Creating a new queue
        oar_queue = queue.Queue()
        wins_queue = queue.Queue()
        for i in range(10):
            oar_queue.put(1700)
            wins_queue.put(0)

        for index, row in group_df.iterrows():

            # Getting details for 400(w-L)/g
            games_played = row['Player_game'] - 1
```

```

            # Adding fake games when less than 10, to have data
            if games_played == 0:
                fake_games = 10
                win_fake = 5
                loss_fake = 5
                win = 0
                loss = 0
            elif games_played < 10:
                fake_games = games-games_played
                win_fake = fake_games/2
                loss_fake = fake_games/2
                prev_10_win = get_prev_10_wins(wins_queue)
                prev_10_loss = games_played-prev_10_win
                win = prev_10_win + win_fake
                loss = prev_10_loss + loss_fake
            else:
                prev_10_win = get_prev_10_wins(wins_queue)
                prev_10_loss = games-prev_10_win
                win = prev_10_win
                loss = prev_10_loss

            #print(index)
            oar_score = get_oar_score(oar_queue)
            # Doing the calculation
            hist_score = oar_score + ((400)*(win-loss))/games

            score_df = pd.concat([score_df, pd.DataFrame({'hist_player_score': [hist_score], 'game_id':
            # After current game, update the variable with current game details
                # Updating bot rating, win/loss
                oar_queue.put(row['Bot_rating'])
                removed_element = oar_queue.get()
                wins_queue.put(row['Player_win'])
                removed_element = wins_queue.get()

        return score_df
def get_oar_score(queue):
    values_list = list(queue.queue)
    oar = statistics.mean(values_list)
    return oar
    #opp_avg_rating = get_opp_avg_rating()

def get_prev_10_wins(wins_queue):
    result_sum = 0
    result_sum = sum(item for item in wins_queue.queue)

    return result_sum
```

A final set of all the new features we found is showcased below.



game_id	int64
first	object
time_control_name	object
game_end_reason	object
lexicon	object
initial_time_seconds	int64
increment_seconds	int64
rating_mode	object
max_overtime_minutes	int64
game_duration_seconds	float64
Bot_nickname	object
Player_nickname	object
Bot_turns_count	int64
Player_turns_count	int64
Bot_max_point	int64
Player_max_point	int64
Bot_min_point	int64
Player_min_point	int64
Bot_avg_bingo	float64
Player_avg_bingo	float64
Bot_word_len	float64
Player_word_len	float64
Bot_count_letters_JQXZ	int64
Player_count_letters_JQXZ	int64
Bot_rack_used_rate	float64
Player_rack_used_rate	float64
Bot_win	int64
Player_win	int64
Bot_game	int64
Player_game	int64
Bot_score	int64
Player_score	int64
Bot_rating	int64
Player_rating	float64
created_at	object
hist_player_performance	float64
Bot_Double_Word	int64
Bot_Triple_Word	int64
Bot_Double_Letter	int64
Bot_Triple_Letter	int64
Player_Double_Word	int64
Player_Triple_Word	int64
Player_Double_Letter	int64
Player_Triple_Letter	int64
hist_player_score_20	float64
last_game_win_flag	int64
Player_max_point_player_wise	int64
Player_avg_bingo_player_wise	float64
Player_word_len_player_wise	float64
Player_score_player_wise	int64
hist_player_score_player_wise	float64
Player_Double_Word_player_wise	float64
Player_Triple_Word_player_wise	float64
Player_Double_Letter_player_wise	float64
Player_Triple_Letter_player_wise	float64
hist_player_RATING_20_player_wise	float64

dtype: object

### 3.3.4 Feature Selection

After immense efforts to capture the nuances of the scrabble game at levels of board usage, goodness of play through turns and lastly historic player level performance, we arrive at these sets of features as a whole. To avoid the curse of dimensionality, we performed feature selection on top of all the engineered features and got the data ready for predictive modeling.

One hot encoding the category variables:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Drop 'game_id'
df = df.drop('game_id', axis=1)

# Set binary column based on 'first' values
df['first_bot'] = df['first'].isin(["HastyBot", "BetterBot", "STEEBot"]).astype(int)
df = df.drop('first', axis=1)

# One-hot encode categorical columns
categorical_columns = ['time_control_name', 'game_end_reason', 'lexicon', 'rating_mode', 'Bot_nickname']
df = pd.get_dummies(df, columns=categorical_columns)

# Drop 'Player_nickname' and 'created_at'
df = df.drop(['Player_nickname', 'created_at', "Bot_game"], axis=1)

# Move 'Player_rating' to the last column
cols = [col for col in df.columns if col != 'Player_rating'] + ['Player_rating']
df = df[cols]

df
```

The following code utilizes a RandomForestRegressor from the scikit-learn library to predict the 'Player\_rating' target variable based on a set of features. The features include various aspects related to the gameplay, such as time control settings, turn counts, word lengths, rack utilization rates, and other relevant parameters. The RandomForestRegressor is trained on the entire dataset (df), where 'X' represents the feature matrix (all columns except 'Player\_rating'), and 'y' represents the target variable ('Player\_rating'). The model is then fitted to the data, and the feature importances are calculated, reflecting the contribution of each feature to the prediction.

The code creates a DataFrame (feature\_importance\_df) to store the computed feature importances, with columns for feature names and their corresponding importance scores. The DataFrame is sorted in descending order based on feature importance, and the results are printed. Furthermore, the code identifies features with importances greater than 0.005, resulting in a subset of selected features (selected\_features). This subset is sorted by importance, and the resulting DataFrame is printed. The selected feature names are extracted, and a new DataFrame (df\_selected) is created by selecting columns corresponding to the chosen features and adding the 'Player\_rating' column.

```

from sklearn.ensemble import RandomForestRegressor
import numpy as np

# Assuming df is your DataFrame
X = df.drop('Player_rating', axis=1) # Features
y = df['Player_rating'] # Target variable

# Initialize the Random Forest Regressor
rf_model = RandomForestRegressor(random_state=42)

# Fit the model to the data
rf_model.fit(X, y)

# Get feature importances
feature_importances = rf_model.feature_importances_

# Create a DataFrame to store the results
feature_importance_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': feature_importances
})

# Sort the DataFrame by importance in descending order
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False)

# Print the sorted DataFrame
print(feature_importance_df)

```

```

selected_features = feature_importance_df[feature_importance_df['Importance'] > 0.005]

# Sort the DataFrame by importance in descending order
selected_features = selected_features.sort_values(by='Importance', ascending=False)

# Print the sorted DataFrame
selected_features

```

```

:

```

	Feature	Importance
24	hist_player_performance	0.271239
39	hist_player_score_player_wise	0.193395
37	Player_word_len_player_wise	0.126998
36	Player_avg_bingo_player_wise	0.074017
40	Player_Double_Word_player_wise	0.044229
33	hist_player_score_20	0.033611
44	hist_player_RATING_20_player_wise	0.031552
42	Player_Double_Letter_player_wise	0.029270
54	lexicon_CSW21	0.027913
41	Player_Triple_Word_player_wise	0.026939
43	Player_Triple_Letter_player_wise	0.024552
57	lexicon_NWL20	0.020995
23	Bot_rating	0.017229
20	Player_game	0.015400
0	initial_time_seconds	0.009590
59	rating_mode_RATED	0.008751
58	rating_mode_CASUAL	0.007115
3	game_duration_seconds	0.006413

## IV. Predictive Modeling

---

### 4.1 Modeling

Modeling serves as a cornerstone in our project, providing a systematic framework to understand, simulate, and make predictions within complex systems. In recognizing the diverse nature of our challenges, we have employed various modeling techniques.

In our modeling process, we are fine-tuning hyperparameters which affect the behavior of the models. This iterative optimization is crucial to enhancing the performance of our algorithms, ensuring they are calibrated for the underlying aspects of the data. Following hyper parameters tuning, the model is evaluated through Root Mean Squared Error (RMSE) to quantify the disparity between predicted and actual values. The objective lies in identifying the optimal set of hyperparameters that yield the lowest RMSE, indicating the model's ability to minimize predictions errors. This process ensures that our models are not only adept at capturing patterns within the data and providing accurate and reliable predictions on new data.

#### 4.1.1 Decision Tree

In this following code, a machine learning pipeline was constructed for predicting player ratings using a Decision Tree Regressor. The dataset, denoted as `df_selected`, is feature selected at earlier stages with the selected features stored in `X` and the corresponding player ratings in `y`. The pipeline incorporated a `StandardScaler` for feature scaling.

To optimize the model's performance, a grid search was conducted using `GridSearchCV`. The grid search involved exploring different hyperparameter combinations for the Decision Tree Regressor, specifically varying `max_depth`, `min_samples_split`, and `min_samples_leaf`. The performance metric used for evaluation was the Root Mean Squared Error (RMSE), configured through the creation of a custom scoring function.

The results of the grid search indicated the optimal hyperparameters for the Decision Tree Regressor. The best configuration was found to be with no specified maximum depth (`None`), a minimum number of samples per leaf set to 1, and a minimum number of samples required to split an internal node set to 2.

Subsequently, the best model, identified by the grid search, was printed along with its associated parameters. A 10-fold cross-validation was performed to assess the model's generalization performance. The RMSE was calculated for each fold, revealing the model's predictive accuracy across different subsets of the dataset.

```

from lightgbm import LGBMRegressor
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.metrics import make_scorer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
import numpy as np

from sklearn.tree import DecisionTreeRegressor

# Assuming df_selected is your DataFrame
X = df_selected.drop('Player_rating', axis=1)
y = df_selected['Player_rating']

# Create a pipeline with DecisionTreeRegressor
pipeline = Pipeline([
    ('scaler', StandardScaler()), # Decision trees don't require scaling, but keeping it for consistency
    ('dt', DecisionTreeRegressor())
])

# Define parameter grid for grid search
param_grid = {
    'dt__max_depth': [None, 6, 8, 12, 15],
    'dt__min_samples_split': [2, 5, 10],
    'dt__min_samples_leaf': [1, 2, 4, 8]
}

# Define RMSE as the scoring metric for GridSearchCV
rmse_scorer = make_scorer(lambda y_true, y_pred: np.sqrt(np.mean((y_true - y_pred)**2)))

# Initialize GridSearchCV with verbose set to a positive integer
grid_search = GridSearchCV(pipeline, param_grid, cv=10, scoring=rmse_scorer, n_jobs=-1, verbose=2)

# Fit the model using partial_fit to print live progress
grid_search.fit(X, y)

# Get the best parameters
best_params = grid_search.best_params_
print("Best Parameters:", best_params)

# Get the best model
best_model = grid_search.best_estimator_
print("\nBest Model:", best_model)

# Get cross-validation scores
print("\nRunning cross-validation. This may take some time...")
kf = KFold(n_splits=10, shuffle=True, random_state=42) # Specify the same number of splits as the GridSearchCV
fold = 1
for train_index, test_index in kf.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    best_model.fit(X_train, y_train)
    y_pred = best_model.predict(X_test)

    fold_rmse = np.sqrt(np.mean((y_test - y_pred)**2))
    print(f"Fold {fold}: RMSE = {fold_rmse}")
    fold += 1

```

```

[CV] END dt__max_depth=8, dt__min_samples_leaf=4, dt__min_samples_split=10; total time= 0.6s
[CV] END dt__max_depth=8, dt__min_samples_leaf=8, dt__min_samples_split=2; total time= 0.6s
[CV] END dt__max_depth=8, dt__min_samples_leaf=8, dt__min_samples_split=2; total time= 0.6s
[CV] END dt__max_depth=8, dt__min_samples_leaf=8, dt__min_samples_split=5; total time= 0.6s
Best Parameters: {'dt__max_depth': None, 'dt__min_samples_leaf': 1, 'dt__min_samples_split': 2}

```

Best Model: Pipeline(steps=[('scaler', StandardScaler()), ('dt', DecisionTreeRegressor())])

Running cross-validation. This may take some time...

```

Fold 1: RMSE = 43.40866003530087
Fold 2: RMSE = 47.60120040977224
Fold 3: RMSE = 42.82020145111497
Fold 4: RMSE = 39.76215263222003
Fold 5: RMSE = 43.34167117933072
Fold 6: RMSE = 44.96476690200695
Fold 7: RMSE = 41.768119080632005
Fold 8: RMSE = 45.28085496993098
Fold 9: RMSE = 46.09177654961641
Fold 10: RMSE = 45.87825848445098

```

Predictions were generated to check the model performance in Kaggle out of sample test data:

```
from datetime import datetime

test_df_selected = test[selected_feature_names + ['Player_rating']]

# Display the selected DataFrame
test_df_selected.head()
# Assuming df_selected is your DataFrame
test_X = test_df_selected.drop('Player_rating', axis=1)
test_y = test_df_selected['Player_rating']
test_X["Predictions"] = best_model.predict(test_X)
timestamp = datetime.now().strftime("%Y%m%d%H%M%S")
file_name = f"final_test_results_{timestamp}.csv"

final_test_results = test_X[["game_id", "Predictions"]]
final_test_results.rename(columns={"Predictions": "rating"}, inplace=True)
final_test_results_reset = final_test_results.reset_index(drop=True)
final_test_results_sorted = final_test_results_reset.sort_values(by='game_id', ascending=True)
final_test_results_sorted.to_csv(file_name, index=False)
```

OOS Performance in Kaggle:



final\_test\_results\_20231201222043.csv

Complete (after deadline) · 33s ago · DT

160.77265

159.75138



## 4.1.2 Neural Networks

Neural networks mimic the basic functioning of the human brain and are inspired by how the human brain interprets information. Thus, we tried to use our bunch of features to interconnect nodes in multiple layered structures, making computers learning and improving. First of all, we create the model with a predefined hidden layer = 100, activation function of relu, and adam optimizer loss function. And then adding multiple layers and 0.2 drop out rate is the process of building neural networks, and setting mean square error as metrics.

```
# Create a pipeline with a KerasRegressor
def create_model(nb_hidden=100, activation='relu', optimizer='adam'):
    model = Sequential()
    model.add(Dense(nb_hidden, input_dim=X.shape[1], activation=activation))
    model.add(Dropout(rate=0.2))
    model.add(Dense(nb_hidden, activation=activation))
    model.add(Dropout(rate=0.2))
    model.add(Dense(nb_hidden, activation=activation))
    model.add(Dropout(rate=0.2))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer=optimizer, metrics=['mean_squared_error'])
    return model

# Create a pipeline with a KerasRegressor
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', KerasRegressor(build_fn=create_model, nb_hidden=50, verbose=0))
])
```

Secondly, we used model checkpoint and early stopping to monitor the performance of the model on training and validation datasets, making sure that once the validation dataset starts to degrade, training can stop. Moreover, we also used different batch sizes, from 8 to 256, and epochs equal 12 and 32 as the hyperparameters to tune the neural network models.

```

# callbacks
checkpoint = ModelCheckpoint('Models/best_model2.h5', monitor='val_loss', verbose=1, save_best_only=True)
early_stopping = EarlyStopping(patience=500)

# Define parameter grid for grid search
param_grid = {
    'model__nb_hidden': [50, 100, 200],
    'model__batch_size': [8, 32, 64, 256],
    'model__epochs': [12, 32],
    'model__callbacks': [[checkpoint, early_stopping]]
}

# Define RMSE as the scoring metric for GridSearchCV
rmse_scorer = make_scorer(lambda y_true, y_pred: np.sqrt(np.mean((y_true - y_pred)**2)))

```

The following is the best model and hyperparameters settings. We used the best model to do cross validation and get the RMSE scores of different folds.

```

best_model
Pipeline
  StandardScaler
  StandardScaler()
  KerasRegressor
    loss=None
    metrics=None
    batch_size=256
    validation_batch_size=None
    verbose=0
    callbacks=[<keras.src.callbacks.ModelCheckpoint object at 0x7cc37a225510>, <keras.src.callbacks.EarlyStopping object at 0x7cc37a227b80>]
    validation_split=0.0
    shuffle=True
    run_eagerly=False
    epochs=12
    nb_hidden=50

```

The RMSE ranges from 117 to 134 and we can see that the validation RMSE is not stable and does not perform relatively well for 10 cross validation folds.

```

Fold 1: RMSE = 119.35602004977798
Fold 2: RMSE = 117.65337941579384
Fold 3: RMSE = 134.5028595569429
Fold 4: RMSE = 123.66606505566963
Fold 5: RMSE = 132.31885938642765
Fold 6: RMSE = 122.35657470638739
Fold 7: RMSE = 119.55547602757717
Fold 8: RMSE = 129.18043279876483
Fold 9: RMSE = 123.83112980816988
Fold 10: RMSE = 121.25506349589665

```

The Kaggle final score ended out to be 132.7.

scrabble\_nn\_pca.csv  
Complete (after deadline) · 1d ago

132.70999

131.96994



### 4.1.3 XGBoost

Two main advantages of XGBoost are execution speed and model performance. When using XGBoost, there's no restriction on the dataset size. Also, XGBoost outperforms other algorithms such as gradient boosting machines. Hence, building XGBoost models is one of our choices. Firstly, we built the pipeline for normalization and XGBoost regressor, and then we set hyperparameters of n\_estimators, max\_depth, and learning rate.



```
# Create a pipeline with an XGBRegressor
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('xgb', XGBRegressor())
])

# Define parameter grid for grid search
param_grid = {
    'xgb__n_estimators': [50, 100, 200],
    'xgb__max_depth': [3, 6, 9, 15, 20],
    'xgb__learning_rate': [0.05, 0.1, 0.2]
}

# Define RMSE as the scoring metric for GridSearchCV
rmse_scorer = make_scorer(lambda y_true, y_pred: np.sqrt(np.mean((y_true - y_pred)**2)))
```

Secondly, we used GridSearchCV to find optimal parameter values from a given set of parameters in a grid, and then fit the model, printing out the best parameters and best model.

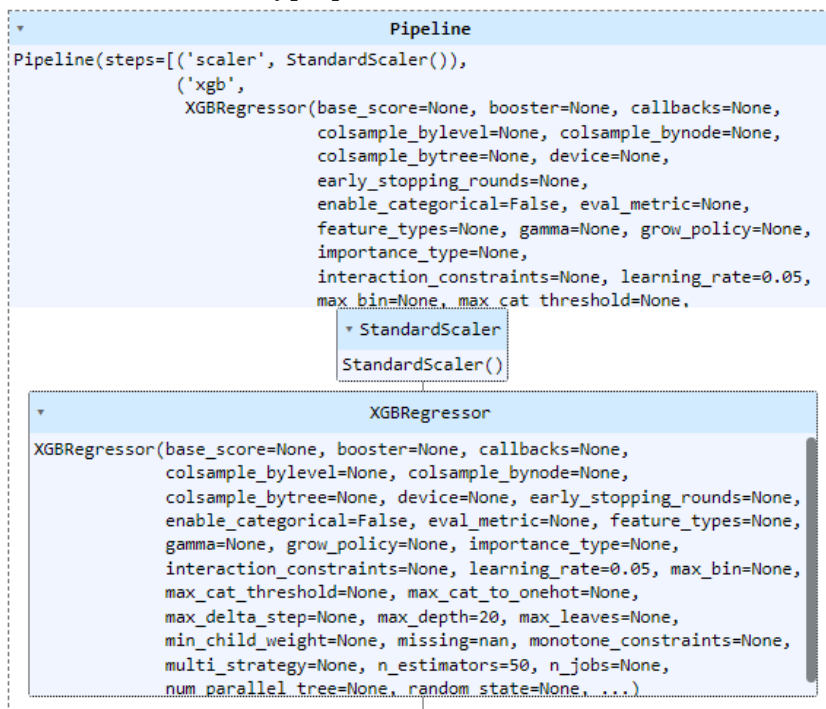
```
# Initialize GridSearchCV
grid_search = GridSearchCV(pipeline, param_grid, cv=10, scoring=rmse_scorer, n_jobs=-1)

# Fit the model
grid_search.fit(train_X, train_y)

# Get the best parameters
best_params_xgb = grid_search.best_params_
print("Best Parameters:", best_params_xgb)

# Get the best model
best_model_xgb = grid_search.best_estimator_
print("Best Model:", best_model_xgb)
```

According to the result, the best hyperparameters are 0.5 learning rate, 50 estimators, and max\_depth equals 20. Thus, we use this hyperparameters combination to do cross-validation.




Finally we do a cross validation for 10 folds, and the following RMSE performance ranges from 44 to 48, which might be a relatively great performance.

```
Running cross-validation. This may take some time...
Fold 1: RMSE = 44.36048877458316
Fold 2: RMSE = 46.217684698351576
Fold 3: RMSE = 46.81593390290792
Fold 4: RMSE = 43.01745416989696
Fold 5: RMSE = 48.008757078552506
Fold 6: RMSE = 48.09294794191704
Fold 7: RMSE = 44.072119431703676
Fold 8: RMSE = 45.69254629772113
Fold 9: RMSE = 47.06493971077543
Fold 10: RMSE = 45.588839606489664
```



The Kaggle final score ended up to be 120.57.

Submission and Description		Private Score ⓘ	Public Score ⓘ	Selected
 <b>final_result_xgb_all.csv</b> Complete (after deadline) · now		<b>120.57152</b>	<b>121.85564</b>	<input type="checkbox"/>

#### 4.1.4 LightGBM

In the following code, a LightGBM regression model is trained on a dataset (`df_selected`) to predict the 'Player\_rating' target variable. The model is built using a pipeline that includes a `StandardScaler` for feature scaling and the `LightGBMRegressor` as the underlying regressor. A grid search is conducted to find the best hyperparameters for the model, considering variations in the number of estimators, maximum depth, learning rate, minimum child samples, subsample, and `colsample_bytree`.

The grid search is performed using 10-fold cross-validation, and the root mean squared error (RMSE) is chosen as the evaluation metric. The cross-validation process involves fitting the model on different training sets and evaluating its performance on corresponding test sets. The printed output displays the RMSE for each fold, providing insights into the model's generalization performance across different subsets of the data.

```

from lightgbm import LGBMRegressor
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.metrics import make_scorer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
import numpy as np

# Assuming df_selected is your DataFrame
X = df_selected.drop('Player_rating', axis=1)
y = df_selected['Player_rating']

# Create a pipeline with LightGBM
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('lgbm', LGBMRegressor())
])

# Define parameter grid for grid search
param_grid = {
    'lgbm__n_estimators': [50, 100, 200],
    'lgbm__max_depth': [6, 8, 12, 15],
    'lgbm__learning_rate': [0.01, 0.1, 0.2],
    'lgbm__min_child_samples': [5, 10, 20],
    'lgbm__subsample': [0.8, 0.9, 1.0],
    'lgbm__colsample_bytree': [0.8, 0.9, 1.0]
}

# Define RMSE as the scoring metric for GridSearchCV
rmse_scorer = make_scorer(lambda y_true, y_pred: np.sqrt(np.mean((y_true - y_pred)**2)))

# Initialize GridSearchCV with verbose set to a positive integer
grid_search = GridSearchCV(pipeline, param_grid, cv=10, scoring=rmse_scorer, n_jobs=-1, verbose=2)

# Fit the model using partial_fit to print live progress
grid_search.fit(X, y)

# Get the best parameters
best_params = grid_search.best_params_
print("Best Parameters:", best_params)

# Get the best model
best_model = grid_search.best_estimator_
print("\nBest Model:", best_model)

# Get cross-validation scores
print("\nRunning cross-validation. This may take some time...")
kf = KFold(n_splits=10, shuffle=True, random_state=42) # Specify the same number of splits as the GridSearchCV
fold = 1
for train_index, test_index in kf.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    best_model.fit(X_train, y_train)
    y_pred = best_model.predict(X_test)

    fold_rmse = np.sqrt(np.mean((y_test - y_pred)**2))
    print(f"Fold {fold}: RMSE = {fold_rmse}")
    fold += 1

```

Running cross-validation. This may take some time...

Fold 1: RMSE = 32.175901007395396  
 Fold 2: RMSE = 35.50796838935834  
 Fold 3: RMSE = 31.45438952471234  
 Fold 4: RMSE = 29.909631556362626  
 Fold 5: RMSE = 37.958366785320784  
 Fold 6: RMSE = 38.068662100878896  
 Fold 7: RMSE = 31.865194330694482  
 Fold 8: RMSE = 33.75604439479598  
 Fold 9: RMSE = 34.985449558879665  
 Fold 10: RMSE = 34.15248055179436



final\_test\_results\_20231201212019.csv  
Complete (after deadline) · 1h ago · light GBM

152.58151

152.535



## 4.1.5 Stacking

Stacking is a powerful ensemble learning strategy that can increase the predictive performance of machine learning models dramatically. Also, it can reduce bias and variation, increase model variety, and improve the

interpretability of the final forecast by merging the predictions of many base models. We combined random forest, lightgbm, and XGboost and used random forest as the final estimator. Moreover, we also fine tune each model's hyperparameters, such as max\_depth, learning rate, and estimator. Finally, we use GridSearchCV to optimize the parameters values.

```
# ridge = Ridge(random_state=1)
# lasso = Lasso(random_state=1)
rf = RandomForestRegressor(n_estimators=10, random_state=1)
lgb = lgb.LGBMRegressor(n_estimators=10, random_state=1)
xgb = XGBRegressor(n_estimators=10, random_state=1)

regressors = [('rf', rf), ('lgb', lgb), ('xgb', xgb)]

# Specify the StackingRegressor with final_estimator
stregr = StackingRegressor(estimators=regressors, final_estimator=RandomForestRegressor(random_state=1))

# Define the parameter grid with correct parameter names
params = {
    'rf__max_depth': [3, 10, 15],
    'xgb__max_depth': [3, 9, 15],
    'xgb__learning_rate': [0.05, 0.1, 0.2],
    'final_estimator__n_estimators': [100, 200, 500],
    'final_estimator__max_depth': [3, 6, 10, 15], # Adjust this based on your final estimator
}

# Initialize GridSearchCV
grid = GridSearchCV(estimator=stregr, param_grid=params, cv=5, refit=True)
grid.fit(X_train_stacking, y_train_stacking)
```

We do cross-validation for 10 folds, and the following RMSE performance ranges from 95 to 101, which might be a relatively high error. We deduced that there might be a lot of overlap in the correct predictions of the ensemble models, causing validation performance not well.

```
[19]: from sklearn.model_selection import GridSearchCV, KFold
# Perform cross-validation with the best stacking model
print("\nRunning cross-validation. This may take some time...")
kf = KFold(n_splits=10, shuffle=True, random_state=42)

fold = 1
for train_index, test_index in kf.split(X_train_stacking):
    X_train, X_test = X_train_stacking.iloc[train_index], X_train_stacking.iloc[test_index]
    y_train, y_test = y_train_stacking.iloc[train_index], y_train_stacking.iloc[test_index]

    best_stacking_model = grid.best_estimator_
    best_stacking_model.fit(X_train, y_train)
    y_pred = best_stacking_model.predict(X_test)

    fold_rmse = np.sqrt(np.mean((y_test - y_pred)**2))
    print(f"Fold {fold}: RMSE = {fold_rmse}")
    fold += 1

Running cross-validation. This may take some time...
Fold 1: RMSE = 95.09473822882399
Fold 2: RMSE = 95.25421251448775
Fold 3: RMSE = 96.71211005687181
Fold 4: RMSE = 98.35368505723501
Fold 5: RMSE = 101.74366078009125
Fold 6: RMSE = 102.87996835415295
Fold 7: RMSE = 95.32937409079312
Fold 8: RMSE = 99.9830024465
Fold 9: RMSE = 98.15863062429969
Fold 10: RMSE = 96.38146947471886
```

The Kaggle final score ended up to be 119.32



ensemble\_rf\_xgb.csv  
Complete (after deadline) · now

119.32834

120.00692



## 4.2 Final Model Selection

We have used various models and chose the model which yielded the lowest RMSE score. Out of all the models, Random Forest had an RMSE of 105.82 which signifies its ability to minimize the predictions errors. Random Forest ensemble model boosts accuracy by spreading out across different features and observations, reducing errors.

The utilization of a Random Forest ensemble model presents advantages to effectively diversify across

features and observations. This diversified approach enhances model stability and robustness, as the aggregate result of multiple models tends to be less noisy than individual counterparts. Despite these advantages, certain drawbacks warrant consideration. Notably, the interpretability of the model is compromised, making it harder to derive the relationship between different features directly and deriving insights.

```
In [37]: # Define the hyperparameter grid for GridSearchCV
param_grid = {
    'max_depth': [2, 6, 8, 10, 12, 16],
    'n_estimators': [100, 200, 300, 400],
    'max_features': ['sqrt', 'log2'],
    'bootstrap': [False]
}

forest = RandomForestRegressor(random_state=42)

grid_search = GridSearchCV(forest, param_grid, scoring='neg_mean_squared_error', cv=3)

grid_search.fit(X_train, y_train)

# Access the best parameters and best estimator
best_params = grid_search.best_params_
best_estimator = grid_search.best_estimator_

# Evaluate on the validation set
validation_score = grid_search.score(X_test, y_test)

# Print or use the results as needed
#print("Best Parameters: ", best_params)
print("Best Negative Mean Squared Error: ", -grid_search.best_score_)
print("Validation Score: ", -validation_score)
```

OOS Performance on Kaggle :



MSBA6421\_FALL\_2023\_Section1\_Scrabble\_team\_Nick.csv

Complete (after deadline) · 22s ago · Final Submission

105.82038

108.07761



# V. Business Value & Future Improvements

---

## 5.1 Summary & Recommendation

Several real-world business applications exist for the model used here, or at least the techniques used to solve this problem. Our most direct example is the obvious crossover into sports, and in particular, sports betting and gambling. With analytics being at the forefront of most sports, we can collect lots of data and information not only about the games themselves, but also about individual player statistics.

A modification to the model to determine the probability of a win could allow a better to directly bet on the money line of games. A few similar modifications could be used to determine the appropriate spread of a game, or even the total points scored in a game. These 3 bets are by far the most common ways to bet on sports events for sports like basketball. While an individual can choose to do this for themselves, there are also many large sports books who are looking to set good lines and maintain their house advantage. Risk management for these casinos employ machine learning models to set competitive lines and improve profitability.

Franchises and teams have long employed data scientists to help make roster improvements and build the best teams they can – or at least at the prices they can afford. Our task for this model only involved data for two months. However, if enough data was collected over a longer time horizon, we believe it to be possible for a model like this to help predict the future abilities of a player in a game. This is likely the most valuable aspect for a franchise, as signing young and talented players is often the cheapest way to be successful.

Customer habits and behavior prediction could also be modeled with the techniques we have used here. Several aspects of consumer loans garner lots of attention and labor from banks: credit/default risk, and prepayment risk. These are both behaviors which can be, and are, modeled extensively at banks and investment shops of all sizes.

## 5.2 Future Improvements

In conclusion, the path forward for enhancing our model involves embracing the power of big data software, enabling real-time predictions as new data continuously streams in—an approach reminiscent of successful game rating systems. Embracing this paradigm shift would bring our model on par with industry standards. Additionally, we are keen on expanding our model exploration beyond Random Forest's success, with a particular focus on Neural Networks known for their high performance. Continuous experimentation with different models holds the promise of uncovering more effective approaches. Furthermore, the journey doesn't end with the 28 features we've developed; we aim to delve deeper into creating additional features derived from historical data, maximizing our model's predictive capabilities. This multi-faceted approach represents our commitment to staying at the forefront of predictive modeling, ensuring our system's ongoing evolution and improvement in predicting player ratings.



UNIVERSITY OF MINNESOTA  
**Driven to Discover<sup>SM</sup>**