Predicting Scrabble Players Rating

Chun-Wei Pan, Alan Chen, Changhyuk (Andy) Byun,
Ssu Hsien Lee, Tsu Jung Liu

Presentation Video
https://www.youtube.com/watch?v=TlIfPrIvOrg

MSBA 6421
Dr. Yicheng Song

Table of Contents

Part 1: Executive Summary

Project Introduction

We aimed to build a predictive model to predict a Scrabble player's rating before playing the game. Scrabble is a two-person crossword board game where players place letter tiles on the board to create words. Players earn points based on several factors, such as the board's location and the word's length.

We used gameplay data from Woogles.io, an online version of Scrabble, to train and test to build the best-performing model. The model performance was evaluated using root mean squared error (RMSE), which captures the average difference between predicted and ground truth ratings.

Methodology

To build a model that minimizes the RMSE, we adopted the following methods:

1. Feature Engineering
    a. We experimented with new and original features to find the features that impact prediction.
2. Model Comparison
    a. We tested five algorithms - Linear Regression, K-Nearest Neighbors (KNN), Decision Trees, Random Forest, and XGBoost - and chose the best-performing model.

Model Evaluation

The training performance of the four models mentioned above is as follows:

	RMSE
Linear Regression	114.543
KNN	82.138
Random Forest	**50.667**
XGBoost	**52.774**

Random Forest and XGBoost were the two best-performing models, with a difference of only about 2. Therefore, we decided to develop these two models further.

Best Model

Our final model was **XGBoos**t with 150 estimators, a 0.0968 learning rate, and a max depth of 4. Detailed hyperparameter settings and final features can be found in Part 4: Final Result.

Part 2: Technical Specification

Tools	Version
JUPYTER NOTEBOOK	7.0.6
Scikit Learn	1.3.2
Numpy	1.23.4
Pandas	1.5.3
MatplotLib	3.7.3
XGBoost	2.0.2

Part 3: Analysis

3-1. Data Preparation

Loading training & testing data
All data used for the analysis have been sourced from the Kaggle competition and are provided in CSV format. The dataset comprises five files, focusing on two primary files: games.csv and turns.csv, which encapsulate most of the information essential for our subsequent analysis.

Data files overview

1. ***games.csv*** - It contains metadata about each game, including a unique game ID, information on which player took the first turn, and the game's duration, among other details.
2. ***turns.csv*** - It encompasses detailed information about every turn in each game, such as the points scored, the current rack configuration, and the move made during each turn.
3. ***train.csv*** - It contains final scores and ratings for each player in each game, with player ratings recorded each player, are as of before the game was played
4. ***test.csv*** - It also contains final scores and ratings for each player in each game. Notably, the datasets contain null values for player ratings, and predicting these missing values is a key task in our analysis.
5. ***sample_submission.csv*** - It serves as a reference for the correct format when submitting predictions. It aids participants in aligning our submissions with the competition requirements.

Data Loading Process

To commence the analysis, we employed the Pandas library to upload and read the datasets into Pandas DataFrames. This enabled efficient handling and manipulation of the data for subsequent tasks. The process resulted in two main DataFrames: one for game metadata (games) and another for turn-level details (turns).

The summary statistics of the loaded data are as follows:

1. *games:* 12 columns, 72,772 rows
2. *turns:* 9 columns, 2,005,497 rows
3. *test:* 4 columns, 44,726 rows
4. *train*: 4 columns, 100,820 rows

```
games = pd.read_csv("games.csv")
sample_submission = pd.read_csv("sample_submission.csv")
test = pd.read_csv("test.csv")
train = pd.read_csv("train.csv")
turns = pd.read_csv("turns.csv")
```

While the game's data frame didn't contain any null values, it's worth noting that there are some missing data in the turns data. Addressing these null values may be a consideration for later stages in our analysis.

```
games.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 72773 entries, 0 to 72772
Data columns (total 12 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   game_id                72773 non-null  int64
 1   first                  72773 non-null  object
 2   time_control_name      72773 non-null  object
 3   game_end_reason        72773 non-null  object
 4   winner                 72773 non-null  int64
 5   created_at             72773 non-null  object
 6   lexicon                72773 non-null  object
 7   initial_time_seconds   72773 non-null  int64
 8   increment_seconds      72773 non-null  int64
 9   rating_mode            72773 non-null  object
 10  max_overtime_minutes   72773 non-null  int64
 11  game_duration_seconds  72773 non-null  float64
dtypes: float64(1), int64(5), object(6)
memory usage: 6.7+ MB
```

```
turns.isna().sum()
```

```
game_id             0
turn_number         0
nickname            0
rack            69390
location       132239
move              454
points              0
score               0
turn_type         395
dtype: int64
```

Merging Data

To predict each player's rating, our initial step involved merging the train data encompassing each player's rating with the game's data, utilizing the common game ID as the merging key. This process allowed us to consolidate relevant information and explore potential features influencing the ratings.

```
train_merge1 = train.set_index("game_id")
games_merge1 = games.set_index("game_id")
merge1 = train_merge1.join(games_merge1)
merge1.head()
```

Introduce Basic Features

Following the data merging process, our attention shifted to feature exploration. In this phase, besides the original features provided by the data set, such as *"max_overtime_minutes," "initial_time_seconds," and " win_or_not,"* we initiated the creation of basic features that we believed could have an impact on the players' ratings. These basic features were the building blocks for our subsequent analysis, laying the groundwork for a comprehensive understanding of the factors influencing player ratings.

1. **Mean points** - There is often a correlation between a player's mean points and their overall rating. Players consistently performing well in terms of points are likely to have higher ratings. Understanding this relationship is crucial for building effective predictive models. Therefore, We grouped by the *"game id"* and the "*nickname"* in the game to calculate the mean points for each player in each game.

```
mean_pts = turns.groupby(["game_id", "nickname"]).agg({'points': 'mean'})
mean_pts.rename(columns={"points":"mean_pts"},inplace=True)
mean_pts
```

2. **Final score** - Besides mean points, we computed another crucial feature for our analysis — the final score for each player in each game. This involved determining a player's final score in the most recent turn of a game.

```
last_score = turns.groupby(["game_id", "nickname"]).agg({'score': 'last'})
last_score.rename(columns={"score":"last_score"},inplace=True)
last_score
```

3. **Count of turns** - Expanding our repertoire of features, we computed the count of turns for each game by summing the total number of turns across all players. This feature provided an essential metric for understanding the overall gameplay dynamics in a given game.

```
count_turn = turns.groupby(["game_id", "nickname"]).agg({"turn_number": "count"})
count_turn.rename(columns={"turn_number":"count_turn"},inplace=True)
count_turn
```

Feature Integration

After generating the three features— '*mean points,' 'last score,' and 'count of turns'*—we reintegrated them into the combined dataset of training and games. Subsequently, we performed one-hot and label encoding to transform the data in the later progress, especially for the categorical features.

Meanwhile, by employing group-by and inner-join techniques, we have successfully handled null values in the train data, laying the groundwork for a more complete and robust dataset for model training and analysis. This meticulous approach enhanced the reliability and accuracy of the insights derived from the integrated dataset.

```
merge1_trans = merge1.reset_index()
merge1_trans = merge1_trans.set_index(["game_id","nickname"])
merge2 = merge1_trans.join(mean_pts)
merge3 = merge2.join(last_score)
merge4 = merge3.join(count_turn)
merge4 = merge4.reset_index()
merge4 = merge4.drop(["game_id","nickname","first", "time_control_name","created_at"],axis=1) #drop first
merge4.head()
```

```
has_null = merge4.isnull().values.any()
has_null_columns = merge4.isnull().any()
has_null_rows = merge4.isnull().any(axis=1)
print(has_null)
print()
print(has_null_columns)
```

```
False

score                              False
rating                             False
initial_time_seconds               False
increment_seconds                  False
max_overtime_minutes               False
game_duration_seconds              False
mean_pts                           False
last_score                         False
count_turn                         False
game_end_reason_CONSECUTIVE_ZEROES False
game_end_reason_RESIGNED           False
game_end_reason_STANDARD           False
game_end_reason_TIME               False
winner_-1                          False
winner_0                           False
winner_1                           False
lexicon_CSW21                      False
lexicon_ECWL                       False
lexicon_NSWL20                     False
lexicon_NWL20                      False
rating_mode_CASUAL                 False
rating_mode_RATED                  False
dtype: bool
```

The dataset was ready for model training after completing the basic feature engineering and data preprocessing steps. The meticulous process of merging, creating, and encoding features provided a preliminary dataset that encapsulates essential information for predicting player ratings.

Feature Engineering

While basic features such as turn counts, final scores for each game, and mean points for each turn provided essential insights, integrating more complex features was important to enhance predictions and delve deeper into the game's dynamics. In this part, we explored how players assemble letters on the board and consider strategic tile placements and historical game performances, as these factors greatly influence the points gained and strategies applied in the game.

1. **Length of Moves**
   Longer moves not only indicate that the player is getting more points but also highlight a player’s adaptability and flexibility. Experienced players often have more knowledge in generating different word combinations to maximize their points. We removed spaces and dots from the string that records the moves and calculated its length.

```
import re
# Len of moves (remove dots)
turns['move_clean'] = turns['move'].astype(str).apply(lambda x: re.sub(r'[^a-zA-Z]', '', x))
turns['move_clean'] = turns['move_clean'].replace('.','')
turns['move_len'] = turns['move_clean'].apply(len)
```

2. **Difficulty of The Letters**
   Each letter in the tiles carries a distinct point value. Leveraging various letters, especially the more challenging ones, allows us to assess a player's proficiency in crafting complex words. We categorized letters into three levels - Difficult, Medium, and Easy - and tally the frequency of usage within each category.

```
# difficulty letters
difficult_letters = ["K", "J", "X", "Q", "Z"]
medium_letters = ["B", "C", "M", "P", "F", "H", "V", "W", "Y"]
easy_letters = ["A", "E", "I", "L", "N", "O", "R", "S", "T", "U", "D", "G"]

turns["difficult_letters"] = turns["move_clean"].apply(lambda x: len([letter for letter in x if letter in difficult_letters]))
turns["medium_letters"] = turns["move_clean"].apply(lambda x: len([letter for letter in x if letter in medium_letters]))
turns["easy_letters"] = turns["move_clean"].apply(lambda x: len([letter for letter in x if letter in easy_letters]))
```

3. **Blank Tiles Used**
   Each match includes two blank tiles that can represent any letter. Players earn 0 points when using blank tiles, irrespective of the letter chosen. Due to the absence of points for these tiles, players may strategize, contemplating their usage or intentionally holding some words for subsequent turns. The dataset records blank tiles in lowercase, so we identified them by lowercase representation.

```
# blank tiles = 0 points
turns["blank_used"] = turns["move_clean"].apply(lambda x: sum(1 for letter in x if letter.islower()))
```

4. **Bingo**
   A "Bingo" occurs when a player uses all seven tiles from their rack in a single turn. This not only earns the player a significant 50-point bonus but also showcases their exceptional vocabulary knowledge and strategic acumen. Creating lengthier words requires adjusting the letters available on the rack and board layout. To identify a "Bingo," we calculated the move's length, assigning a value of 1 for a “Bingo” move and 0 for other turns.

```
# bingo = extra 50 points
turns["is_bingo"] = turns["move_len"].apply(lambda x: 1 if x==7 else 0)
```

5. **Location Bonus**
   Different locations offer distinct bonuses like triple word or double letter scores, influencing the player’s decision on each move. We defined a function to evaluate these locations based on predefined bonus lists.

```python
def location_bonus(location):
    bonus = 0
    if location in triple_word_score_lo:
        bonus = 4
    elif location in double_word_socre_lo:
        bonus = 3
    elif location in triple_letter_score_lo:
        bonus = 2
    elif location in double_letter_score_lo:
        bonus = 1
    return bonus

triple_word_score_lo = ['1A','3H','1O']
double_word_socre_lo = ['2B','3C','4D','5E','8H','5K','4L','3M','2O','14B','13C','12D','11E','11K','12L','13M','14N']
triple_letter_score_lo = ['2F','2J','6B','6F','6J','6N','10B','10F','10J','10N','14F','14J']
double_letter_score_lo = ['1D','1L','3G','3I','4A','4H','4O','7C','7G','7I','7M',\
                          '8D','8L','9C','9G','9I','9M','12A','12H','12O','13G','13I',\
                          '15D','15L']

turns['bonus'] = turns["location"].apply(location_bonus)
```

6. **Game Level**
One player in each game competes against one of the three bots: BetterBot (beginner), STEEBot (intermediate), and HastyBot (advanced). The varying bot levels reflect the players' skill and impact their game strategy.

```python
conditions = [
    (df['nickname'] == "BetterBot") | (df['first'] == "BetterBot"),
    (df['nickname'] == "STEEBot") | (df['first'] == "STEEBot"),
    (df['nickname'] == "HastyBot") | (df['first'] == "HastyBot")
]

choices = [1, 2, 3]

df['game_level'] = np.select(conditions, choices, default=0)
```

7. **Win-Loss Rate**
This feature provides insights into a player's consistency and strategic skills. We calculated the cumulative count of games won and the total games played for each player. After that, we computed the ratio of wins to total games to assess a player's performance and consistency across time.

```python
merge2['cumulative_wins'] = merge2.groupby('nickname')['win_or_not'].cumsum()
merge2['cumulative_games'] = merge2.groupby('nickname').cumcount() + 1
merge2['win_loss_rate'] = merge2['cumulative_wins'] / merge2['cumulative_games']
```

8. **Last 10 Games Mean Score**
While the Win-Loss rate considers all historical games, we believe players can improve their skills over time. Therefore, focusing on the performance from the last 10 games might be more representative of recent gameplay trends. We calculated the mean score from these last 10 games using a rolling window to get deeper insights into a player's evolving strategy and adaptability.

```python
merge2['mean_past_10_games_score'] = merge2.groupby('nickname')['score'].rolling(window=10).mean().reset_index(level=0, drop=True)
merge2['mean_past_10_games_score'] = merge2['mean_past_10_games_score'].shift(1)
```

9. **Encoding**

Before integrating new features into the model, it was crucial to format the features for machine learning algorithms. Label encoding, our selected technique among various encoding methods, converts categorical data by assigning a unique numerical label to each category in a variable. We defined functions - `game_difficulty`, `lex_difficulty`, `ratemode`- to assign distinct numerical values to specific categories.

For instance, we calculated the average rating for each lexicon category and established distinct numeric representations based on these average ratings. This custom labeling method ensured that the resulting numerical labels corresponded to the average ratings of the lexicon categories, allowing for a structured and graduated encoding scheme. We applied the same logic to the other two columns.

```
def lex_difficulty(lex):
    lex_level = 0
    if lex == 'ECWL':
        lex_level = 1
    elif lex == 'NSWL20':
        lex_level = 2
    elif lex == 'NWL20':
        lex_level = 3
    elif lex == 'CSW21':
        lex_level = 4
    return lex_level

train_data['lex_level'] = train_data['lexicon'].apply(lex_difficulty)
train_data = train_data.drop(['lexicon'],axis = 1)
train_data
```

```
def game_difficulty(bot):
    difficulty = 0
    if bot == 'HastyBot':
        difficulty = 3
    elif bot == 'STEEBot':
        difficulty = 2
    elif bot == 'BetterBot':
        difficulty = 1
    return difficulty

turns['bot_difficulty'] = turns['nickname'].apply(game_difficulty)
turns_difficulty = turns.groupby(['game_id']).agg({'bot_difficulty':'max'})
```

```
def ratemode(mode):
    mode = 0
    if mode == 'CASUAL':
        mode = 0
    elif mode == 'RATED':
        mode = 1
    return mode

train_data['rating_mode'] = train_data['rating_mode'].apply(ratemode)
train_data
```

10. Normalization

Normalization was necessary for certain models we have tried and advantageous for others. To ensure equitable contributions from each feature in distance calculations and prevent any single feature from dominating and biasing the results, we normalized the data to scale them between 0 and 1.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_normalized = scaler.fit_transform(X)
```

3-2. Model Evaluation

Evaluation Metric: Root Mean Square Error

To compare the model performances, we used Root Mean Square Error (RMSE) as the performance metric. Root Mean Square Error is employed when the target variable is numeric, as is the case with our scenario—rating. It calculates the square root of the mean of the squared differences between the predicted and actual values. Lower RMSE values indicate a closer alignment between predicted and actual values, indicating a better model performance.

Nested Cross Validation

To mitigate the risk of overfitting while simultaneously optimizing hyperparameters, we have used Bayesian optimization with nested cross-validation, specifically the BayesSearchCV technique. Nested cross-validation (Nested CV) involves two layers of cross-validation: the outer loop is used to assess the model's performance, while the inner loop is dedicated to hyperparameter tuning. This nested structure ensures a more robust evaluation, as it tests the model's ability to generalize to new data while optimizing its parameters.

```python
def evaluate_model(model, search_space, X, y, num_trials=NUM_TRIALS):
    non_nested_scores = np.zeros(num_trials)
    nested_scores = np.zeros(num_trials)
    best_params = []

    for i in range(num_trials):
        inner_cv = KFold(n_splits=4, shuffle=True, random_state=i)
        outer_cv = KFold(n_splits=4, shuffle=True, random_state=i)

        if search_space:  # Check if search_space is not empty
            clf = BayesSearchCV(estimator=model, search_spaces=search_space, cv=inner_cv, n_iter=30, scoring=rmse_scorer)
            clf.fit(X, y)
            non_nested_scores[i] = -clf.best_score_
            best_params.append(clf.best_params_)
        else:
            # For models with no parameters to tune, just fit the model directly
            clf = model
            clf.fit(X, y)
            predictions = clf.predict(X)
            non_nested_scores[i] = rmse(y, predictions)
            best_params.append({})

        # Nested CV with parameter optimization
        nested_score = cross_val_score(clf, X=X, y=y, cv=outer_cv, scoring=rmse_scorer)
        nested_scores[i] = -nested_score.mean()

    avg_non_nested_score = non_nested_scores.mean()
    avg_nested_score = nested_scores.mean()

    std_non_nested_score = non_nested_scores.std()
    std_nested_score = nested_scores.std()

    return avg_non_nested_score, std_non_nested_score, avg_nested_score, std_nested_score, best_params
```

Model Selection

Our approach to model selection involved starting with a diverse range of models and allowing their performance to guide our choice. We initially included Linear Regression, K-Nearest Neighbors (KNN), Decision Trees, Random Forest, and XGBoost. The objective was to compare these models based on their predictive capabilities.

Our primary criterion for evaluating model performance was the RMSE score. Based on this metric, Random Forest and XGBoost have emerged as the top performers. This made us decide to focus more on these two models.

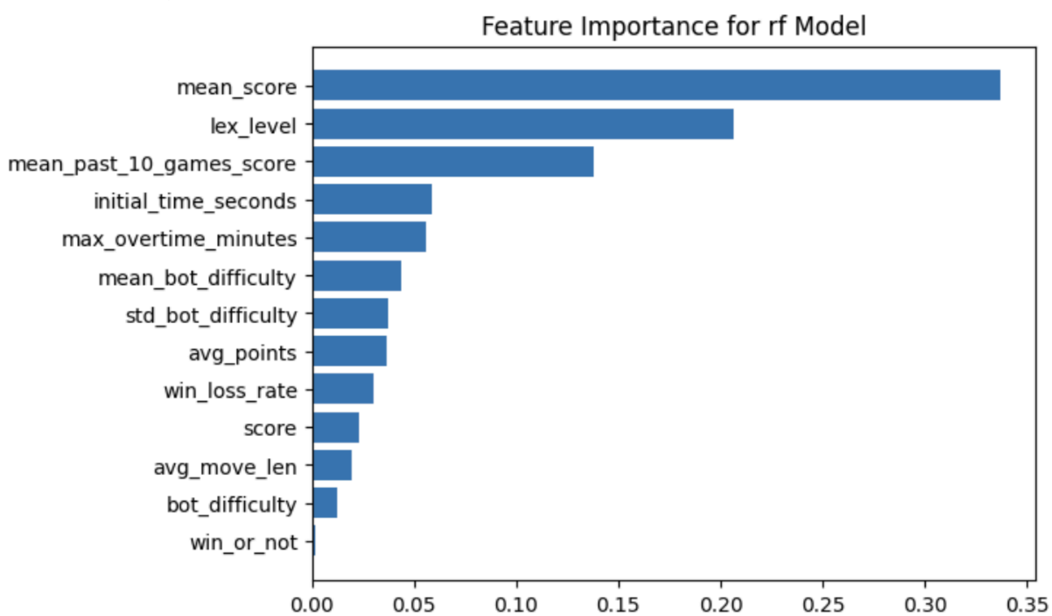
```
LinearRegression: Non-Nested CV Avg RMSE: 114.500, Std Dev: 0.000
LinearRegression: Nested CV Avg RMSE: 114.543, Std Dev: 0.005
Best parameters found: [{}, {}, {}]

KNeighborsRegressor: Non-Nested CV Avg RMSE: 82.140, Std Dev: 0.314
KNeighborsRegressor: Nested CV Avg RMSE: 82.138, Std Dev: 0.290
Best parameters found: [OrderedDict([('n_neighbors', 6), ('weights', 'distance')]), OrderedDict([('n_neighbors', 6), ('we

RandomForestRegressor: Non-Nested CV Avg RMSE: 50.410, Std Dev: 0.071
RandomForestRegressor: Nested CV Avg RMSE: 50.667, Std Dev: 0.145
Best parameters found: [OrderedDict([('max_depth', 30), ('max_features', 'sqrt'), ('min_samples_leaf', 1), ('min_samples_

XGBRegressor: Non-Nested CV Avg RMSE: 51.656, Std Dev: 0.043
XGBRegressor: Nested CV Avg RMSE: 52.774, Std Dev: 0.735
Best parameters found: [OrderedDict([('colsample_bytree', 0.9), ('learning_rate', 0.29999999999999993), ('max_depth', 5),
```

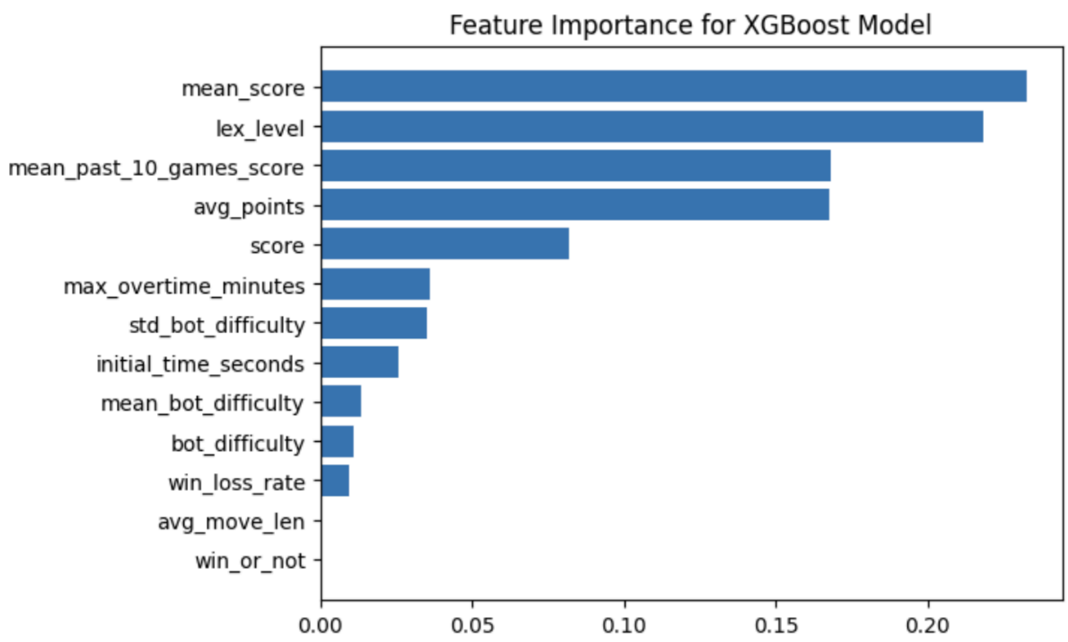
Feature Importance

Feature importance refers to the techniques used to identify and rank the relative importance of different features (or input variables) used in a model. This concept is crucial because not all features contribute equally to the predictions made by a model. Some features have a stronger influence on the outcome, while others may have little to no effect.

In our case, we continually assessed feature importance to identify elements that may be less critical to our predictions. By selectively dropping these less impactful features, we aimed to streamline our model, focusing on the most influential variables. This iterative process of examining and refining the feature set was instrumental in improving model performance, leading to more accurate and efficient predictive outcomes.

The following is the feature importance for two of our models:





3-3. Improvements

Historical Features

To represent players' performance more precisely, we introduced the historical features, the sum or average of the basic features from previous games, better representing their rating in the competition.
By using the *rolling()* function, along with *groupby()* function, we calculated the cumulative features, including player features like *'rolling_score_avg,' 'rolling_win_rate,'* or turn features like *'rolling_avg_length_of_move'*. Following is one of the examples.

```python
def cumm_player_features_bot(df, window_width):
    df= df.sort_values(by=["nickname", "created_at"])
    original_df_columns = df.columns

    for bot_name in df["bot_name"].unique():
        df[f'rolling_score_avg_{bot_name}'] = (df[df['bot_name'] == bot_name].groupby(['nickname'])['score'].rolling(window=window_width, min_periods=1)
               .sum().reset_index(level=0, drop=True) - df[df['bot_name'] == bot_name]['score']) / (df[df['bot_name'] == bot_name].groupby('nickname') \
                ['score'].rolling(window=window_width, min_periods=1).count().reset_index(level=0, drop=True) - 1)

        df[f'rolling_win_{bot_name}'] = (df[df['bot_name'] == bot_name].groupby('nickname')['win_or_not'].rolling(window=window_width, min_periods=1) \
                .sum().reset_index(level=0, drop=True) - df[df['bot_name'] == bot_name]['win_or_not'])

        df[f'rolling_win_rate_{bot_name}'] = df[df['bot_name'] == bot_name][f'rolling_win_{bot_name}'] / (df[df['bot_name'] == bot_name] \
                .groupby('nickname')['win_or_not'].rolling(window=window_width, min_periods=1).count().reset_index(level=0, drop=True) - 1)
        df[f'rolling_game_time_{bot_name}'] = (df[df['bot_name'] == bot_name].groupby('nickname')['game_duration_seconds']. \
                rolling(window=window_width, min_periods=1).sum().reset_index(level=0, drop=True) - df['game_duration_seconds']) \
                / (df[df['bot_name'] == bot_name].groupby('nickname')['game_duration_seconds'].rolling(window=window_width, min_periods=1) \
                   .count().reset_index(level=0, drop=True) - 1)

#  df[df.columns.difference(original_df_columns)] = df[df.columns.difference(original_df_columns)].fillna(0)

    df = df.sort_index()
    return df[df.columns.difference(original_df_columns)]
```

Remove Anomalies

From the exploration of the dataset, we found that some players' ratings always remain the same, which is 1500. The reason may be they only play 'CASUAL' games so their ratings never change. Thus, these records were excluded from our training to reduce the negative influence.

```
# get the only 1500 rating players and drop them
users_1500 = df[df["rating"] == 1500]["nickname"]
anomalous = df[df["nickname"].isin(users_1500)].groupby("nickname").\
    agg({'nickname':'count', 'rating' : lambda x : np.sum(x == 1500)})

anomalous["ratio"] = anomalous["rating"] / anomalous["nickname"]
anomalous_users = anomalous[(anomalous["ratio"] >= 1.0) & (anomalous["nickname"] > 1)].index
df = df[~df["nickname"].isin(anomalous_users)]
```

Separate Different Lexicon and Time_control_name

While reading and exploring the rules of Scrabble, we found that for different 'time_control_name' and 'lexicon', players will be rated by different rating systems. Thus, we tried aggregating the cumulative features by different combinations of 'time_control_name' and lexicons.

```
def cumm_player_features_lexicon_time_control_name(df, window_width):
    df= df.sort_values(by=["nickname", "created_at"])
    original_df_columns = df.columns

    for lexicon in all_lexicon:
        for mode in all_time_control_name:
            df[f'rolling_score_avg_{lexicon}_{mode}'] = (df[(df['lexicon'] == lexicon) & (df['time_control_name'] == mode)].groupby(['nickname'])['score'
            df[f'rolling_win_{lexicon}_{mode}'] = (df[(df['lexicon'] == lexicon) & (df['time_control_name'] == mode)].groupby('nickname')['win_or_not'].ro
            df[f'rolling_win_rate_{lexicon}_{mode}'] = df[(df['lexicon'] == lexicon) & (df['time_control_name'] == mode)][f'rolling_win_{lexicon}_{mode}'
            df[f'rolling_game_time_{lexicon}_{mode}'] = (df[(df['lexicon'] == lexicon) & (df['time_control_name'] == mode)].groupby('nickname')['game_dura
    # df[df.columns.difference(original_df_columns)] = df[df.columns.difference(original_df_columns)].fillna(0)

    df = df.sort_index()
    return df[df.columns.difference(original_df_columns)]
```

Ensemble Models

We also tried ensemble models, including XGBoost, LightGBM, and Random Forest, to mitigate the possible overfitting. However, it didn't help improve our RMSE.

```
from sklearn.ensemble import VotingRegressor
ensemble_model = VotingRegressor([('xgb_model_1', xgb_model), ('xgb_model_2', bayes_xgb),
                                  ('lgb_model', lgb_model)])
ensemble_model.fit(X, y)
predictions_ensemble = ensemble_model.predict(test_X)
```

Part 4: Final Result

4-1. Final Model and Features

The final model we used is XGBoost, with the following specifications:

Model	XGBoost
Features	1. Length of Moves 2. Difficulty of Letters 3. Blanks Tiles Used 4. Bingo 5. Location Bonus 6. Game Level 7. Win-Loss Rate 8. Last 10 Games Mean Score 9. Rolling Average Score 10. Rolling Win Rate 11. Rolling Average Length of Move

4-2. Optimized hyperparameter

By using Nested-cross-validation and manually testing on different models and hyperparameters, we found that the following hyperparameters selected for the XGBoost model gave us the best RMSE performance.

colsample_bytree	**0.7**
learning_rate	**0.0968**
max_depth	**4**
n_estimators	**150**
reg_alpha	**1**
reg_lambda	**1**
subsample	**0.9**

4-3. Results in Training dataset

```
{'Training': 4.218743604491624, 'Validation': 57.545816649126}
```

4-4. Final Results on Kaggle

xgboost_submission_8_window30.csv Complete (after deadline) · 2d ago	109.50917	111.07828