MLB Per-Season Home Run Projections

Ryan Dean

Project Aim & Targets

My main aim is to project a players total number of home runs in a season based entirely on all of their hitting stats from the season prior.

1. Prepare data effectively
2. Model same year data
3. Model staggered data

Introducing the Data

- Context
- Origin
- Types

```
1 df = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/cs356/baseball/Batting.csv')
2 print(df.shape) #wow
3 df.describe().round(2)
```

```
(113799, 24)
```

	yearID	stint	G	G_batting	AB	R	H	2B	3B	HR	...
count	113799.00	113799.00	113799.00	1615.00	113799.00	113799.00	113799.00	113799.00	113799.00	113799.00	...
mean	1969.63	1.08	50.38	31.42	137.42	18.24	35.84	6.14	1.21	2.87	...
std	40.45	0.29	46.74	48.66	182.99	27.89	51.87	9.60	2.55	6.41	...
min	1871.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	...
25%	1939.00	1.00	12.00	0.00	3.00	0.00	0.00	0.00	0.00	0.00	...
50%	1979.00	1.00	34.00	1.00	44.00	4.00	8.00	1.00	0.00	0.00	...
75%	2004.00	1.00	78.00	49.50	220.00	26.00	54.00	9.00	1.00	2.00	...
max	2023.00	5.00	165.00	162.00	716.00	198.00	262.00	67.00	36.00	73.00	...

8 rows × 21 columns

Key:

```
[ ]  1 features = {
     2     'yearID': 'Year',
     3     'playerID': 'Player',
     4     'G': 'Games',
     5     'AB': 'At Bats',
     6     'R': 'Runs',
     7     'H': 'Hits',
     8     '2B': 'Doubles',
     9     '3B': 'Triples',
    10     'HR': 'Homeruns',
    11     'RBI': 'Runs Batted In',
    12     'SB': 'Stolen Bases',
    13     'CS': 'Caught Stealing',
    14     'BB': 'Base on Balls',
    15     'SO': 'Strikeouts',
    16     'IBB': 'Intentional walks',
    17     'HBP': 'Hit by pitch',
    18     'SH': 'Sacrifice hits',
    19     'SF': 'Sacrifice flies',
    20     'GIDP': 'Grounded into double plays'
    21 }
```

Preparing the Data

```
# First, getting rid of unnecessary columns
# We don't need player/team/time identifiers (yet), as we want to look only at the underlying numbers
gamesbatting = df['G_batting']
print(gamesbatting.isnull().sum())

gold = df['G_old']
print(gold.isnull().sum())
# G_batting and G_old are also dropped, as over 98% of values are empty for each

# initial model df
m_df_init = df.drop(columns=['playerID', 'yearID', 'stint', 'teamID', 'lgID', 'G_batting', 'G_old'])
m_df_init.describe().round(2)
```

```
112184
113799
```

	G	AB	R	H	2B	3B	HR	RBI	SB	CS	BB	SO	IBB	HBP	SH	SF	GIDP
count	113799.00	113799.00	113799.00	113799.00	113799.00	113799.00	113799.00	113043.00	111431.00	90257.00	113799.00	111699.00	77148.00	110983.00	107731.00	77695.00	88357.00
mean	50.38	137.42	18.24	35.84	6.14	1.21	2.87	16.60	2.86	1.14	12.69	20.75	1.01	1.07	2.11	1.00	2.84
std	46.74	182.99	27.89	51.87	9.60	2.55	6.41	26.12	7.51	2.63	20.48	29.20	2.65	2.32	4.08	1.91	4.64
min	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
25%	12.00	3.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
50%	34.00	44.00	4.00	8.00	1.00	0.00	0.00	3.00	0.00	0.00	2.00	8.00	0.00	0.00	0.00	0.00	0.00
75%	78.00	220.00	26.00	54.00	9.00	1.00	2.00	24.00	2.00	1.00	17.00	29.00	1.00	1.00	3.00	1.00	4.00
max	165.00	716.00	198.00	262.00	67.00	36.00	73.00	191.00	138.00	42.00	232.00	223.00	120.00	51.00	67.00	19.00	36.00

Imputation

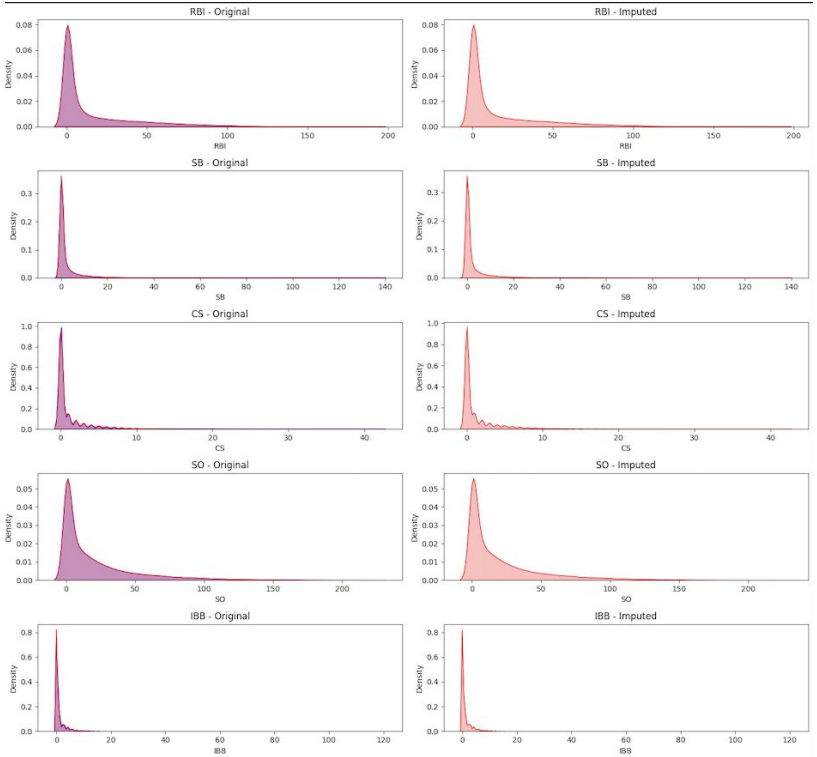
NaN's

	0
G	0
AB	0
R	0
H	0
2B	0
3B	0
HR	0
RBI	756
SB	2368
CS	23542
BB	0
SO	2100
IBB	36651
HBP	2816
SH	6068
SF	36104
GIDP	25442

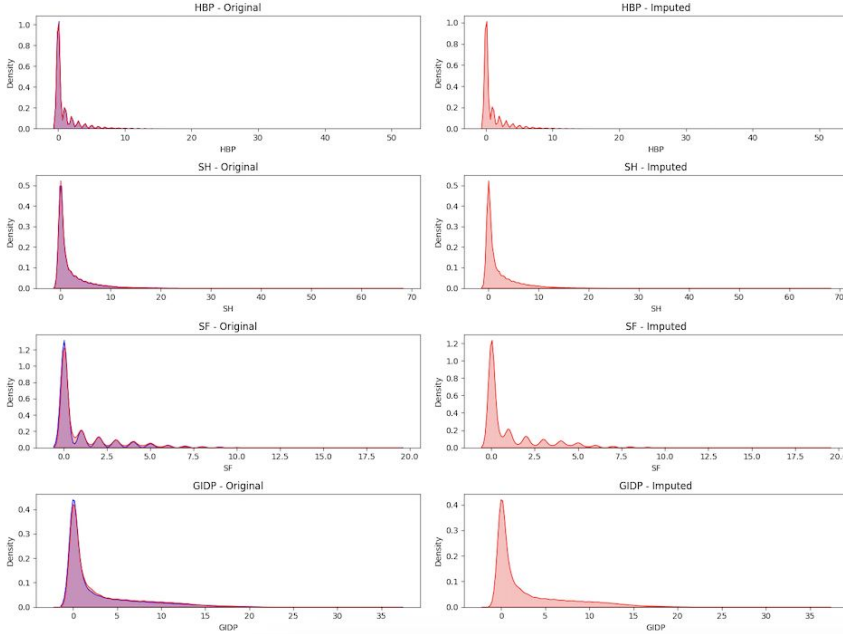
- Slow, about 6 minutes to run
- Split this into smaller steps?
- What's happening?

```
[ ]
targets = ['RBI', 'SB', 'CS', 'SO', 'IBB', 'HBP', 'SH', 'SF', 'GIDP']
other_features = ['G', 'AB', 'R', 'H', '2B', '3B', 'HR', 'BB']
# New df, for imputed values
m_df = m_df_init.copy()

rf_model = RandomForestRegressor()
lr_model = LinearRegression()

# From here, the loop is carried for each feature that's missing values, listed in targets
for target in targets:
    # Splitting the orignal df into empty and filled values
    train_data = m_df[m_df[target].notnull()]
    predict_data = m_df[m_df[target].isnull()]

    # A random forest regressor is used to select the top 3 features for each target
    # Only the top 3 are used, to avoid overfitting as I impute
    rf_model.fit(train_data[other_features], train_data[target])

    feature_importances = rf_model.feature_importances_

    organized_features = np.argsort(feature_importances)[-3:]
    top_3_features = [other_features[i] for i in organized_features]

    if not predict_data.empty:
        # Now, linear regression imputation for missing data
        lr_model.fit(train_data[top_3_features], train_data[target])

        predicted_values = lr_model.predict(predict_data[top_3_features])
        m_df.loc[predict_data.index, target] = predicted_values
```

Resulting KDE Plots:



- Kernel Density Estimation Plots to test validity of imputed data
  - One for each feature
- Like a Histogram
- Looking for Similarity of Shape
- Good here because we have a large dataset, and a lot of imputations



Feature Importances

```python
# Time to define and split the target and features
X = m_df.drop(columns=['HR'])
y = m_df['HR']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=356)
# We already have a random forest model defined

rf_model.fit(X_train, y_train)
```

```python
# Finding feature importances
importances = rf_model.feature_importances_
feature_names = X.columns

feature_df = pd.DataFrame({'Feature': feature_names, 'Importance': importances})
feature_df = feature_df.sort_values(by='Importance', ascending=False)

print(feature_df)
```

Indices are before sorting, so looks funky

	Feature	Importance
6	RBI	0.719214
10	SO	0.087493
13	SH	0.084213
11	IBB	0.016618
2	R	0.013202
9	BB	0.010232
3	H	0.010095
4	2B	0.009429
7	SB	0.008133
1	AB	0.007628
14	SF	0.007028
5	3B	0.006418
0	G	0.005812
15	GIDP	0.005488
8	CS	0.005148
12	HBP	0.003849

Do these make sense intuitively? Surprises?

Feature Engineering

```
## These kinda make sense

# Hits per at bat
m_df['H/AB'] = m_df['H'] / m_df['AB']

# Hits per game
m_df['H/G'] = m_df['H'] / m_df['G']

# Strikeouts per at bat
m_df['SO/AB'] = m_df['SO'] / m_df['AB']

# RBI excluding own runs
m_df['RBI-R'] = m_df['RBI'] - m_df['R']

## Playing around a bit more

# Sacrifice hits times intentional walks
m_df['SH*IBB'] = m_df['SH'] * m_df['IBB']

# intentional walks + hit by pitch
m_df['IBB+HBP'] = m_df['IBB'] + m_df['HBP']

# Stolen bases per at bat
m_df['SB/AB'] = m_df['SB'] / m_df['AB']
```

BAD
——>

- Oh heavens, it turns out you can't divide by 0, and my data has a good amount of seasons with no AB's (among other 0 values)

SH	SF	GIDP	H/AB	H/G	SO/AB	RBI-R	SH*IBB	IBB+HBP	SB/AB
13799.00	113799.00	113799.00	93812.00	113799.00	93841.00	113799.00	113799.00	113799.00	93837.00
2.21	1.11	3.05	0.21	0.45	inf	-1.68	4.39	2.09	NaN
4.08	1.84	4.55	0.12	0.42	NaN	10.28	14.43	3.94	NaN
-0.18	-0.03	-0.90	0.00	0.00	0.00	-108.00	-0.93	-0.10	-inf
0.00	0.00	0.00	0.15	0.00	0.10	-2.00	0.00	0.00	0.00
0.00	0.02	1.00	0.23	0.39	0.18	0.00	0.00	0.00	0.00
3.00	1.70	4.89	0.27	0.80	0.30	1.00	0.73	2.76	0.02
67.00	19.00	36.00	1.00	4.00	inf	58.00	308.95	129.00	inf

- New plan:
  - Remove rows with 0 AB's
  - Is this allowed? Yes, we want to study the relationship between hitting stats and home runs, no AB's means no hitting stats.
  - What about accounting for injuries? Unpredictable/random anyway so not a concern

Fixes

19,987 seasons with no at bats!

	AB
0	19987
1	4289
2	3016
3	2527
4	2266
...	...
683	1
716	1
701	1
704	1
705	1

```
1 m_df2 = m_df.copy()
2 m_df2 = m_df2[m_df2['AB'] != 0]
3
4 m_df2.head()
5
6 m_df2['AB'].value_counts()
```

	AB
1	4289
2	3016
3	2527
4	2266
5	1694
...	...
688	1
716	1
701	1
704	1
705	1

93,812 total rows (seasons) remain

New Feature Importances!

- Same Process with new ‘m_df2’
  - Repeated the Split
  - Applied the newly engineered features, now successful
  - Tested for feature importances

	Feature	Importance
6	RBI	0.710467
10	SO	0.089205
18	SO/AB	0.050457
13	SH	0.043500
11	IBB	0.013717
2	R	0.010787
9	BB	0.008318
4	2B	0.007367
22	SB/AB	0.007304
14	SF	0.005797
5	3B	0.005126
17	H/G	0.005002
16	H/AB	0.004949
3	H	0.004754
21	IBB+HBP	0.004511
15	GIDP	0.004300
19	RBI-R	0.004280
1	AB	0.004111
0	G	0.003956
8	CS	0.003754
20	SH*IBB	0.003145
7	SB	0.002742
12	HBP	0.002449

Now I drop 5 features with the lowest feature importances:

```
[ ]  1 # Dropping the 5 lowest importances
     2 df_drop = X.copy()
     3 df_drop = df_drop.drop(columns=['HBP','SB','SH*IBB','CS','G'])
     4 df_drop.head().round(2)
```

	AB	R	H	2B	3B	RBI	BB	SO	IBB	SH	SF	GIDP	H/AB	H/G	SO/AB	RBI-R	IBB+HBP	SB/AB
1	2	0	0	0	0	0.0	0	0.0	0.00	1.0	0.0	0.0	0.00	0.00	0.00	0.0	0.00	0.0
3	1	0	0	0	0	0.0	0	1.0	0.00	0.0	0.0	0.0	0.00	0.00	1.00	0.0	0.00	0.0
8	1	0	0	0	0	0.0	0	1.0	0.00	0.0	0.0	0.0	0.00	0.00	1.00	0.0	0.00	0.0
9	468	58	131	27	6	69.0	28	39.0	2.96	6.0	4.0	13.0	0.28	1.07	0.08	11.0	5.96	0.0
10	602	105	189	37	9	106.0	49	61.0	5.00	7.0	4.0	20.0	0.31	1.24	0.10	1.0	8.00	0.0

Data is prepped, model time

Training Models

- Redid split to account for dropped features

- 6 Models
  - Linear Regression
  - Regression Tree (Depth=5)
  - Bagging
  - Random Forest
  - Gradient Boosting
  - XGBoosting

- Took a bit to run, but nothing crazy and ran all at once

```
41 # XGBoosting
42 xgb_reg = xgb.XGBRegressor(n_estimators=100)
43 xgb_reg.fit(X_train, y_train)
44 y_pred_xgb = xgb_reg.predict(X_test)
45 MSE_xgb = mean_squared_error(y_test, y_pred_xgb)
46 RMSE_xgb = np.sqrt(MSE_xgb)
47 R2_xgb = r2_score(y_test, y_pred_xgb)
```

Repeated for each:

- Defined the model
- Fit to train data
- y_pred
- MSE (sklearn)
- RMSE (np)
- R2 (sklearn)

Success?

XGBoost wins, highest coefficient of determination. Also lowest RMSE, in context about 1.9 HR's away from actual

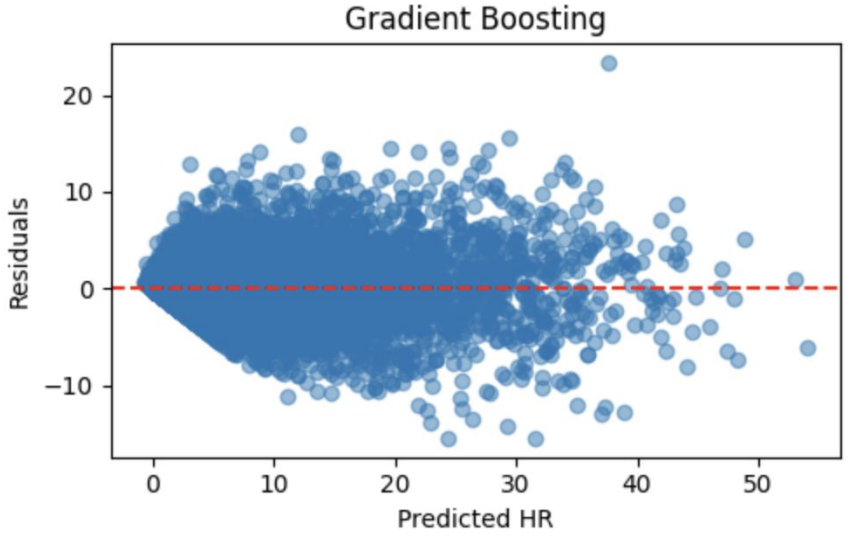
```
# MSE, RMSE, and R^2 of the six models, printing into dataframe
MSE = []
RMSE = []
R2 = []
models = [lin_reg, reg_tree, bag_reg, rf_reg, gb_reg, xgb_reg]
titles = ['Linear Regression', 'Regression Tree', 'Bagging', 'Random Forest', 'Gradient Boosting', 'XGBoosting']
for i, model in enumerate(models):
  y_pred = model.predict(X_test)
  MSE.append(mean_squared_error(y_test, y_pred))
  RMSE.append(np.sqrt(MSE[i]))
  R2.append(r2_score(y_test, y_pred))

df_output = pd.DataFrame({'Model': titles, 'MSE': MSE, 'RMSE': RMSE, 'R^2': R2})
print(f'{df_output}\n')
```

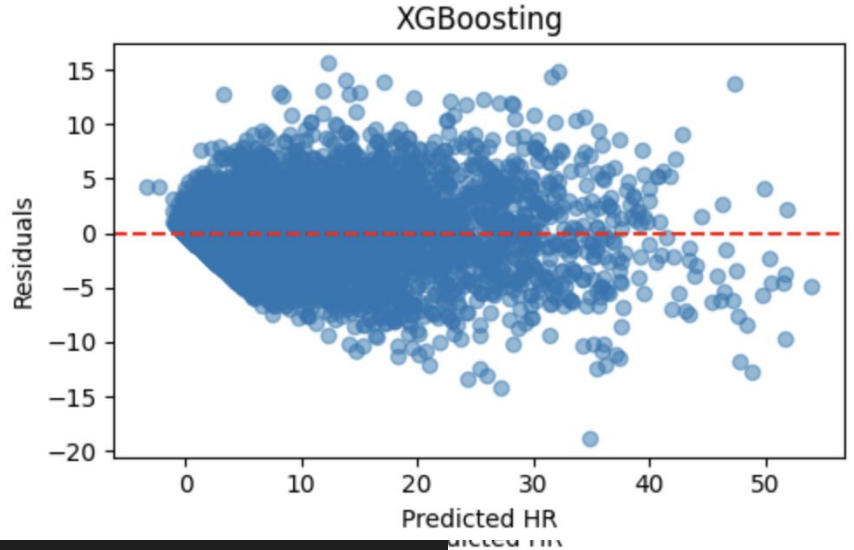
	Model	MSE	RMSE	R^2
0	Linear Regression	5.649847	2.376941	0.881461
1	Regression Tree	7.197520	2.682819	0.848989
2	Bagging	3.828586	1.956677	0.919673
3	Random Forest	3.813810	1.952898	0.919983
4	Gradient Boosting	4.468011	2.113767	0.906257
5	XGBoosting	3.588124	1.894234	0.924718

Visualization - Residual Plots

(y - y_pred)

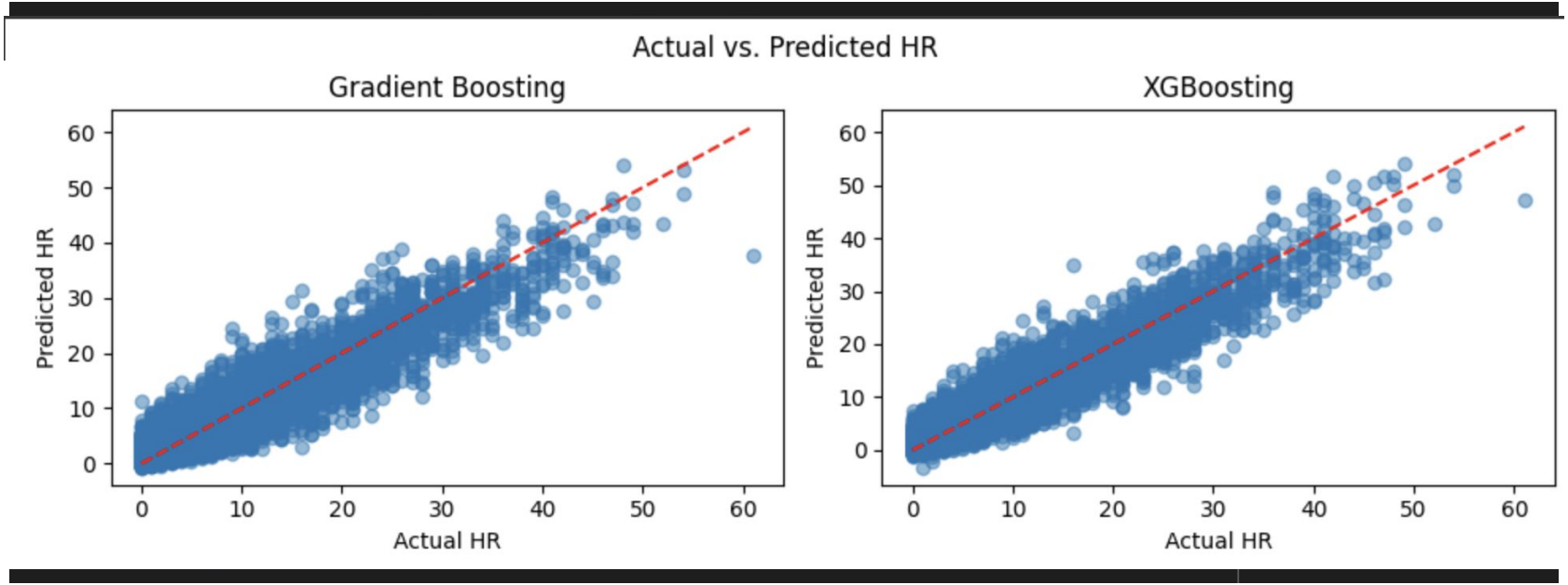
```
1 # First, plotting residuals
```






```
18 plt.tight_layout()
19 plt.show()
```

Visualization - Actual vs Prediction



Modeling off Previous Season

So, starting from the top(ish)

- First, copying csv df and dropping unnecessary columns
- Second, get df to the correct shape by removing 0 AB seasons

```
[ ]  1 df2 = df.copy()
     2
     3 # Keeping yearID this time, it's what we're isolating
     4 df2.drop(columns=['stint', 'teamID', 'lgID', 'G_batting', 'G_old'], inplace=True)
     5 df2.head()
```

	playerID	yearID	G	AB	R	H	2B	3B	HR	RBI	SB	CS	BB	SO	IBB	HBP	SH	SF	GIDP
0	aardsda01	2004	11	0	0	0	0	0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0
1	aardsda01	2006	45	2	0	0	0	0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	1.0	0.0	0.0
2	aardsda01	2007	25	0	0	0	0	0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0
3	aardsda01	2008	47	1	0	0	0	0	0	0.0	0.0	0.0	0	1.0	0.0	0.0	0.0	0.0	0.0
4	aardsda01	2009	73	0	0	0	0	0	0	0.0	0.0	0.0	0	0.0	0.0	0.0	0.0	0.0	0.0

```
1 df_year = df2[df2['AB'] != 0]
2 print(df_year['AB'].value_counts())
3
4 df_year.head(10)
```

- Note: fortunately, the df was already in the correct order, sorting by playerID then yearID, which is what we want

- Third, adding columns to our old df_drop

```
1 df_drop['yearID'] = df_year['yearID']
2 df_drop['playerID'] = df_year['playerID']
3 df_drop['HR'] = df_year['HR']
4 df_drop.head(10).round(2)
```

Shifting by One

- Grouping by all unique playerID's
- Then shifting values associated with THAT player down by one
- Combining (we need HR in the right place)
- NaN's are good sign, meaning there's no previous season. This means that players rookie years aren't assigned with other players final years.

```
# Creating shifted features
lagged_features = df_drop.groupby('playerID').shift(1)

# Renaming features
lagged_features.columns = [f'{col}_prev_year' for col in lagged_features.columns]

# Combining df's
df_lagged = pd.concat([df_drop, lagged_features], axis=1)

print(df_lagged.shape)
df_lagged.head()
```

```
(93812, 41)
```

	AB	R	H	2B	3B	RBI	BB	SO	IBB	SH	...	SF_prev_year	GIDP_prev_year	H/A
1	2	0	0	0	0	0.0	0	0.0	0.000000	1.0	...	NaN	NaN	
3	1	0	0	0	0	0.0	0	1.0	0.000000	0.0	...	0.0	0.0	
8	1	0	0	0	0	0.0	0	1.0	0.000000	0.0	...	0.0	0.0	
9	468	58	131	27	6	69.0	28	39.0	2.961088	6.0	...	NaN	NaN	
10	602	105	189	37	9	106.0	49	61.0	5.000000	7.0	...	4.0	13.0	

Last Data Prep

```
# First, initial years data
df_lagged.drop(columns=['AB', 'R', 'H', '2B', '3B', 'RBI', 'BB', 'SO','IBB',
                        'SH', 'SF', 'GIDP', 'H/AB', 'H/G', 'SO/AB','RBI-R',
                        'IBB+HBP','SB/AB'], inplace=True)
df_lagged.head()
```

```
# Now, rookie years, so drop every row with no previous yearID data
df_lagged.dropna(subset=['yearID_prev_year'], inplace=True)
print(df_lagged.isnull().sum().sum())
df_lagged.head(10)
```

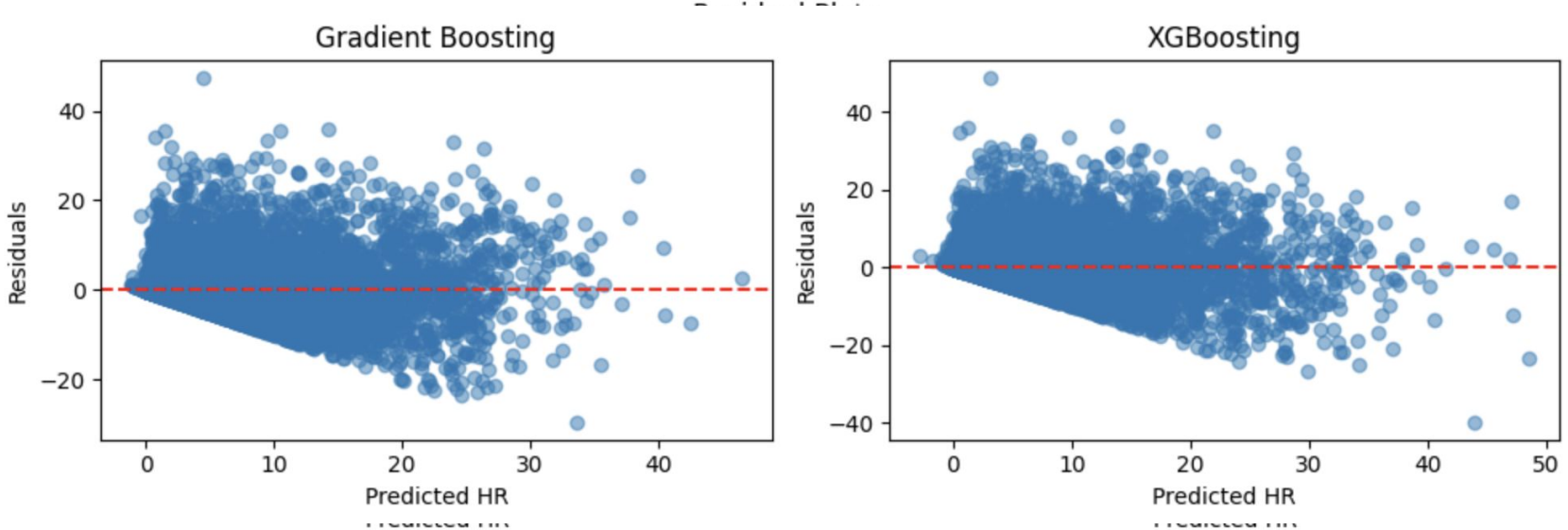
0

	yearID	playerID	HR	AB_prev_year	R_prev_year	H_prev_year	2B_prev_year	3B_prev_year	RBI_prev_year	BB_prev_year	...
3	2008	aardsda01	0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	...
8	2015	aardsda01	0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...
10	1955	aaronha01	27	468.0	58.0	131.0	27.0	6.0	69.0	28.0	...
11	1956	aaronha01	26	602.0	105.0	189.0	37.0	9.0	106.0	49.0	...
12	1957	aaronha01	44	609.0	106.0	200.0	34.0	14.0	92.0	37.0	...
13	1958	aaronha01	30	615.0	118.0	198.0	27.0	6.0	132.0	57.0	...
14	1959	aaronha01	39	601.0	109.0	196.0	34.0	4.0	95.0	59.0	...
15	1960	aaronha01	40	629.0	116.0	223.0	46.0	7.0	123.0	51.0	...
16	1961	aaronha01	34	590.0	102.0	172.0	20.0	11.0	126.0	60.0	...
17	1962	aaronha01	45	603.0	115.0	197.0	39.0	10.0	120.0	56.0	...

New models (same code)

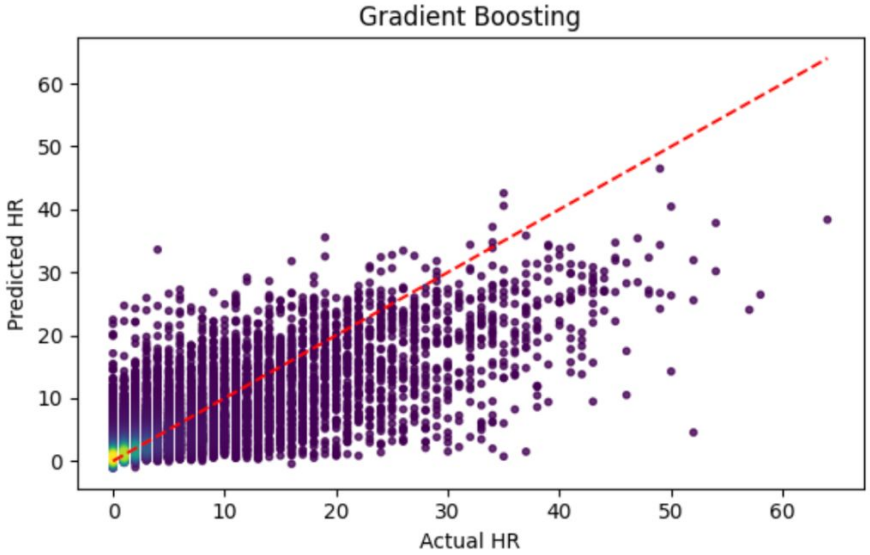
	Model	MSE	RMSE	R^2
0	Linear Regression	22.255131	4.717534	0.594182
1	Regression Tree	22.791565	4.774051	0.584400
2	Bagging	22.211880	4.712948	0.594970
3	Random Forest	22.317416	4.724131	0.593046
4	Gradient Boosting	21.431541	4.629421	0.609200
5	XGBoosting	22.351336	4.727720	0.592427

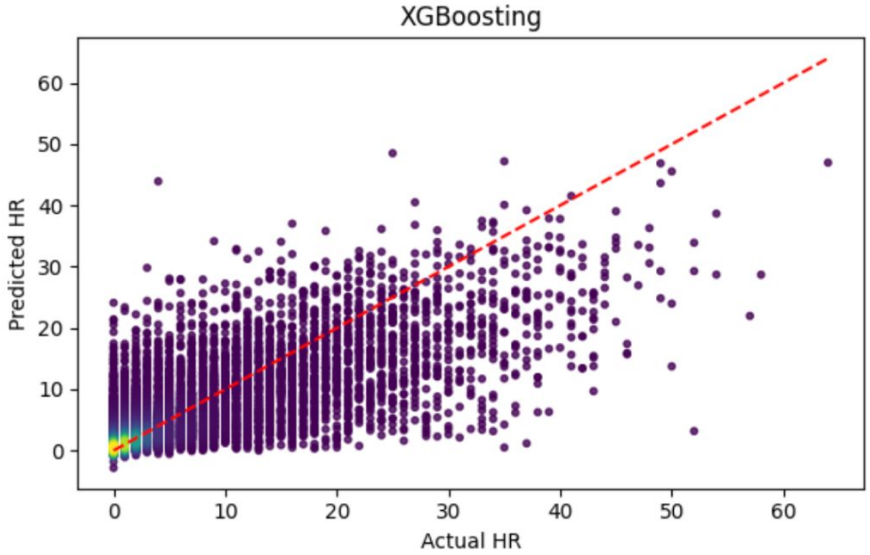
Residuals


Gradient Boosting
Residuals
Predicted HR
40
20
0
−20
0
10
20
30
40
XGBoosting
Residuals
Predicted HR
40
20
0
−20
−40
0
10
20
30
40
50

Actual vs Predicted

```
fig, axs = plt.subplots(3, 2, figsize=(12, 12))
```






```
    ax.set_ylabel('Predicted HR')

plt.tight_layout()
plt.show()
```

Improving Ideas

- Feature importances for staggered model, separate feature engineering would be better practice
- Log scale graphs when showing density
- Improvements to computational efficiency
  - Currently 20-30 minutes to run all cells
- Would I have more success if I removed rows with 0 AB's BEFORE I imputed data early on? Either way better order of operations
- Career stats instead of previous year
  - Idea 1: Using a players average stats (per season) to predict home runs
    - Pros: Easier next step, would account for outlier seasons
    - Cons: Wouldn't account for player development/regression
  - Idea 2: Using data from each previous year played (remaining separate) to predict home runs
    - Pros: Accounts for development/regression*, Would account for outlier players (elite) AND seasons, overall better predictive power
    - Cons: Harder to build, probably computationally intensive (each players own df)
  - I believe both ideas would improve upon my current model to some extent, and I aim to continue working, starting with the simpler idea 1

*I would likely first have to provide the model with some way of recognizing average player development and regression, which may mean building a separate model first? Necessary maybe not, but would certainly improve the model if I could pull it off

Questions?