Linear Regression Model Analysis for Machine Learning

Main objective of analysis

The objective of this analysis is to select a proper linear regression model to predict the winnings for Major League Baseball teams.

The analysis is conducted with linear regression approaches, which starts from vanilla linear regression and followed by regularization with Lasso and Ridge regression.

The expected outcome of this analysis will be a suggested linear regression model with proper hyperparameter setting that generates the highest R2 score.

Brief description of the data set and a summary of its attributes

The data set contains historical baseball statistics for Major League Baseball teams from 1871 through 2014. The statistics cover all personal performance data such as batting, pitching, and also “general” team performance data. In this study, I am going to focus on team performance data and use it as the baseline to create a proper model in order to predict team winnings.

Initial plan for data exploration

The data exploration plan is described as following:

1. Data description and insight.
    a. The source of data.
    b. Feature identification.
    c. Relationship (correlation) between features.
2. Data cleaning and data integrity
    a. Are there any missing values present? Are we going to fix them and how?
    b. Are there any outliers in the dataset? Are we going to fix them and how?
3. Feature engineering.
    a. To create new features from the existing data set if necessary.

Actions taken for data cleaning and feature engineering

Based on the data exploratory plan, here are the actions I have taken for the data exploratory:

1. Describe the data set.
   As a baseball fan, it is always fun to analyze what a winning team is made of. How do batting, pitching and defense performance contribute to a team's success? Is it purely a money game of acquiring superstars, or can we develop some team chemistry with highly potential but less expensive players?
   Therefore, I start my study with a baseball data set created by Sean Lahman, which contains comprehensive Major League Baseball records from 1871 through 2014.

   The full data set can be retrieved here:
   http://www.seanlahman.com/baseball-archive/statistics/

   The data set includes 27 CSV data files and one readme TXT file. Here is the file list on my working environment:

```
  215,487 AllstarFull.csv
6,568,014 Appearances.csv
    8,019 AwardsManagers.csv
  246,487 AwardsPlayers.csv
   22,464 AwardsShareManagers.csv
  225,729 AwardsSharePlayers.csv
6,697,820 Batting.csv
  941,195 BattingPost.csv
  404,474 CollegePlaying.csv
7,134,518 Fielding.csv
  286,443 FieldingOF.csv
1,671,765 FieldingOFsplit.csv
  724,299 FieldingPost.csv
  175,319 HallOfFame.csv
  163,256 HomeGames.csv
  133,932 Managers.csv
    3,474 ManagersHalf.csv
   11,651 Parks.csv
2,646,243 People.csv
4,275,780 Pitching.csv
  520,762 PitchingPost.csv
   29,765 readme2014.txt
  774,214 Salaries.csv
   61,246 Schools.csv
   10,685 SeriesPost.csv
  585,193 Teams.csv
    3,238 TeamsFranchises.csv
    1,556 TeamsHalf.csv
```

2. Identify the features and check the data types.
   The study will start from the team's overall performance. Through the data set I would like to figure out key offensive and defensive factors that contribute to a team's annual winnings and ranking. Furthermore, it is also interesting to see how a team's overall payroll correlates to its performance. Therefore, two CSV files, Teams.csv and Salaries.csv will become the major data source of analytics.

Let’s check the columns and data types of these 2 CSV files:

a. Teams.csv

Columns	Description	Data Type
yearID	Year	int64
lgID	League	object
teamID	Team	object
franchID	Franchise	object
divID	Team's division	object
Rank	Position in final standings	int64
G	Games played	int64
GHome	Games played at home	float64
W	Wins	int64
L	Losses	int64
DivWin	Division Winner (Y or N)	object
WCWin	Wild Card Winner (Y or N)	object
LgWin	League Champion(Y or N)	object
WSWin	World Series Winner (Y or N)	object
R	Runs scored	int64
AB	At bats	int64
H	Hits by batters	int64
2B	Doubles	int64
3B	Triples	int64
HR	Homeruns by batters	int64
BB	Walks by batters	int64
SO	Strikeouts by batters	float64
SB	Stolen bases	float64
CS	Caught stealing	float64
HBP	Batters hit by pitch	float64
SF	Sacrifice flies	float64
RA	Opponents runs scored	int64
ER	Earned runs allowed	int64
ERA	Earned run average	float64

CG	Complete games	int64
SHO	Shutouts	int64
SV	Saves	int64
IPOuts	Outs Pitched (innings pitched x 3)	int64
HA	Hits allowed	int64
HRA	Homeruns allowed	int64
BBA	Walks allowed	int64
SOA	Strikeouts by pitchers	int64
E	Errors	int64
DP	Double Plays	float64
FP	Fielding percentage	float64
name	Team's full name	object
park	Name of team's home ballpark	object
attendance	Home attendance total	float64
BPF	Three-year park factor for batters	int64
PPF	Three-year park factor for pitchers	int64
teamIDBR	Team ID used by Baseball Reference website	object
teamIDlahman45	Team ID used in Lahman database version 4.5	object
teamIDretro	Team ID used by Retrosheet	object

b. Salaries.csv

Columns	Description	Data Type
yearID	Year	int64
teamID	Team	object
lgID	League	object
playerID	Player ID code	object
salary	Salary	int64

3. Missing value treatment and outlier removal.
    a. Data selection and segmentation

Although the Teams.csv data file provides us 143 year-long team performance data (1871 to 2014), I will have my focus on modern baseball statistics ranging from 1985 to 2012. The reasons are as following:

i. Recent baseball statistics are more representative of modern baseball competition.
ii. The salary data will also be part of the data set, and the earliest team salary data recorded in the Salaries.csv is 1985 year - data.

Therefore, the data set is streamlined to 798 records.

By examining the data columns, I have found that some of them are less relevant to the study subject. Columns such as 'park', 'teamIDBR', 'teamIDlahman45', and 'teamIDretro' are either name or ID related which I believe could be removed from the data set.

As a result, the streamlined Teams data set contains 798 records with 44 columns.

```
# Extract 1985-2012 data

# Make a data copy
team_df = raw_teams_df.copy(deep=True)
team_df = team_df.loc[(team_df['yearID']>1984) & (team_df['yearID']<2013)]
# print(team_df.shape)

# Remove unnecessary columns
unnecessary_columns = ['park', # Name of team's home ballpark
                       'teamIDBR', # Team ID used by Baseball Reference website
                       'teamIDlahman45', # Team ID used in Lahman database version 4.5
                       'teamIDretro'] # Team ID used by Retrosheet
team_df.drop(columns=unnecessary_columns, axis=1, inplace=True)
print(team_df.shape)
```

```
(798, 44)
```

As mentioned, the team payroll data is part of the data set. Therefore I merge team statistics with each team's 22-men roster payroll data.

```
In [364]: # Here we merge the teams data set with salaries data set
          # so that we can put into team salaries data as indepedent variable.
          salaries_df = raw_salaries_df.copy(deep=True)
          salaries_df = salaries_df.loc[(salaries_df['yearID']>1984) & (salaries_df['yearID']<2013)]
          aggregated_salaries_df = salaries_df.groupby(by=['yearID','teamID'], axis=0, as_index=False).sum()
          aggregated_salaries_df

          team_df = team_df.merge(right=aggregated_salaries_df, on=['yearID','teamID'])
          team_df.head()
```

Out[364]:

	yearID	lgID	teamID	franchID	divID	Rank	G	Ghome	W	L	...	BBA	SOA	E	DP	FP	name	attendance	BPF	PPF	salary
0	1985	NL	ATL	ATL	W	5	162	81.0	66	96	...	642	776	159	197	0.976	Atlanta Braves	1350137.0	105	106	14807000
1	1985	AL	BAL	BAL	E	4	161	81.0	83	78	...	568	793	129	168	0.979	Baltimore Orioles	2132387.0	97	97	11560712
2	1985	AL	BOS	BOS	E	5	163	81.0	81	81	...	540	913	145	161	0.977	Boston Red Sox	1786633.0	104	104	10897560
3	1985	AL	CAL	ANA	W	2	162	79.0	90	72	...	514	767	112	202	0.982	California Angels	2567427.0	100	100	14427894
4	1985	AL	CHA	CHW	W	3	163	81.0	85	77	...	569	1023	111	152	0.982	Chicago White Sox	1669888.0	104	104	9846178

5 rows × 45 columns

The dimension of the selected data set is 798 records with 45 columns.

b. Missing data treatment

Basically, the 2 data files, Teams.csv and Salaries.csv, are relatively "clean" in terms of missing value. However, they are not that perfect and ready to be analyzed.

The missing value is due to the strike in 1994. Since the regular season was interrupted due to the strike, there were no division, league and world series winners respectively for the entire MLB teams.

The outcome is obvious and straightforward: there are no values from DivWin (division winner), WCWin (wild card winner), LgWin (league winner), and WSWin (world series winner) columns. Since these are categorical type columns, I simply update these missing values into 'N' for the sake of consistency.

Another missing part is the WCWin missing prior to 1994. This is because the wild card was first instituted in MLB in 1994, so it is reasonable that there would be no such value from 1985 to 1993. And even though the wild card rule was set in 1994, the season was unfortunately interrupted due to the strike which led to no wildcard value in that year too.

In conclusion, for missing DivWin, LgWin, and WSWin values in 1994, I reset them to 'N'. For missing WCWin values from 1985 to 1994, I also reset them to N.

```
In [394]: # check these columns
for na_column in na_columns:
    print(na_column + ' : ' + str(main_df[na_column].unique()))
# Fix the nan value to N
# DivWin, LgWin WSWin are missing in 1994 due to the strike, so we fix it to N.
# We also set N to wild card win (WCWin) since it was first introduced in 1994.
main_df.replace(np.nan, 'N', inplace=True)
# check the fixed data set; there should be no nan now.
main_df[['DivWin','WCWin','LgWin','WSWin']].info()
```

```
DivWin : ['N' 'Y']
WCWin : ['N' 'Y']
LgWin : ['N' 'Y']
WSWin : ['N' 'Y']
<class 'pandas.core.frame.DataFrame'>
Int64Index: 798 entries, 0 to 797
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   DivWin  798 non-null    object
 1   WCWin   798 non-null    object
 2   LgWin   798 non-null    object
 3   WSWin   798 non-null    object
dtypes: object(4)
memory usage: 31.2+ KB
```

Now the data set does not contain any missing value.

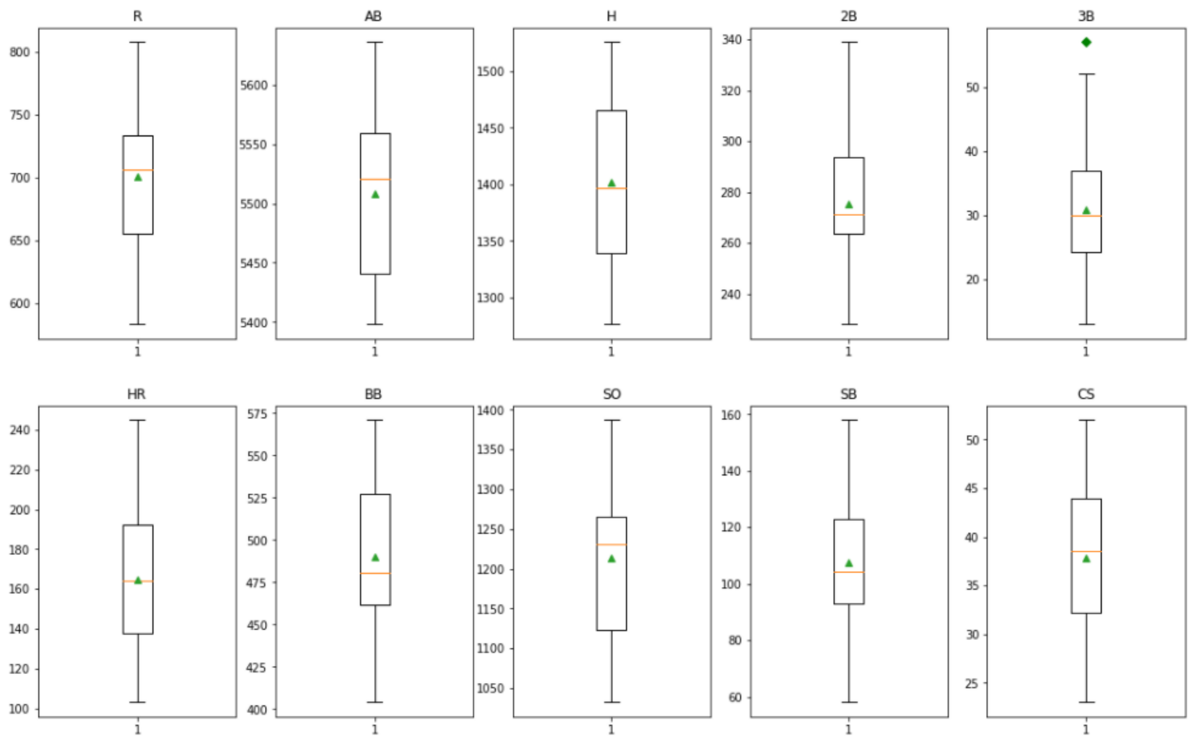
c. Outlier

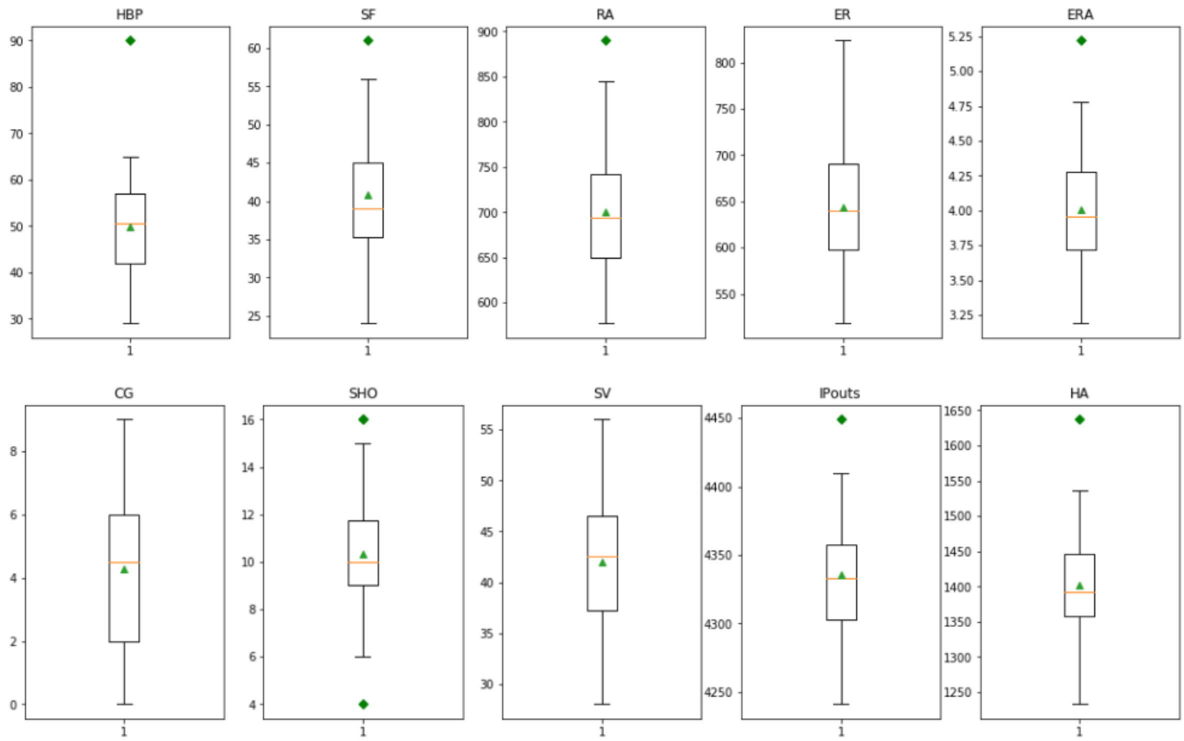
After checking the missing data, it comes to identify the potential outliers in the data set. Firstly, I pick up all numeric columns and get rid of non - numeric ones. Then, for some numeric columns, such as ID related and attendance, are less statistically meaningful, therefore these columns are excluded from the checking list as well.

As a result, the candidate columns to be verified are: 'R', 'AB', 'H', '2B', '3B', 'HR', 'BB', 'SO', 'SB', 'CS', 'HBP', 'SF', 'RA', 'ER', 'ERA', 'CG', 'SHO', 'SV', 'IPouts', 'HA', 'HRA', 'BBA', 'SOA', 'E', 'DP', 'FP', 'salary', 'OBP', 'SLG', and 'OPS' respectively. Even though some of them may not be utilized for modeling and prediction, it is no harm to have a check to each of them.

In addition, since the data set covers MLB team statistics ranging from 1985 to 2012, it is more reasonable to verify outliers separately on a yearly manner instead of checking them on a mixed, multi - year data set to prevent misleading.

I will use visualization tools such as boxplot to identify potential outliers. Here is some samples of visualization on 2012 year data:





There is an interesting fact regarding the outlier verification. There are some outliers within each yearly data, however, if we compare the results from multiple yearly data, we can see the features that contain outliers are changing among the years. For example, in 1989 data there are outliers in the ‘H’ feature and no outliers in ‘SB’ feature. However it is just the opposite in 1990 data since the outlier appears in ‘SB’ but not in ‘H’.

I believe it is the nature of baseball game statistics. Let’s say, some teams were focusing on producing short - range hits so that they had outstanding ‘H’ figures, and in 1990 some teams had recruited players with astonishing speed so that the ‘SB’ (Stolen Base) figures were sky - high. Therefore, the outliers on baseball statistics may be regarded as the combination of talented players, tactics and other factors and it is a part of the game. I would rather keep these figures instead of removing them.

4. Feature engineering.
    a. Transform categorical data

Inside the data set, there are four categorical columns that may have effects on future analysis: DivWin, WCWin, LgWin, and WSWin. These columns are indicators of postseason winnings to a team and are represented in value Y (winner) and N.

For the sake of simplicity, I will transform value Y into numeric value 1 and value N to so that I can directly utilize these features for modeling.

```
In [477]: # Transform categorical data
main_df['DivWin'].replace(to_replace={'N':0, 'Y':1}, inplace=True)
main_df['WCWin'].replace(to_replace={'N':0, 'Y':1}, inplace=True)
main_df['LgWin'].replace(to_replace={'N':0, 'Y':1}, inplace=True)
main_df['WSWin'].replace(to_replace={'N':0, 'Y':1}, inplace=True)
main_df[['DivWin', 'WCWin', 'LgWin', 'WSWin']]
```

Out[477]:

	DivWin	WCWin	LgWin	WSWin
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
...	...	...	...	...
793	0	1	0	0
794	0	0	0	0
795	0	1	0	0
796	0	0	0	0
797	1	0	0	0

798 rows × 4 columns

b. Transform text (object) value to dummies

There are also four text columns in the data set, which are teamID, franchID, lgID and divID. For the teamID and franchID I believe they are for the team identification usage only and have little effect on modeling. Therefore I tend not to do anything to them at this moment.

However for lgID and divID, since there would be a scenario to analyze teams' performance within and across the leagues and divisions, so I decide to transform these columns into dummy features:

- lgID_AL (American League)
- lgID_NL (National League)
- divID_C (Central Division)
- divID_E (Eastern Division)
- divID_W (Western Division)

```
43  lgID_AL    798 non-null   uint8
44  lgID_NL    798 non-null   uint8
45  divID_C    798 non-null   uint8
46  divID_E    798 non-null   uint8
47  divID_W    798 non-null   uint8
```

c. Create domain data

Based on modern baseball game statistics and observation, several interesting indicators have been introduced as key indexes to evaluate offensive performance. The most widely adapted one is On-base Plus Slugging (OPS), which is the summary of On - Base Percentage (OBP) and Slugger (SLG).

The formula is described as below:

- OBP (On-Base Percentage) =（H + BB + HBP）/（AB + BB + HBP + SF）
- SLG (Slugging Percentage) = ((1*H)+(2*2B)+(3*3B)+(4*HR))/AB
- OPS = OBP + SLG

To provide more comprehensive review of teams' performance, I will add these 3 features into the data set:

name	attendance	salary	OBP	SLG	OPS
Atlanta Braves	1350137.0	14807000	0.314881	0.429425	0.744306
Baltimore Orioles	2132387.0	11560712	0.335599	0.514954	0.850552
Boston Red Sox	1786633.0	10897560	0.346522	0.513986	0.860508
California Angels	2567427.0	14427894	0.332738	0.459206	0.791945
Chicago White Sox	1669888.0	9846178	0.315143	0.470750	0.785893
...	...	...	...	...	...
St. Louis Cardinals	3262109.0	110300862	0.337542	0.507471	0.845013
Tampa Bay Rays	1559681.0	64173500	0.316691	0.478511	0.795202
Texas Rangers	3460280.0	120510974	0.333603	0.541682	0.875285
Toronto Blue Jays	2099663.0	75009200	0.309241	0.491708	0.800949
Washington Nationals	2370794.0	80855143	0.322152	0.520214	0.842366

Variation of models and model selection

Here to check again the data set that is going to be analyzed:

```
In [16]: main_df.shape
Out[16]: (798, 51)
```

Among these columns, there are only two with object data types, which are teamID and franchID. They are maintained for descriptive usage only and are not included in the analysis model. The yearID, although it is numerical, is also maintained for grouping only.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 798 entries, 0 to 797
Data columns (total 51 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   yearID       798 non-null    int64
 1   teamID       798 non-null    object
 2   franchID     798 non-null    object
 3   Rank         798 non-null    int64
 4   G            798 non-null    int64
 5   Ghome        798 non-null    float64
 6   W            798 non-null    int64
 7   L            798 non-null    int64
 8   DivWin       798 non-null    int64
 9   WCWin        798 non-null    int64
 10  LgWin        798 non-null    int64
 11  WSWin        798 non-null    int64
 12  R            798 non-null    int64
 13  AB           798 non-null    int64
 14  H            798 non-null    int64
 15  2B           798 non-null    int64
 16  3B           798 non-null    int64
 17  HR           798 non-null    int64
 18  BB           798 non-null    float64
 19  SO           798 non-null    float64
 20  SB           798 non-null    float64
 21  CS           798 non-null    float64
 22  HBP          798 non-null    float64
 23  SF           798 non-null    float64
 24  RA           798 non-null    int64
 25  ER           798 non-null    int64
 26  ERA          798 non-null    float64
 27  CG           798 non-null    int64
 28  SHO          798 non-null    int64
 29  SV           798 non-null    int64
 30  IPouts       798 non-null    int64
 31  HA           798 non-null    int64
 32  HRA          798 non-null    int64
 33  BBA          798 non-null    int64
 34  SOA          798 non-null    int64
 35  E            798 non-null    int64
 36  DP           798 non-null    int64
 37  FP           798 non-null    float64
 38  name         798 non-null    object
 39  attendance   798 non-null    float64
 40  BPF          798 non-null    int64
 41  PPF          798 non-null    int64
 42  salary       798 non-null    int64
 43  lgID_AL      798 non-null    uint8
 44  lgID_NL      798 non-null    uint8
 45  divID_C      798 non-null    uint8
 46  divID_E      798 non-null    uint8
 47  divID_W      798 non-null    uint8
 48  OBP          798 non-null    float64
 49  SLG          798 non-null    float64
 50  OPS          798 non-null    float64
```

1. Define dependent and independent variables

The objective of this analysis is to predict team winnings. Therefore, the “W” column is the dependent variable.

There are 47 other columns excluding “W” and some descriptive columns, however, I am not going to take all of them as independent variables. The major reason is to avoid multicollinearity and over - complexity.
The initial collinearity check on the columns looks like following:

```
In [46]: # columns = ['SB', 'CS', 'ERA', 'SV', 'FP', 'salary', 'OPS']
         # columns = ['ERA', 'SV', 'FP', 'salary', 'OPS']
         # X = main_df[columns]
         X = main_df.drop(columns=['W', 'yearID', 'teamID', 'franchID', 'Rank', 'name'])
         y = main_df[['W']]
         # collinearity check
         c = check_collinearity(X)
         print(c.sort_values(by='VIF', ascending=False))
```

```
     variables           VIF
41      divID_W           inf
40      divID_E           inf
37      lgID_AL  9.007199e+15
38      lgID_NL  9.007199e+15
39      divID_C  4.503600e+15
43          SLG  5.004000e+14
42          OBP  4.740631e+14
44          OPS  7.764827e+13
8            AB  7.910836e+02
9             H  6.962946e+02
12           HR  5.347585e+02
20           ER  4.228249e+02
21          ERA  3.015761e+02
25       IPouts  2.807865e+02
13           BB  2.630986e+02
10           2B  1.226013e+02
19           RA  1.121428e+02
0             G  1.042462e+02
30            E  7.872958e+01
34          BPF  7.584450e+01
35          PPF  7.573013e+01
32           FP  6.922004e+01
1         Ghome  2.777214e+01
7             R  2.751044e+01
2             L  1.996146e+01
11           3B  1.832546e+01
26           HA  1.529565e+01
17          HBP  1.439818e+01
27          HRA  4.499898e+00
29          SOA  3.693973e+00
28          BBA  3.425075e+00
36       salary  3.383319e+00
14           SO  3.328390e+00
16           CS  3.209339e+00
24           SV  3.206821e+00
22           CG  3.003216e+00
18           SF  2.749842e+00
33   attendance  2.430130e+00
5         LgWin  2.330192e+00
23          SHO  2.264632e+00
15           SB  2.264246e+00
3        DivWin  2.189370e+00
31           DP  2.171072e+00
6         WSWin  1.981983e+00
4         WCWin  1.375637e+00
```

It is clear that if we take all of the columns as independent variables, then it is very likely that we would have a very unstable model due to the large multicollinearity shown as above.

To fix the multicollinearity as much as possible, I look into the columns and streamline them into offensive, pitching and defensive perspectives. Take OPS as an example:

a. OBP (On-Base Percentage) =（H + BB + HBP）/（AB + BB + HBP + SF）
b. SLG (Slugging Percentage) = ((1*H)+(2*2B)+(3*3B)+(4*HR))/AB
c. OPS = OBP + SLG

So theoretically I can only keep OPS and remove OBP, SLG and other offensive indicators that contribute to OPS calculation. Based on the same logic, for the pitching and defensive columns I choose to keep representative ones. As the result, the first release of independent variables are:

Perspective	Independent Variables
Offensive	OPS, SB
Pitching	ERA, SV
Defensive	FP, CS
Payroll	salary

Let’s again check the multicollinearity:

```
In [47]: columns = ['SB', 'CS', 'ERA', 'SV', 'FP', 'salary', 'OPS']
         # columns = ['ERA', 'SV', 'FP', 'salary', 'OPS']
         X = main_df[columns]
         # X = main_df.drop(columns=['W', 'yearID', 'teamID', 'franchID', 'Rank', 'name'])
         y = main_df[['W']]
         # collinearity check
         c = check_collinearity(X)
         print(c.sort_values(by='VIF', ascending=False))

           variables         VIF
         6       OPS  406.498714
         4        FP  387.603739
         2       ERA   93.536406
         3        SV   38.159911
         1        CS   22.076440
         0        SB   17.510795
         5    salary    4.139910
```

It is much smaller than the initial variable set. Therefore I would take these 7 variables into the modeling.

2. Define train and test data set

Once the independent variables are set, it is necessary to split the main data set into train and test sets so that we can minimize the potential for bias in the evaluation and validation process.
Here I use the train_test_split function with 30% test data to distinguish training and testing sets.
As an alternative, I would also use KFold and cross_val_predict functions to separate training and testing data as the comparison.

3. Model varification

I would start the model verification from simple linear regression, then come up with Lasso and Ridge models to see the effects of regularization. The R square score, mean square error and mean absolute error would be used to validate the model quality.

a. Vanilla linear regression (baseline and polynomial)

The first try is to implement simple linear regression through train_test_split. The train set is to train our model and then use the test set to generate the prediction.

```
In [170]: # Train test split and linear regression
          # Target is to predict winnings by independent variables
          # W is our dependent variable, and also get rid of some irrelavant (descriptive) variables

          # Create train and test split
          X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
          lg = LinearRegression()
          lg.fit(X_train, y_train)
          y_hat = lg.predict(X_test)
          r2, mse, mae = check_score(y_test, y_hat)
          print('The R2 score is {}'.format(r2))
          print('The MSE score is {}'.format(mse))
          print('The MAE score is {}'.format(mae))

          The R2 score is 0.8180062845836948
          The MSE score is 27.610860573073253
          The MAE score is 3.8506144807578
```

The R square score, mean square error and mean absolute error are 0.8180, 27.6108 and 3.8506 respectively.

Alternatively, by using KFold with 6 split to generate train and test sets, here is the prediction and statistics:

```
In [318]: # Use KFold
          fold=6
          kf = KFold(n_splits=fold, shuffle=True, random_state=1)
          r2s = []
          mses = []
          maes = []
          coef_kf = []
          kf_coef_df = pd.DataFrame()
          for train_index, test_index in kf.split(X):
              X_train_kf, X_test_kf = X.values[train_index], X.values[test_index]
              y_train_kf, y_test_kf = y.values[train_index], y.values[test_index]
              lg.fit(X_train_kf, y_train_kf)
              y_hat_kf = lg.predict(X_test_kf)
              r2, mse, mae = check_score(y_test_kf, y_hat_kf)
              r2s.append(r2)
              mses.append(mse)
              maes.append(mae)
              coef_kf.append(lg.coef_)
          #     print(lg.coef_)

          kf_coef_df['coef'] = coef_kf
          kf_coef_df['R2'] = r2s
          print('The average R2 score is {}'.format(np.average(r2s)))
          print('The average MSE score is {}'.format(np.average(mses)))
          print('The average MAE score is {}'.format(np.average(maes)))

          The average R2 score is 0.8069767205976724
          The average MSE score is 27.246855420494814
          The average MAE score is 3.8480840615681857
```

The average R square score, mean square error and mean absolute error are 0.8069, 27.2468 and 3.8480 respectively. There is no big difference between using KFlod and explicitly train_test_split (slightly better MSE/MAE but worse R2).

And if the standard scaler is implemented to scale independent variables, the prediction based on scaled variables is almost identical to un - scaled prediction. The average R square score, mean square error and mean absolute error are 0.806976, 27.246855 and 3.848084. The MSE and MAE with scaled variables are slightly smaller.

In [319]:
```
# Use KFold with standard scaler
# The result is very close to each other
# kf = KFold(n_splits=fold, shuffle=True, random_state=1)
scaler = StandardScaler()
r2s_s = []
mses_s = []
maes_s = []
coef_kf = []
for train_index, test_index in kf.split(X):
    X_train_kf, X_test_kf = X.values[train_index], X.values[test_index]
    y_train_kf, y_test_kf = y.values[train_index], y.values[test_index]
    X_train_kf_s = scaler.fit_transform(X_train_kf)
    X_test_kf_s = scaler.transform(X_test_kf)
    lg.fit(X_train_kf_s, y_train_kf)
    y_hat_kf = lg.predict(X_test_kf_s)
    r2, mse, mae = check_score(y_test_kf, y_hat_kf)
    r2s_s.append(r2)
    mses_s.append(mse)
    maes_s.append(mae)
print('The average R2 score with standard scaler is {}'.format(np.average(r2s_s)))
print('The average MSE score is {}'.format(np.average(mses_s)))
print('The average MAE score is {}'.format(np.average(maes_s)))
```

```
The average R2 score with standard scaler is 0.8069767205977
The average MSE score is 27.246855420496544
The average MAE score is 3.8480840615705945
```

Another way is to use the cross_val_predict function to perform the cross validation.To simplify the process, the Pipeline object is integrated as the task container to organize and encapsulate standard scaler, polynomial feature and linear regression. Then the Pipeline object cooperates with cross_val_predict to go through the defined tasks and predict by each fold accordingly.
Here I would also set the same folds and test through different polynomial degrees to figure out the best one.

In [327]:
```
# Try Pipeline with Polynomial through cross_val_predict
# kf = KFold(n_splits=6, shuffle=True, random_state=1)
degrees = np.linspace(1, 4, num=4)
for degree in degrees:
    degree = degree.astype(np.int)
    p = PolynomialFeatures(degree=degree)
    estimator = Pipeline([("scaler", scaler),
                          ('polynomial', p),
                          ("regression", lg)])
    y_predict = cross_val_predict(estimator, X, y, cv=kf)
    r2, mse, mae = check_score(y, y_predict)
    model.append('Vanilla linear_cross_val_predict')
    degree_list.append(degree)
    alpha_list.append(0)
    r2_list.append(r2)
    mse_list.append(mse)
    mae_list.append(mae)
    print('The average R2 score with {} degree Polynomial using cross_val_predict with standard scaler is {}'.format(degree,r2))
```

```
The average R2 score with 1 degree Polynomial using cross_val_predict with standard scaler is 0.8107844242256144
The average R2 score with 2 degree Polynomial using cross_val_predict with standard scaler is 0.8189940083495032
The average R2 score with 3 degree Polynomial using cross_val_predict with standard scaler is 0.7717386032419475
The average R2 score with 4 degree Polynomial using cross_val_predict with standard scaler is -0.10349347343313786
```

It is obvious that the degree 2 polynomial feature could bring us the best outcome and the effect is dramatically decreased when the degree is greater than 3. Thus, the R square score, mean square error and mean absolute error are 0.818994, 26.064683 and 3.796733 at degree 2. Also this outcome is better than that generated from non - polynomial linear regression model.

To my interest, I would like to see if it would also get a better outcome if the polynomial feature is adopted in the previous linear regression model:

In [183]:
```
# Use KFold with standard scaler and Polynomial Feature
# The result is very close to cross_val_predict w/ scaler and Polynomial Feature
kf = KFold(n_splits=6, shuffle=True, random_state=1)
# degree = 2 generates the best outcome
p = PolynomialFeatures(degree=2)
scaler = StandardScaler()
r2s_s = []
mses_s = []
maes_s = []
coef_kf = []
for train_index, test_index in kf.split(X):
    X_train_kf, X_test_kf = X.values[train_index], X.values[test_index]
    y_train_kf, y_test_kf = y.values[train_index], y.values[test_index]
    X_train_kf_s = scaler.fit_transform(X_train_kf)
    X_test_kf_s = scaler.transform(X_test_kf)
    X_train_kf_s_p = p.fit_transform(X_train_kf_s)
    lg.fit(X_train_kf_s_p, y_train_kf)
    y_hat_kf = lg.predict(p.transform(X_test_kf_s))
    r2, mse, mae = check_score(y_test_kf, y_hat_kf)
    r2s_s.append(r2)
    mses_s.append(mse)
    maes_s.append(mae)
print('The average R2 score with standard scaler is {}'.format(np.average(r2s_s)))
print('The average MSE score is {}'.format(np.average(mses_s)))
print('The average MAE score is {}'.format(np.average(maes_s)))
```

```
The average R2 score with standard scaler is 0.814838819542642
The average MSE score is 26.06468344141634
The average MAE score is 3.7967327312483747
```

Through the verification listed above, it is clear that with degree 2 polynomial feature, a better prediction outcome is returned.

For now, either train_test_split with 30% test set or cross_val_predict having 6 folds with the polynomial feature can generate the similar prediction outcome.

b. Regularization through Lasso and Ridge

The next step is to adopt model regularization with hyperparameter tuning. I would have Lasso and Ridge models for L1 and L2 regularization respectively and check which one would generate a better outcome. I am using the same cross_val_predict mechanism with the same fold amount here to avoid comparison bias.

```
In [322]: # Hyperparameter tuning
# Try Lasso with Polynomial by cross_val_predict
alphas = np.geomspace(1e-10, 1e0, num=11)
# degree=4 and above will generate negative r2 score
degrees = np.linspace(1, 3, num=3)
for degree in degrees:
    degree = degree.astype(int)

    poly = PolynomialFeatures(degree=degree)
    # alphas
    for alpha in alphas:
#         print('The poly degree is {} and Lasso alpha is {}'.format(degree, alpha))
        lasso = Lasso(alpha=alpha, max_iter=1000000, random_state=1)
        estimator = Pipeline([('scaler', scaler),
                              ('polynomial', poly),
                              ('regression', lasso)])
        y_predict = cross_val_predict(estimator, X, y, cv=kf)
        r2, mse, mae = check_score(y, y_predict)
        model.append('Lasso')
        degree_list.append(degree)
        alpha_list.append(alpha)
        r2_list.append(r2)
        mse_list.append(mse)
        mae_list.append(mae)

        ridge = Ridge(alpha=alpha)
        estimator_ridge = Pipeline([('scaler', scaler),
                              ('polynomial', poly),
                              ('regression', ridge)])
        y_predict_ridge = cross_val_predict(estimator_ridge, X, y, cv=kf)
        r2_ridge, mse_ridge, mae_ridge = check_score(y, y_predict_ridge)
        model.append('Ridge')
        degree_list.append(degree)
        alpha_list.append(alpha)
        r2_list.append(r2_ridge)
        mse_list.append(mse_ridge)
        mae_list.append(mae_ridge)
df1['Model'] = model
df1['Polynomial Degree'] = degree_list
df1['Alpha'] = alpha_list
df1['R2'] = r2_list
df1['MSE'] = mse_list
df1['MAE'] = mae_list
df1.sort_values(by=['R2'], ascending=False)
```

By looping the same alpha and degree values, Lasso and Ridge models are executed through pipeline and cross_val_predict. The execution outcome is as following:

Out[385]:

	Model	Polynomial Degree	Alpha	R2	MSE	MAE
45	Lasso	2	1.000000e-01	0.822517	25.557446	3.757648
43	Lasso	2	1.000000e-02	0.820106	25.904525	3.786644
41	Lasso	2	1.000000e-03	0.819121	26.046374	3.795319
48	Ridge	2	1.000000e+00	0.819058	26.055469	3.796518
39	Lasso	2	1.000000e-04	0.819007	26.062772	3.796592
...	...	...	...	...	...	...
56	Ridge	3	1.000000e-07	0.771739	32.869415	4.196827
54	Ridge	3	1.000000e-08	0.771739	32.869415	4.196827
52	Ridge	3	1.000000e-09	0.771739	32.869415	4.196827
50	Ridge	3	1.000000e-10	0.771739	32.869415	4.196827
69	Lasso	3	1.000000e+00	0.762432	34.209545	4.398540

66 rows × 6 columns

The best prediction outcome is generated by Lasso with alpha value 0.1 and degree 2 polynomial feature. The R square score, mean square error and mean absolute error are 0.822517, 25.557446 and 3.757648.

4. Model selection

I will set the threshold by setting the R square score greater than 0.818, which is the best score generated by vanilla linear regression and see which model and parameter combination can outperform it.

```
In [405]: df1.loc[(df1['Model']=='Vanilla linear_train_test_split')
                  |(df1['Model']=='Vanilla linear_cross_val_predict')
                  |((df1['Model']=='Lasso')&(df1['R2']>0.818))
                  |((df1['Model']=='Ridge')&(df1['R2']>0.818))].sort_values(by=['R2'], ascending=False)
```

Out[405]:

	Model	Polynomial Degree	Alpha	R2	MSE	MAE
45	Lasso	2	1.000000e-01	0.822517	25.557446	3.757648
43	Lasso	2	1.000000e-02	0.820106	25.904525	3.786644
41	Lasso	2	1.000000e-03	0.819121	26.046374	3.795319
48	Ridge	2	1.000000e+00	0.819058	26.055469	3.796518
39	Lasso	2	1.000000e-04	0.819007	26.062772	3.796592
46	Ridge	2	1.000000e-01	0.819001	26.063705	3.796704
37	Lasso	2	1.000000e-05	0.818995	26.064493	3.796719
44	Ridge	2	1.000000e-02	0.818995	26.064585	3.796730
35	Lasso	2	1.000000e-06	0.818994	26.064664	3.796731
42	Ridge	2	1.000000e-03	0.818994	26.064674	3.796732
33	Lasso	2	1.000000e-07	0.818994	26.064682	3.796733
40	Ridge	2	1.000000e-04	0.818994	26.064682	3.796733
31	Lasso	2	1.000000e-08	0.818994	26.064683	3.796733
38	Ridge	2	1.000000e-05	0.818994	26.064683	3.796733
29	Lasso	2	1.000000e-09	0.818994	26.064683	3.796733
36	Ridge	2	1.000000e-06	0.818994	26.064683	3.796733
27	Lasso	2	1.000000e-10	0.818994	26.064683	3.796733
34	Ridge	2	1.000000e-07	0.818994	26.064683	3.796733
32	Ridge	2	1.000000e-08	0.818994	26.064683	3.796733
30	Ridge	2	1.000000e-09	0.818994	26.064683	3.796733
28	Ridge	2	1.000000e-10	0.818994	26.064683	3.796733
2	Vanilla linear_cross_val_predict	2	0.000000e+00	0.818994	26.064683	3.796733
0	Vanilla linear_train_test_split	0	0.000000e+00	0.818006	27.610861	3.850614
1	Vanilla linear_cross_val_predict	1	0.000000e+00	0.810784	27.246855	3.848084
3	Vanilla linear_cross_val_predict	3	0.000000e+00	0.771739	32.869415	4.196827
4	Vanilla linear_cross_val_predict	4	0.000000e+00	-0.103493	158.901967	7.253268

With model regularization and hyperparameter tuning it seems the Lasso model with alpha value 0.1 and degree 2 polynomial feature is the choice, since it provides the highest R square score and also the lowest MSE and MAE. Actually, it is interesting that with degree 2 polynomial feature, both Lasso and Ridge can perform better at different alpha levels than vanilla linear regression.

Key Findings and Insights, which walks your reader through the main drivers of your model and insights from your data derived from your linear regression model.

As described in the previous section, the initiative of this analysis is to find out the relationship between team overall performance with payroll and team winnings. Therefore, through the key offensive, pitching and defensive variables, the prediction model is trained and tested.

The original main dataset consists of detailed performance statistics. By studying and analyzing these columns we categorize the columns into 3 types (offensive, pitching and defensive) and then extract key variables from them to form the independent variables. Since the calculated OPS is used to better describe how a player performs on the offensive side, I can drop the elements that contribute to OPS from the model variable list.

In general, during variable selection, when testing R square scores with different variable sets, the observation is the R square score decreases when the variable numbers decrease. Theoretically the higher R square is preferred, however it could also be a sign of overfitting and high complexity.

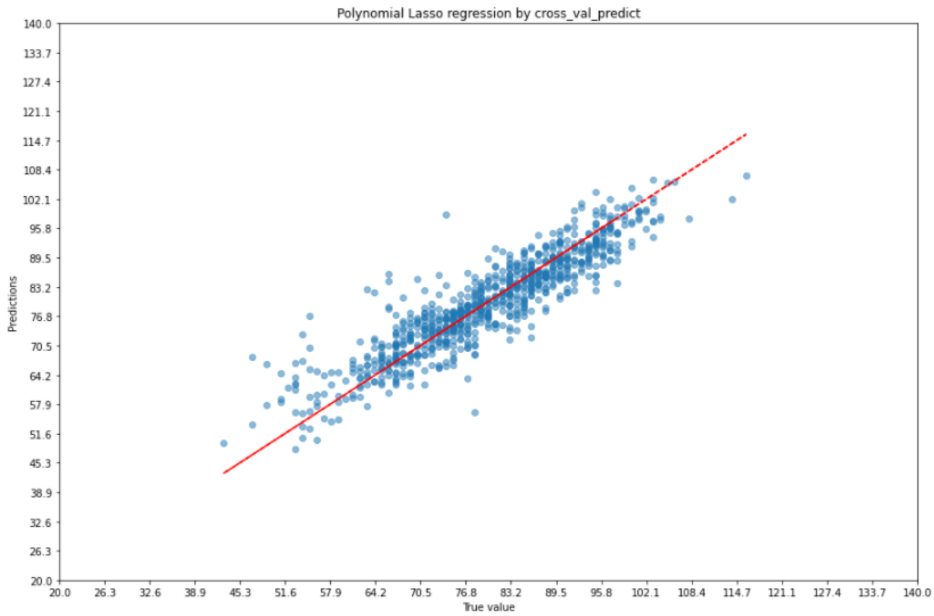
As a result, to balance the complexity and accuracy, there are 7 variables out of 47 chosen to be placed in the model based on the following reasons:

1. Multicollinearity and overfitting caused by too many variables.
2. Simplicity and better variable explainality.

While validating the various linear regression models, I have found that both Lasso and Ridge model can provide better R square scores than the vanilla linear regression model with hyperparameter tuning. The best one based on the current main data set is the Lasso model with degree 2 polynomial feature and alpha value 0.1.

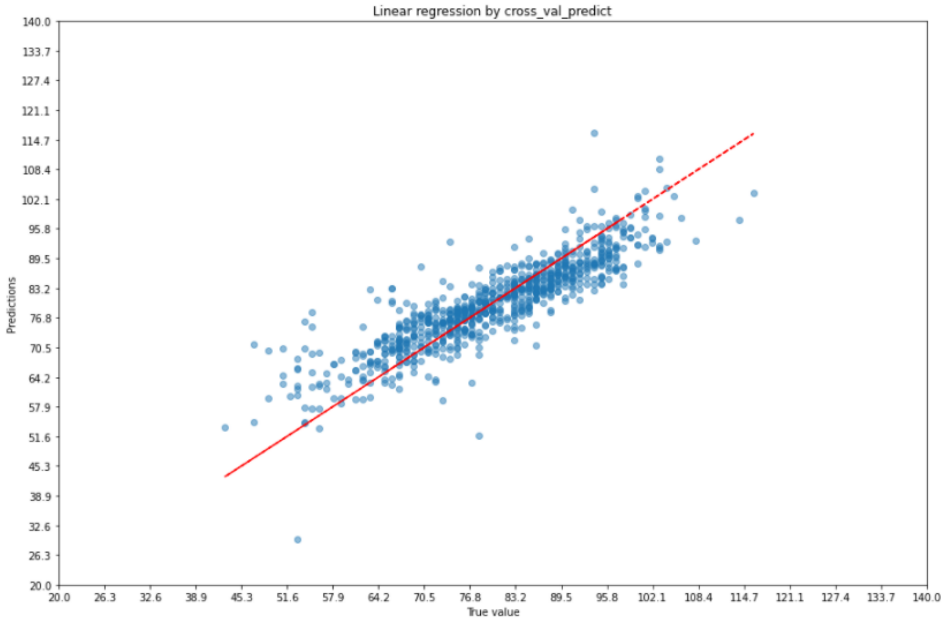
Here is the visualization of the selected Lasso model to plot real data and predicted data:

In [406]:
```
# With hyperparameter tuning it seems Lasso model with alpha value 0.1 and 2 degree polynomial is the choice
# plot our best Lasso
poly = PolynomialFeatures(degree=2)
lasso_ = Lasso(alpha=0.1, max_iter=1000000, random_state=1)
estimator = Pipeline([('scaler', scaler),
                      ('polynomial', poly),
                      ('regression', lasso_)])
y_predict_lasso = cross_val_predict(estimator, X, y, cv=kf)
r2, mse, mae = check_score(y, y_predict_lasso)
get_scatter(y, y_predict_lasso, title='Polynomial Lasso regression by cross_val_predict')
```



And in contrast, let’s visualize the prediction results from linear regression drove by cross_val_predict with degree 2 polynomial feature:

```
In [407]: # plot linear regression (cross_val_predict) with degree 2 Polynomial
          get_scatter(y, y_predict, title='Linear regression by cross_val_predict')
```



Through the effort of regularization, the Lasso model has more convergent predictions than the linear regression model.

Also, when comparing both models through complexity, Lasso also reduces the model weights (coefficients) as well.

Here is the vanilla linear regression model coefficient data through degree 1 to 4 polynomial feature:

In [650]:

```python
# Try Pipeline with Polynomial through cross_val_predict
# kf = KFold(n_splits=6, shuffle=True, random_state=1)
degrees = np.linspace(1, 4, num=4)
coef_df = pd.DataFrame()
for degree in degrees:
    degree = degree.astype(np.int)
    p = PolynomialFeatures(degree=degree)
    estimator = Pipeline([("scaler", scaler),
                          ('polynomial', p),
                          ("regression", lg)])
    y_predict = cross_val_predict(estimator, X, y, cv=kf)
    r2, mse, mae = check_score(y, y_predict)
    model.append('Vanilla linear_cross_val_predict')
    degree_list.append(degree)
    alpha_list.append(0)
    r2_list.append(r2)
    mse_list.append(mse)
    mae_list.append(mae)
    print('The average R2 score with {} degree Polynomial using cross_val_predict with standard scaler is {}'.format(degree,r2))
    intercepts, coefs = check_coef(estimator, X, y, cv=kf)
    df2 = get_coef_info(intercepts, coefs)
    coef_df = coef_df.append(df2)
coef_df
```

```
The average R2 score with 1 degree Polynomial using cross_val_predict with standard scaler is 0.8107844242256144
The average R2 score with 2 degree Polynomial using cross_val_predict with standard scaler is 0.8189940083495032
The average R2 score with 3 degree Polynomial using cross_val_predict with standard scaler is 0.7717386032419475
The average R2 score with 4 degree Polynomial using cross_val_predict with standard scaler is -0.10349347343313697
```

Out[650]:

	Intercept	Coef_Sum	Coef_Amount	Coef_Not_Zero
0	80.085714	20.162446	8	7
1	79.324812	20.661110	8	7
2	80.091729	20.685877	8	7
3	79.831579	20.195613	8	7
4	79.822556	20.731976	8	7
5	79.572932	20.569739	8	7
0	80.157839	28.134017	36	36
1	79.338715	28.459091	36	36
2	79.937691	29.556901	36	36
3	79.472956	28.714114	36	36
4	79.641898	27.534470	36	36
5	79.241028	28.068870	36	36
0	79.355590	62.095036	120	120
1	78.372392	65.185076	120	120
2	79.709471	73.966530	120	120
3	78.654418	70.750935	120	120
4	79.116971	66.059799	120	120
5	78.566166	65.379648	120	120
0	79.693164	319.102129	330	330
1	78.286920	308.762498	330	330
2	79.609727	421.171196	330	330
3	79.031212	361.696255	330	330
4	78.870411	330.498931	330	330
5	78.723575	312.928058	330	330

The Ridge model coefficient data with alpha 0.1 at degree 2 polynomial feature:

In [646]:

```python
# Get Ridge coefficient
poly = PolynomialFeatures(degree=2)
ridge_ = Ridge(alpha=0.1)
estimator1 = Pipeline([('scaler', scaler),
                       ('polynomial', poly),
                       ('regression', ridge_)])
intercepts, coefs = check_coef(estimator1, X, y, cv=kf)
df4 = get_coef_info(intercepts, coefs)
df4
```

Out[646]:

	Intercept	Coef_Sum	Coef_Amount	Coef_Not_Zero
0	80.157477	28.125415	36	35
1	79.338309	28.449944	36	35
2	79.937564	29.546024	36	35
3	79.472594	28.703211	36	35
4	79.641568	27.526430	36	35
5	79.240735	28.059058	36	35

Finally, the Lasso model coefficient data with alpha 0.1 at degree 2 polynomial feature:

In [643]:
```
# Get Lasso coefficient
poly = PolynomialFeatures(degree=2)
lasso_ = Lasso(alpha=0.1, max_iter=1000000, random_state=1)
estimator1 = Pipeline([('scaler', scaler),
                       ('polynomial', poly),
                       ('regression', lasso_)])

intercepts, coefs = check_coef(estimator1, X, y, cv=kf)
df3 = get_coef_info(intercepts, coefs)
df3
```

Out[643]:

	Intercept	Coef_Sum	Coef_Amount	Coef_Not_Zero
0	79.962539	23.109046	36	21
1	79.127834	23.515627	36	24
2	79.869961	24.375593	36	23
3	79.480512	23.662302	36	25
4	79.587913	23.682548	36	21
5	79.174824	23.137316	36	19

It shows that the Lasso model contributes less weights and the least model complexity through feature selection.

As an additional observation, I also check the prediction quality of ElasticNet regression:

In [656]:
```
poly = PolynomialFeatures(degree=2)
elastic = ElasticNet(alpha=0.1)
estimator2 = Pipeline([('scaler', scaler),
                       ('polynomial', poly),
                       ('regression', elastic)])
y_predict2 = cross_val_predict(estimator2, X, y, cv=kf)
r2, mse, mae = check_score(y, y_predict2)
print('Elastic regression by cross_val_predict')
print('The average R2 score is {}'.format(r2))
print('The average MSE score is {}'.format(mse))
print('The average MAE score is {}'.format(mae))
intercepts, coefs = check_coef(estimator2, X, y, cv=kf)
df5 = get_coef_info(intercepts, coefs)
df5
```

```
Elastic regression by cross_val_predict
The average R2 score is 0.8191768083201039
The average MSE score is 26.03836042678374
The average MAE score is 3.82661512086831
```

Out[656]:

	Intercept	Coef_Sum	Coef_Amount	Coef_Not_Zero
0	79.975488	23.024210	36	28
1	79.098546	23.427947	36	28
2	79.878424	24.413111	36	29
3	79.396107	23.755660	36	29
4	79.537852	23.396548	36	24
5	79.146400	23.095621	36	27

As expected the ElasticNet approach provides the combination of the L1 and L2 regularization.

Suggestions for next steps

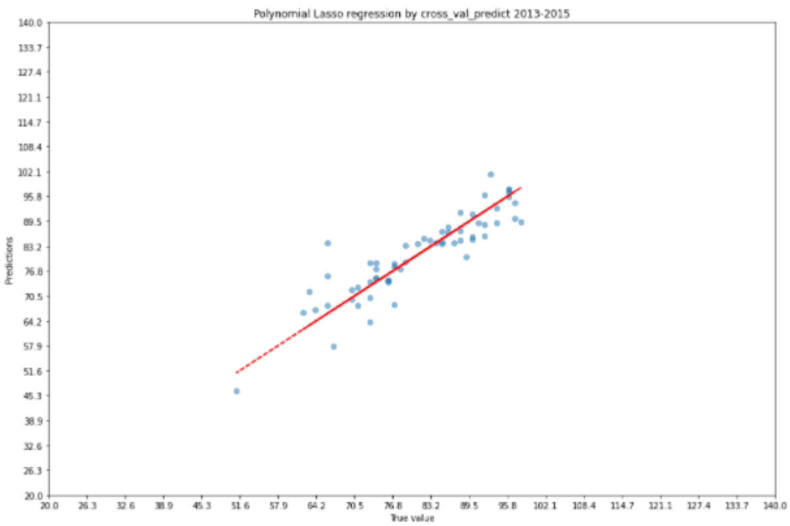
So far based on the main data set, the Lasso model seems to be working well in terms of accuracy and complexity.
Since this model is based on the main data set that accumulated from 1985 to 2012 baseball seasons, it is interesting to see if the model is still efficient and outperforms the vanilla linear regression model.

Here is the sample result by using the same model and running with the data set on different time intervals, for example from 2013 to 2015.

```
In [973]: # With hyperparameter tuning it seems Lasso model with alpha value 0.1 and 2 degree polynomial is the choice
          # plot our best Lasso
          poly = PolynomialFeatures(degree=2)
          lasso_ = Lasso(alpha=0.1, max_iter=1000000, random_state=1)
          estimator = Pipeline([('scaler', scaler),
                                ('polynomial', poly),
                                ('regression', lasso_)])
          y_predict_lasso = cross_val_predict(estimator, X, y, cv=kf)
          r2, mse, mae = check_score(y, y_predict_lasso)
          print('The average R2 score with standard scaler is {}'.format(r2))
          print('The average MSE score is {}'.format(mse))
          print('The average MAE score is {}'.format(mae))
          get_scatter(y, y_predict_lasso, title='Polynomial Lasso regression by cross_val_predict 1985-2012')

          # for post check on smaller data set
          y_predict_lasso_post = cross_val_predict(estimator, X1, y1, cv=kf)
          r2, mse, mae = check_score(y1, y_predict_lasso_post)
          print('The average R2 score with standard scaler is {}'.format(r2))
          print('The average MSE score is {}'.format(mse))
          print('The average MAE score is {}'.format(mae))
          get_scatter(y1, y_predict_lasso_post, title='Polynomial Lasso regression by cross_val_predict 2013-2015')
```

```
The average R2 score with standard scaler is 0.8225165129075208
The average MSE score is 25.557446275461334
The average MAE score is 3.7576475702764456
The average R2 score with standard scaler is 0.7993742317629622
The average MSE score is 23.496565493981016
The average MAE score is 3.584726885157677
```



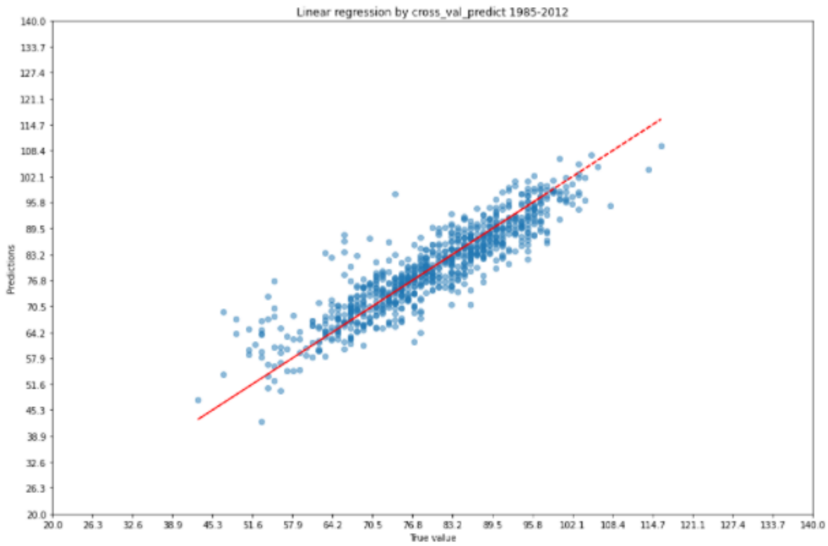
For now it still seems efficient and reasonable.

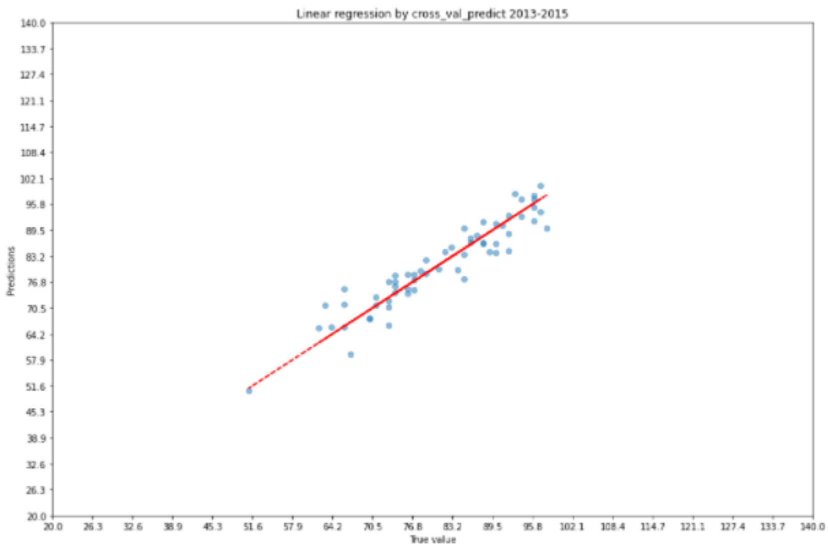
As a comparison, I would like to take vanilla linear regression to predict the same data set again, but no polynomial feature effect (degree = 1):

```
n [972]: # for post check on smaller data set
p = PolynomialFeatures(degree=1)
estimator = Pipeline([("scaler", scaler),
                      ('polynomial', p),
                      ("regression", lg)])
y_predict = cross_val_predict(estimator, X, y, cv=kf)
r2, mse, mae = check_score(y, y_predict)
print('Linear regression by cross_val_predict')
print('The average R2 score is {}'.format(r2))
print('The average MSE score is {}'.format(mse))
print('The average MAE score is {}'.format(mae))
get_scatter(y, y_predict, title='Linear regression by cross_val_predict 1985-2012')

y_predict1 = cross_val_predict(estimator, X1, y1, cv=kf)
r2, mse, mae = check_score(y1, y_predict1)
print('For post check on Linear regression by cross_val_predict')
print('The average R2 score is {}'.format(r2))
print('The average MSE score is {}'.format(mse))
print('The average MAE score is {}'.format(mae))
get_scatter(y1, y_predict1, title='Linear regression by cross_val_predict 2013-2015')
```

```
Linear regression by cross_val_predict
The average R2 score is 0.8107844242256144
The average MSE score is 27.246855420496544
The average MAE score is 3.8480840615705945
For post check on Linear regression by cross_val_predict
The average R2 score is 0.8833940469810209
The average MSE score is 13.656468140530267
The average MAE score is 2.884856511413741
```





Interestingly, it outperforms the Lasso model on R square score, MSE and MAE.

So, it would be the next topic to analyze if the data set size matters when selecting the best model and whether the polynomial feature should be applied on the variables. Would it be better off to apply a different model by different data set size, or come up with a generally accepted one? For example, is it better to adopt vanilla linear regression to predict the current season winnings based on in - season statistics? It would be a question that is more related to the business initiative and also to the question that we are going to solve.

In addition, adding and validating other columns such as attendees into independent variables can be considered as well.