Winning is the Name of the Game

Predicting Individual NBA Player Win Share with Machine Learning

By Joseph Earnshaw

The Proposal - So what's the question?

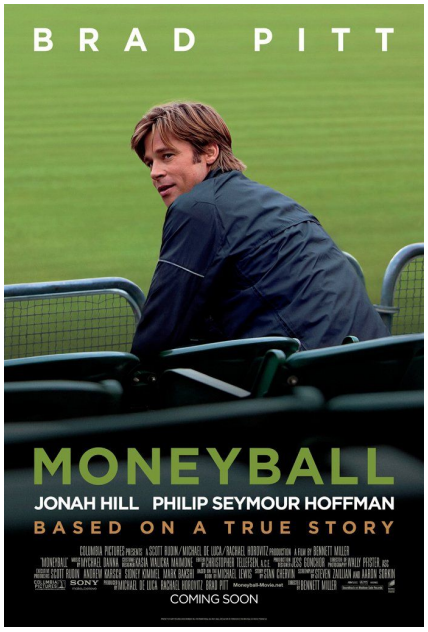


Image via Amazon

The original motivation for this project was trying to predict if the Sacramento Kings were going to make the playoffs this year. So far, I've found some very interesting data and insights with regards to both team-level and individual player-level data that could potentially help answer this question. However, I've decided to pivot in a new direction, namely to focus on predicting an individual player's win share.

There are a few reasons for this:

- Firstly, as of today (March 29, 2019), the Sacramento Kings are in ninth place in the NBA's Western Conference. This normally wouldn't be a bad position to be in late in the season, however, the San Antonio Spurs, who are currently in eighth place are six and a half games ahead of them with only seven games left in the regular season.

  What does this mean? That it is virtually impossible for the Kings to make the playoffs this year (FiveThirtyEight has the odds at <1%). While I could still theoretically continue in this direction, I think the value it would provide would be minimal at best.

- Secondly, I'm a big fan of the movie *Moneyball*, which is about how the Oakland A's used analytics to compete with (and beat) teams like the Red Sox and Yankees despite having one of the lowest payrolls in all of baseball. As I was doing my exploratory data analysis, particularly when it came to data on individual players, I kept asking myself: which players are diamonds in the rough, so to speak?

We all know players like LeBron James, Kevin Durant, and Anthony Davis, who are well-known superstars who stats back up the fact that they are for the most part the primary contributors to their teams' success. There are so many other players in the league though and I became particularly interested in trying discovering the

players that may not be particularly popular with the fans but are (or at least should be) very popular with the front office personnel of NBA teams. This may be of particular value considering that for teams that don't make the playoffs, they will look towards finding players in the offseason via free agency to bolster their roster with the hopes of making the playoffs in 2019-2020.

So in summary, I am pivoting the focus of my project. Initially, I was trying to answer the following question: 'will the Sacramento Kings make the playoffs this season?'. Due to a change in circumstances and perspective, I will now focus on trying to answer the following question: can I predict an individual player's win share? By answering this question, I hope to discover deeper insights into how much player's and their individual skill sets contribute to a team's win total.

The Data 'Rodeo' - Wrangling the data

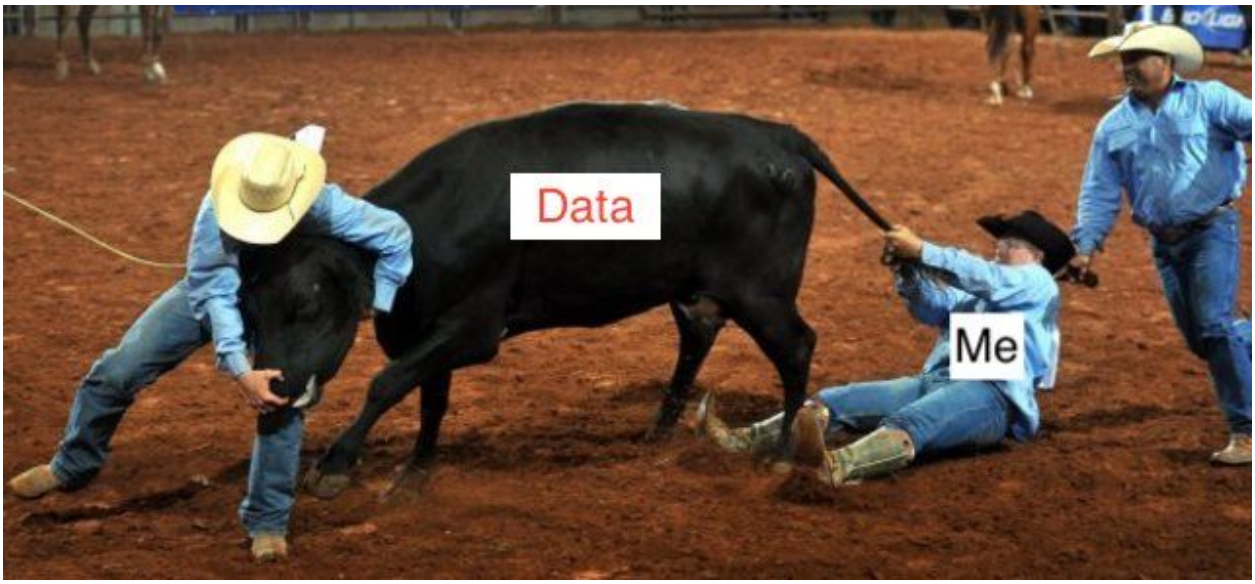


Image by Times Record News Wichita Falls

This project originally got started with one rather simple question: will the Sacramento Kings make the playoffs in 2019? However, since writing the project proposal, the Kings have gone from a record of 25 wins and 25 losses to a record of 40 wins and 42 losses. This record puts them in 9th place in the Western Conference, 6.5 games back of the 8th place San Antonio Spurs. With only two games left it is impossible for them to make the playoffs this year. But, hey there is always next year! While I was exploring the individual player data though, another question came to mind: can we predict how good a player is and how much he contributes to a team winning?

So we have a question we want to try and answer, what is the next step! Data wrangling! And in this case, no we won't be herding livestock. Instead, we're going to be wrangling NBA data from the wild, wild web! With certain fields, getting data that is useful can be the most difficult part of the project, if you can even find it at all. However, the NBA has seen a boon in the last few years when it comes to analytics. The Golden State Warriors have risen from an also-ran to perennial NBA title contender, and they credited much of this success to their use of analytics (check out this article). The rest of the NBA has followed suit, with nearly every team having some sort of data analytics department. This has not only helped teams on the court, but it's also been a boon to fans too particularly those who are analytically inclined.

Side note: I find it interesting that the two teams that started the 'analytics revolution' in their respective sports are both located in Oakland (i.e. the Oakland Athletics for MLB and the Golden State Warriors for the NBA).

After the excitement about finally picking a question I wanted to answer for this project, I was hit with the realization of how do I get the data to even start this? Luckily, this question was quickly answered. I follow Nate Silver's blog FiveThirtyEight and in addition to analysis on politics, culture, and economics, the blog has a section completely devoted to the NBA. Namely, it predicts a teams chance of making the playoffs which are updated after every game and depth chart revision. After reviewing the details of how FiveThirtyEight NBA predictions worked, I noticed one of their references was basketball-reference.com so I decided to check it out.

What I found amounted to the 'El Dorado' of basketball statistics. There was data on individual players and teams dating back to the late 1940's! For the most recent seasons, they had data sets on advanced statistics like Player Efficiency Rating (measures per-minute production), Win Shares (estimate of the number of wins contributed by a player) and Box +/- (estimate of the points per 100 possessions a player contributed above a league-average player). I'd found my treasure chest, now it was time to gather the data and get it ready for analysis!

With this in mind, I decided to use individual player data that combined statistics like points, assists, and rebounds (just to name a few) on a per 100 possession basis and advanced statistics like true shooting percentage and win shares (which I'll get to a little later). For clarification, why did I decide to use per 100 possession versus per game? In one word: pace. Teams play the game at different speeds, with some being more oriented towards the fast-break and others geared more towards slowing things down and running plays on the offensive end. As a result, teams that play faster will have more possessions per game on average than slower teams which has a direct impact on player stats. Per 100 possession statistics eliminate this discrepancy by showing on average how many points (or rebound, steals, etc.) he score per 100 possessions.

Luckily, the data wrangling process was not as hard as I was expecting. Using the urllib package with BeautifulSoup I was able to create a function that scraped data from basketball-reference.com and returned the raw data in a pandas DataFrame. Here is a step-by-step of how the function worked:

1. Create variable that stores URL we'll be scraping
2. Create variable -- html -- that opens the url
3. BeautifulSoup then passes through the website and returns it as an object
4. Uses list comprehension to find column headers and rows (i.e. player statistics), stores them in variables called 'header' and 'player_stats', respectively
5. Use pandas.DataFrame with the variable 'player_stats' as the data and 'headers' for the columns argument
6. Adds a column called 'Year' (mainly for organization purposes)
7. Returns the DataFrame

Using this function I was able to scrape per 100 possession and advanced statistics from basketball-reference.com for the past ten seasons (which did not include 2018-2019 as it is not complete yet). When I originally gathered data, I had to copy and paste into a text editor and then save it as a CSV file, which was quite tedious and was not particularly fun. This process sped up that process significantly, as gathering the data took seconds.

However, the raw data was not clean and there were a few steps I needed to take in order to make it usable for analysis. Luckily with a little experimentation, I was able to create two functions -- clean_per_poss &

clean_advanced -- that returned cleaned up pandas DataFrames ready for analysis. Here is the general overview of what they did:

1. Dropped duplicate players
2. Replaced ‘None’ rows with ‘NaN’
3. Dropped rows with ‘NaN’
4. Dropped blank column(s) -- the advanced stats DataFrame returned two blank columns while the per-possession one only had one
5. Converted variables to numerics
6. Reset the index

While it took some time up front to tinker and create these functions, it helped cut down the clean-up time significantly in the long run. Once they were created, all that we needed to do was pass the raw DataFrame into the function and a second later a clean DataFrame was passed out! Then we merged the two data sets (per 100 possessions and advanced stats) together and voila, we had data for every player that played in the NBA for that particular season!

At the very end, we concatenated each seasons DataFrame together to return a data set that combined all ten seasons into one DataFrame. After checking and addressing null values -- which essentially was just replacing the null values with 0 because most of the observations had not taken a 3-point shot for example -- we had a clean and usable data set! Next step: exploratory data analysis!

Discovering a New World - Exploratory Data Analysis of NBA Players

When I first began this project, I was initially focused on whether or not the Sacramento Kings would make the playoffs. I pulled team statistics from the past 5 seasons and individual player statistics from the past 10 seasons and began some initial exploration.

From the initial EDA of the team stats, there was a clear pattern amongst many of the variables indicating which teams made the playoffs from those that didn’t. However, I was particularly fascinated by the individual player stats.

This was for a few reasons. Firstly, while basketball is a team sport, the talent level of each individual player directly contributes to that particular team's ability to win. Basically, the more talented your players, the higher the likelihood you are going to win games.

Secondly, looking at this from a team perspective is similar to looking at the economy from a macro perspective. You are able to assess a general direction but aren’t really able to capture the small nuances that cause the economy to go that particular direction. The more I explored, the more I became interested in trying to assess an individual player’s ability to contribute.

My initial hypothesis was that by examining each part of a player’s game, we could highlight the skills that contribute to arguably the most important aspect of professional basketball: winning. Then I started asking myself questions like ‘does a lockdown defender have a bigger impact on winning than a ball-handler?’. Or ‘does being a better offensive player play a more significant impact than a defensive-oriented player?’.

Lastly, I had a lot more data to work with. For the team data, I could gather hundreds of observations while for individual players I could gather thousands, depending on how many seasons back I wanted to go. It seemed like a no-brainer and is why I decided to go the individual-player route.

When it comes to my exploratory data analysis, my focus was primarily on visualizing the forty or so potential explanatory variables and their relationship with our target variable, win shares per 48 minutes. Why did I choose this particular metric? Well for one thing 48 minutes is equal to 1 NBA game. We could've gone down the more macro route of trying to predict overall win shares (for the whole season) but I didn't think this would provide the same value as what a player did on a game-by-game basis. Additionally, I planned on comparing this number — i.e. win shares per 48 minutes — with the salary to generate a unique perspective on the value a player provided to a team compared to their cost (i.e. their salary).

One of the first interesting aspects of the data was in regards to the number of games played in. It was significantly left-skewed, with a sharp uptick around the 60 game mark. This potentially signaled that while teams have between 13-15 players on their active roster on any given night, that only a portion of these players actually get playing time.

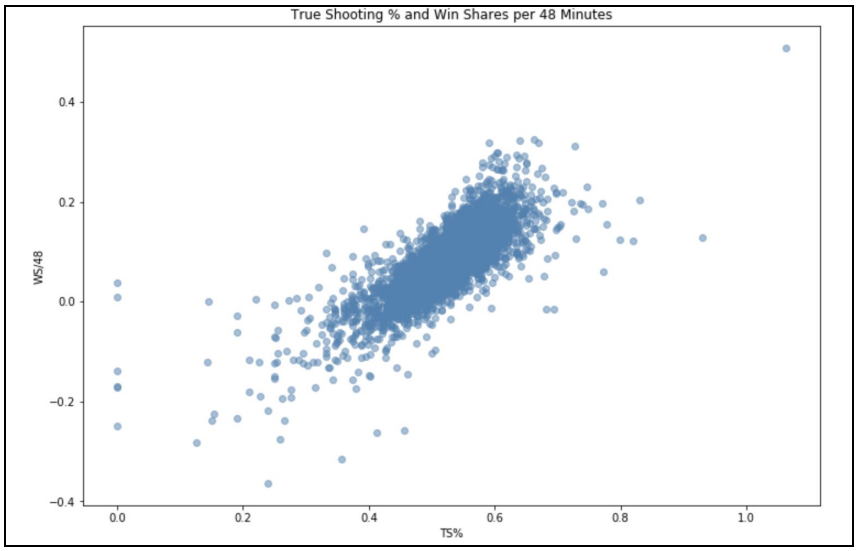
When we took a look at the number of games started, there was further evidence supporting the previously stated 'portion of players' hypothesis. The overwhelming majority of players started exactly zero games! From there, the histogram quickly descends with a relatively small number of players starting between 20 and 70 games. At 82 games, there is a spike which gives us some evidence that teams could be using a core group of starters, leading to only a portion of players playing in any given game.

Now when creating the first scatter plots — focusing on win shares per 48 minutes and minutes played, field goals, and 3-point field goals — there was a significant distortion of the graphs due to outliers. To address this, we used the PER statistic which stands for player efficiency rating.

Now why use PER for this? Firstly, because it is somewhat similar to win shares in that it uses both offensive and defensive statistic to give a holistic picture of a player's overall skill set. Secondly, it had a statistical cut-off in that players had to have played 6.09 MPG (minutes per game). After instituting the PER cut-off the distortion was for the most part eliminated. I want to note however that I didn't institute the second part of the cut-off which was that the player had to have played in more than 500 minutes total that season as well. The primary reason for this was due to the fact that it would have eliminated approximately 46% of my data set if I had and I did not want to eliminate so much data for what at best would be a marginal return.

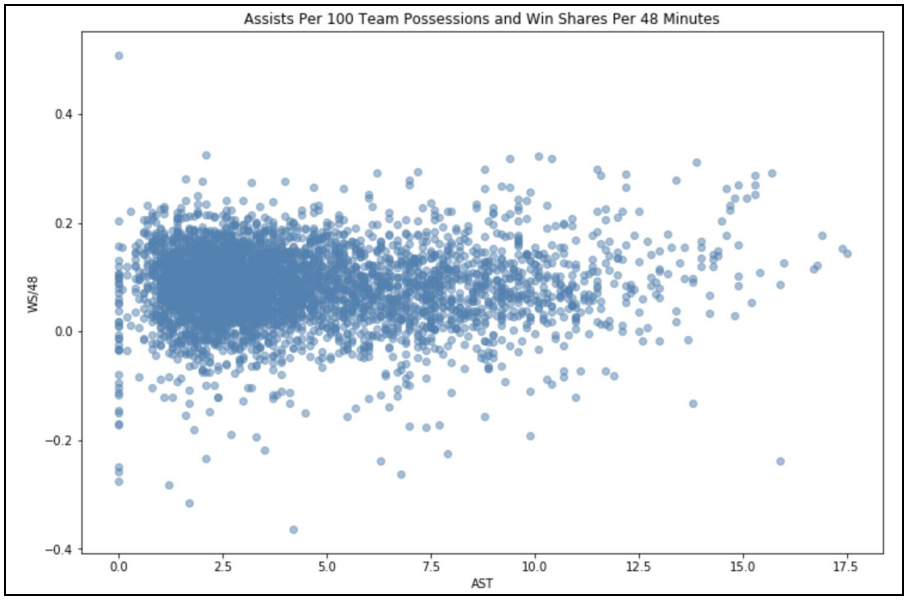
That being said there were some interesting takeaways from the exploratory data analysis with some being expected and others being rather unexpected/disappointing.

The first, and perhaps the most obvious was the correlation between shooting and win share. Field goals made, field goal percentage and true shooting percentage — which essentially combines field goal percentage, free throw percentage, and three-point percentage into one — all appeared to have a strong positive impact on win share. Basically, the more you shoot and the more shots you make, the higher your likely contribution is going to be towards your team winning.



Scatter plot showing the strong relationship between true shooting % and win share per 48 minutes

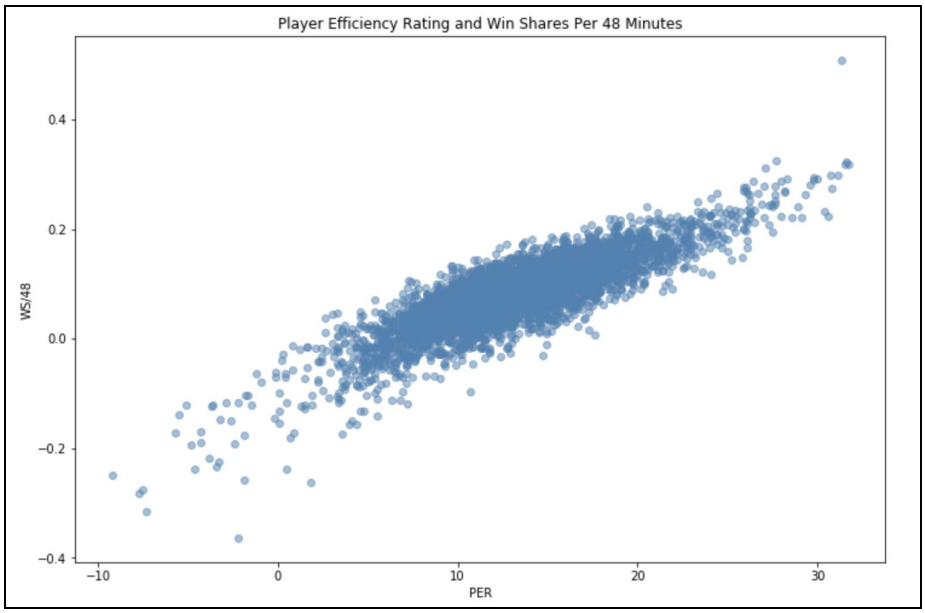
The second takeaway was that assists and rebounds don't appear to have any significant relationship with a player's win share. This was rather surprising since outside of scoring these are perhaps the two most common statistics used to describe how 'good' a player is. Additionally, it is generally assumed that better teamwork generally equals more wins but we're not seeing that to be the case at least with these particular graphs. With regards to rebounding, if you squint just right there may appear to be a slight positive correlation with win share but overall I'd say their relationship is ambiguous at best.



Assists and win shares, not too much to see here…

The third takeaway was that offense is king. Most of the variables associated with defense — like steals and blocks — returned as indiscernible blobs with no relationship whatsoever with win share. After seeing the relationships with offensive variables and shooting in particular, being a good offensive player who can shoot the ball well looks to have more of an impact on winning than being a good rebounder who can block shots.

In a way this makes sense. A hypothesis I have is that while grabbing steals or blocking shots may lead to a team gaining additional possessions, the rarity of these respective statistics (averages usually hover around two per 100 possessions for both) cancels out any positive impact they may have.



Don’t just see, observe: despite a strong relationship with WS/48, have to be careful using variables like PER because they could induce multicollinearity

The fourth and perhaps most important takeaway is the potential for multicollinearity between variables. While I didn’t technically test for this, there are quite a few variables that interact with each other. A prime example of this would be the three variables associated with FG — field goals made, field goals attempted and field goal percentage. The three have a very intricate relationship, with making more field goals (or shooting more) having a direct impact on field goal percentage.

There are a few that may not be as obvious, especially to those unfamiliar with basketball statistics. One is PER — player efficiency rating — which is calculated by combining numerous offensive and defensive variables together. There are weights given to each but we’ll have to be careful when it comes time to determine which features to use for our machine learning algorithms.

All About the ‘W’ - In-depth ML Analysis & Model Creation

LINEAR REGRESSION: AN OVERVIEW

For this project we are trying to determine a player's WS/48, i.e., how many 'wins' he contributes for every 48 minutes he plays. While statistics like total points, rebounds and assists are all vital statistics when evaluating any given player, but there is something that trumps all other statistics in basketball (and pretty much all sports): winning.

The statistics mentioned above only describe a particular segment of that player's contribution to his team whereas WS/48 (win shares per 48 minutes) attempts to combine all these into one statistic. I'm going to admit up front that it is not perfect. However, it's like the saying: "All models are wrong, but some are useful." In this case, WS/48 is a useful 'model' in bringing together players offensive and defensive statistics into a single number that allows us to see his level of contribution towards a team winning.

Now I want to recap what we've done so far up to this point for this project:

- Pulled Per Possession and Advanced statistics from basketball-reference.com for the past ten seasons (not including the 2018-2019 season)
- Cleaned up raw data into a usable format
- Performed Exploratory Data Analysis of features of the clean data set

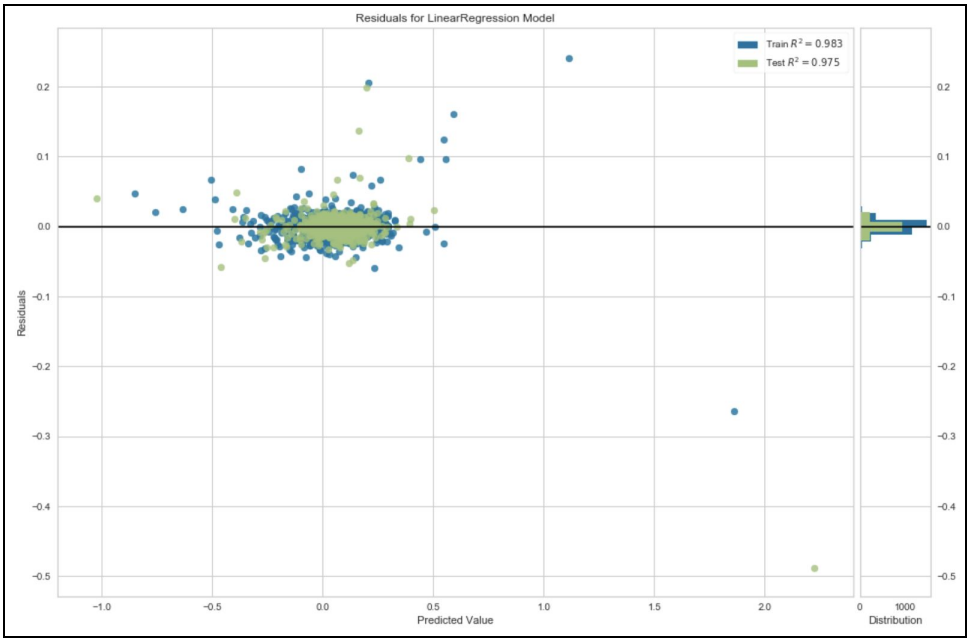
Now we are on the fourth and final step: use machine learning to create a model that can approximately predict a player's WS/48 statistic.

How are we going to do this? Two words: linear regression. That's where we'll begin, with scikit-learn's linear regression algorithm.

Linear regression attempts to model the relationship between a dependent variable and one or more independent variables. If you are using one independent variable, i.e., a one-to-one relationship between the independent and dependent variable, it is called simple linear regression. You are not limited to just one though; you can use more than one independent variable, which is called multiple linear regression. In summary:

- Simple = 1 independent variable
- Multiple = 2+ independent variables

Now without further ado, let's dive in!



Residual Plot For Linear Regression Model with all variables

ASSESSMENT: MULTIPLE LINEAR REGRESSION

The performance of the model looks pretty amazing! The first statistic, the R squared, basically indicates how close the data is to the fitted regression line. The scale for this number is between 0 and 1.0, 0 indicating that the model does not explain any of the variability of the response variable (it would basically just be a random assortment of points) and 1 indicating a 'perfect' model in that the model explains all of the variability of the response variable. With our R^2 statistic being ~0.96, this model looks nearly perfect!

Next, we'll take a look at the RMSE, or root mean square error. This metric also indicates how close the observed data points are to the model's predicted values and is extremely useful in that it presents results in the same units as the response variable, i.e., WS/48. Here our RMSE is ~0.0177 telling us that on average our predictions were this far off from the observed value.

Lastly, let's look at the residual plot. For the most part, it looks good; the residuals (i.e., the difference in the predicted value and observed value) are not systematically high or low and approximately centered on zero. Also, the histograms to the right look roughly normally distributed, which is a fundamental assumption of residuals in OLS regression. There appear to be a few outliers, mainly two that severely underestimated WS/48 which is somewhat concerning and worth looking into further. Overall though, this is an impressive start.

It looks like we got ourselves a model then, right? WRONG.

However, before I go further I want to create the same model as before but this time with the statsmodels module, which is way better than sklearn in regards to statistical data exploration and tests.

OLS Regression Results

Dep. Variable:	y	R-squared:	0.989
Model:	OLS	Adj. R-squared:	0.989
Method:	Least Squares	F-statistic:	9884.
Date:	Wed, 08 May 2019	Prob (F-statistic):	0.00
Time:	09:24:01	Log-Likelihood:	13826.
No. Observations:	4759	AIC:	-2.757e+04
Df Residuals:	4717	BIC:	-2.730e+04
Df Model:	42		
Covariance Type:	nonrobust		

Summary of statsmodels OLS Regression model

```
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.61e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Warning indicating strong possibility of multicollinearity

INTERPRETATION OF STATSMODELS RESULT

Using the same features as the model we created with sklearn, we were able to develop a model that had an R-squared of 0.989, which was slightly better (albeit we did not split the data into train/test sets, a big no-no).

However, when we scroll to the bottom, we see under Warnings a message talking about a condition number being large, indicating the possibility of strong multicollinearity. What does this mean?

While I won't go too deeply into exactly what the condition number is (as it is a little above my mental pay-grade and involves the use of matrices and derivatives) but from a high level, it is a measure of how sensitive a function is to changes or errors in the input. For linear regression, it's a diagnostic for multicollinearity.

Now, this model has a high condition number which indicates multicollinearity, but what exactly does multicollinearity mean?

Collinearity means there is an association between two explanatory (independent) variables, which means that two of the features we're using have a relationship with each other. Now, multicollinearity is when two or more explanatory variables (i.e., features) in a multiple regression model are highly linearly related. 1

Going back to our statsmodels example, we can see from the features that this makes total sense. A quick example is in regards to G (games played) and GS (games started); if a player plays in more games, there is probably going to be a higher likelihood that he is going to have started in more games. In other words, these two variables have a relationship! Another example would be in all the shooting percentages -- FG%, 3P%, 2P%, FT% -- as they all depend on the number of makes and attempts which are also in the feature set!

Hopefully, you can see a trend with regards to our feature set and multicollinearity. With basketball statistics and the relatedness that a lot of them have with each other, I wasn't surprised when this happened.

Additionally, let's take a look at the P>|t| column above. This column represents the p-values that test the null hypothesis that the coefficient of that particular variable (i.e., feature) does not affect the relationship. Generally, a general rule-of-thumb is to use a threshold of 0.05, with a number below this meaning we reject the null hypothesis of no effect. Instead, we favor the alternative hypothesis which is that this coefficient does have an impact on the response.

What is slightly confusing, however, is that there are some variables, like FG and PTS, that indicated a pretty significant relationship during our exploratory data analysis (via scatter plots) but have high p-values (i.e., we cannot reject the null hypothesis that they have no effect). I hypothesize that the high complexity (i.e., high # of total features) of the model is leading to 'noise' crowding out the relationship.

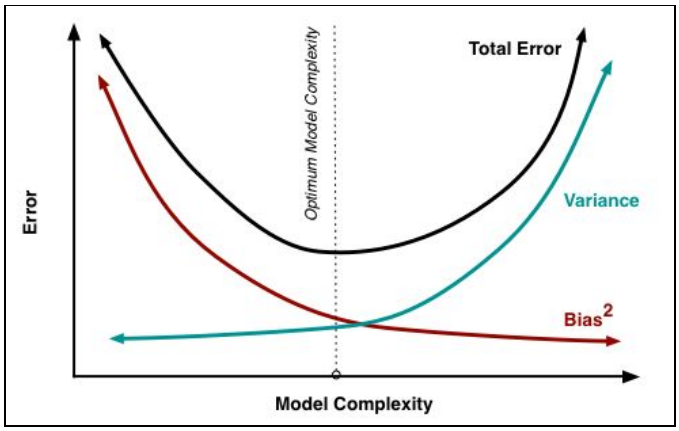
At this point we've poked quite a few holes in this model but where do we go from here?

The logical next step would seem to point towards feature selection, which is a fundamental concept in machine learning and involves selecting the most relevant variables for use in predicting the response variable, which in our case is WS/48.

Now we could begin by doing something along the lines of examining R-squared values of each of the variables and WS/48, then use a subset of variables to create a model. However, this can be tedious, and there is no guarantee that we won't reencounter multicollinearity.

Which is what brings me to a new concept, called regularization but first we must discuss that bias-variance tradeoff.

THE BIAS-VARIANCE TRADEOFF



Source: Scott Fortmann-Roe

The graph above is a visual representation of the bias-variance tradeoff.

Bias is associated with the features of the model, in that the more features you include, the less likely you are to miss the relevant relations between the features and response variable. For the most part, the more features you add, the better you can capture the relationship and the less 'biased' the model becomes.

Variance has to do with sensitivity to 'noise,' which are random disturbances in the data. Overfitting can happen when an algorithm begins to model this random noise versus the features we input into it.

As you can see the more complex we make a model, we decrease the bias which is a good thing. However, there is a tradeoff, and that is we can begin to increase the variance exponentially, which is not a good thing. This increase in variance is what has happened in our first models; we've made two models that have low bias but high variance.

Some may be wondering though, why is this whole bias-variance tradeoff a big deal? It has to do with generalizing to unseen data.

When we gather data, we are usually collecting a sample from the whole population. For example, we have gathered data from the past 10 NBA seasons yet the NBA's inaugural season was in 1946! So out of approximately 72 seasons, we have data for only 10 of those. What we are hoping for though is to create a model with this sample that we could also use on data from the 1977 season as well, and would still do a good job. In other words, we want to be able to generalize it which means we are can confidently apply it to any NBA season (given we have the statistics from that particular season).

At this point, I would not feel comfortable doing that. There might be specific patterns or 'noise' within this particular dataset that the model is including and if we were to test it on a new data set (which doesn't have this specific noise), it would probably fail. Moreover, why would you want to make a model that performs well in for only a particular context?

Knowing this we want to decrease the bias (i.e., model complexity) of our model in hopes that we can also reduce the variance and in turn minimize the total error.

LASSO OR RIDGE?

What we want to do is address the issue of overfitting and regularization is one way to do this. It does this by adding penalties to the original OLS equation, as the model complexity increases.

The two examples we'll look at -- Lasso and Ridge -- add the penalty in slightly different ways.

Lasso Regression (Least Absolute Shrinkage and Selection Operator) adds a penalty equivalent to the absolute value of the magnitude of the coefficients. Now, what does the presence of a significant coefficient mean?

It means that that particular feature is a good predictor for the outcome. However, when it becomes too large, the algorithm begins to model the intricate relations (i.e., the 'noise') which causes overfitting.

Lasso, also known as L1 regularization, enforces a penalty on these coefficients, with weights of specific coefficients forced to zero. What this particular algorithm does is perform variable selection, with the variables that are relevant having non-zero weights. However, we need to be careful because these variables are not always the most important. For example, say some variables display collinearity between each other, Lasso chooses one based on its performance in that particular data set. If you feed it another data set, it could very well select the other variable.

Because of this, Lasso is a useful tool for feature selection, but in regards to building a model, it may not be the best choice which is why we'll look at Ridge Regression.

While Lasso performs L1 regularization, Ridge regression performs L2 regularization which is slightly different. Instead of a penalty utilizing the absolute value, L2 takes the sum of the squared values of the coefficients multiplied by some alpha, which is a hyperparameter in the model.

Equations below are the cost functions of Lasso and Ridge algorithms. (*Lasso on top, Ridge on the bottom*)

$$\underset{\beta}{\text{minimize}} \left\{ \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \right\} \quad \text{subject to} \quad \sum_{j=1}^{p} |\beta_j| \leq s \tag{6.8}$$

and

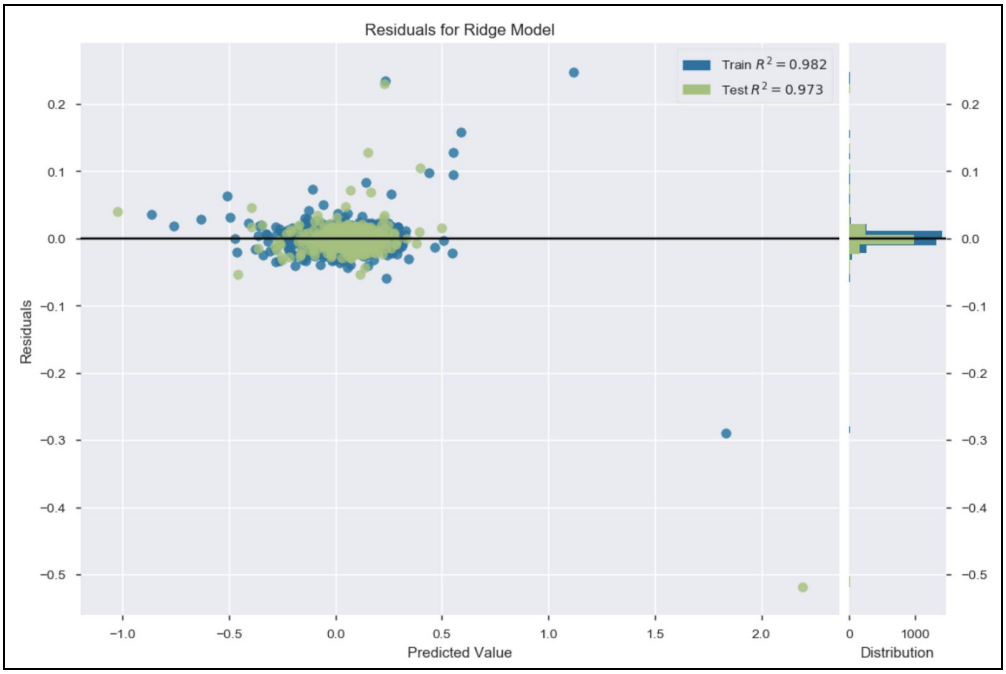
$$\underset{\beta}{\text{minimize}} \left\{ \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \right\} \quad \text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 \leq s,$$

Source: Utkarsh Gupta

The parameter alpha (α) can take a value from 0 to essentially infinity, and its value has different effects on the model. The closer to 0 alpha is, the closer we get to simple linear regression (i.e., less of a penalty on the coefficients); the higher the value of alpha gets though, the closer the coefficients get to zero meaning the model isn't modeling anything! So we'll need to optimize this when we create our ridge regression model.

Now, why is ridge regression better for this particular use case? The primary reason is that it does not eliminate any of the features. It penalizes coefficients but won't remove them.

From a domain perspective, this is a perfect starting point for predicting an NBA player's win share. Basketball includes many different, and the better players are going to have a more well-rounded skill set (i.e., they'll be able to score plus play defense, grab rebounds, etc.). I want to try and prevent losing any of the variables because they all to some extent play a role in contributing to how 'good' a player is. However, certain parts of the game are more important in contributing to wins than others, and with ridge regression, I hope we'll be able to create a model that does that.



Residual plot for baseline ridge regression model

ASSESSMENT: RIDGE REGRESSION -- Baseline Model

It looks like our first ridge regression model is setting the bar high! Firstly, let's walk through the steps:

1. Create the regressor by calling Ridge from sklearn.linear_model library
2. Fit the model on the training features and response variable
3. Evaluate the model on the hold out (i.e., test) data and check the RMSE and R-squared score
4. Plot residuals to check the assumption that the error terms (i.e., residuals) have an approximate mean of zero with constant variance

First, let's discuss why we split the data into training and test sets. After all, why can't we use all the data to create the model, won't more data make it better?

One of the primary reasons why machine learning can be so useful is that when you create a good model, it is then able to generalize to new data and accurately predict values for it. We're now going to return to a familiar subject: overfitting. When we feed in all of our data to train a model, there is a chance that the model fits the 'noise' associated with that specific data set. As a result, when fed new data with its own unique 'noise,' that's different from the 'noise' associated with the original data, the predictions have a higher likelihood of being unreliable.

With machine learning we want to avoid overfitting which is why splitting the data is a common practice. After separating the data into a training and test set, you fit (i.e., train) the model on the training set and then make

predictions on the test set, which the model has not seen. Then you compare those predictions with the known labels and assess how accurate the model was.

As you can see above, we fit the model on X_train and y_train, the training features and their associated response (i.e., WS/48). Next, we then inputted that trained model to the evaluate function. This function takes as input the X_test data, predicts WS/48 for each observation in the feature test set, then compares it to the actual label in the y_test set.

As output, it returns the RMSE and R-squared value. RMSE stands for 'root mean square error' and is represented by the following equation:

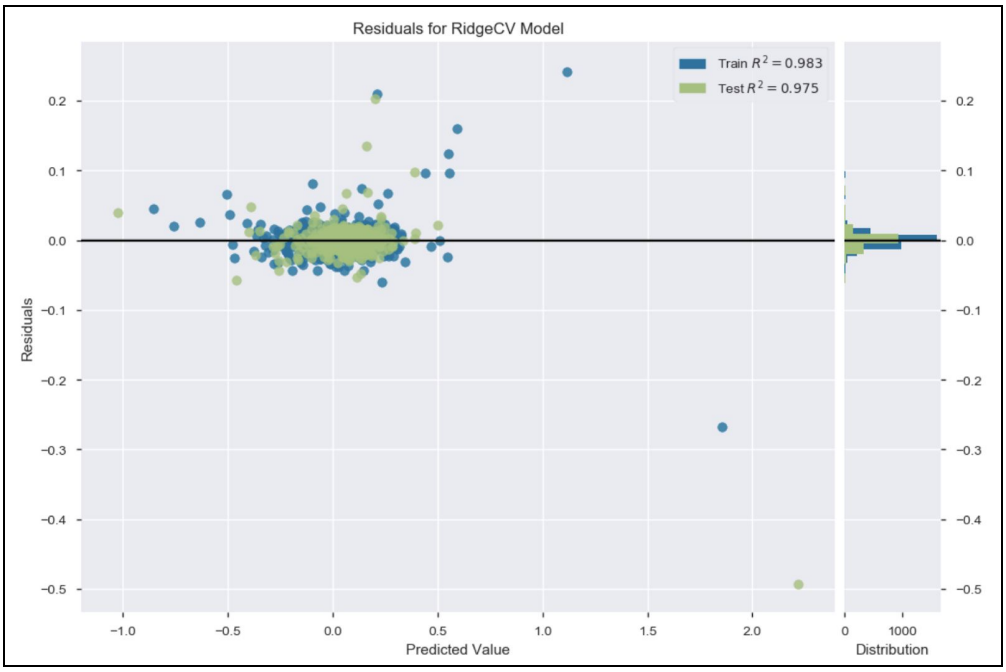
$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$$

Without getting too technical, the RMSE represents a measure of the spread of the residuals (i.e., the difference between the predicted value and the actual value) and relays how concentrated the data is around the line of best fit. We want this value to be as low as possible, since the smaller it is, the closer the values are on average to this line. Additionally, it has the benefit of being expressed in the same units as the predictions so we can gain a numerical sense of how far off the predictions are.

On the other hand, the R-squared value is not expressed in the same units but can help us better understand how well the features (i.e., the independent variables) explain the variability in the response (i.e., dependent variable). R-squared can take a value between 0 and 1, with values closer to 1 representing a model that better explains the variability of the response (i.e., dependent) variable. In summary, the higher the R-squared, the better the model fits the data.

Lastly, one of the main assumptions of linear models is that the residuals are normally distributed, which is known as homoscedasticity (meaning "same variance"). If the residuals differ in which case, a particular pattern may be present within the distribution, than homoscedasticity is violated meaning the model may not be the best fit for that specific data.

After review these metrics let's turn back to the model's results. Our RMSE is low, and the R-squared value is nearly 1, indicating a near perfect fit. Additionally, the residuals appear to be approximately normally distributed save for a few outliers. Again this is a great start, but there is yet another preventative measure we can take against overfitting, called cross-validation!



Residual plot for cross-validated ridge regression model

ASSESSMENT: RIDGE REGRESSION -- Cross-Validated Model

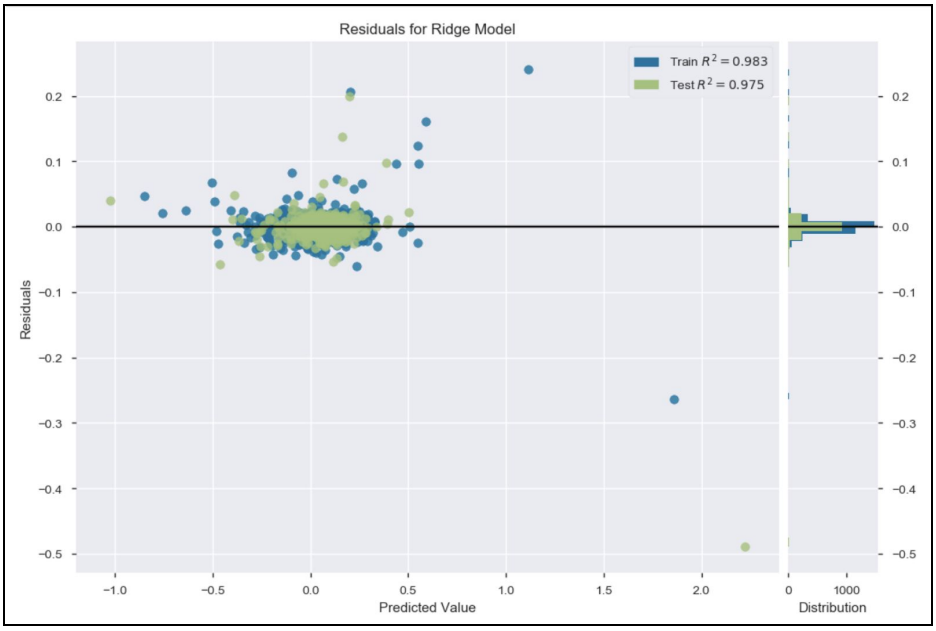
What gives? I split my data so why do we have to go a step further with cross-validation?

Splitting data is always a good first step however model performance can be influenced by the way the data was split. An easy example to understand would if you were creating a model that was trying to predict sales for each day of the week. Now, what if by arbitrarily splitting the data into just a training and test, you included mostly weekends in the test set while the training data mainly was weekdays? Most of the time, buying patterns are different on weekdays versus weekends. By training a model on just weekdays, it is going to perform poorly on the test set primarily comprised of weekend observations. In summary, the model cannot be generalized to any day of the week, only weekdays!

To avoid this, we use cross-validation. How it works is by further splitting the data into K-folds (with k being defined by you). Five and ten are probably the two most common values used for K. For example, if we are doing 5-fold cross-validation, we split the data into five folds, or subsets, of the original data. The first fold would be held out as a test data set, and then the model would be trained on the remaining four folds, representing the training data, and then we predict using that first fold and calculate our chosen performance metric (which is RMSE and R-squared in our case). After that, we would take the second fold as the test set and use fold one, three, four, and five as the training data. This process continues until each of the five folds been used as the test data set.

One thing we must remember though is that the more folds we create, the more computationally expensive it's going to get!

When we look above, we can see that the R-squared values are nearly identical when compared to our original baseline model. Now, why would we cross-validate if the performance of the model is the same? It's a preventative measure against overfitting. If there is one concept I want to get across is that we want to avoid overfitting at all costs to produce a model that can be applied to new, unseen data and produce reliable results. While it may not have been necessary for this instance, we don't know this until we use cross-validation which makes it an essential practice when producing machine learning models.



Residual plot for alpha-tuned ridge regression model

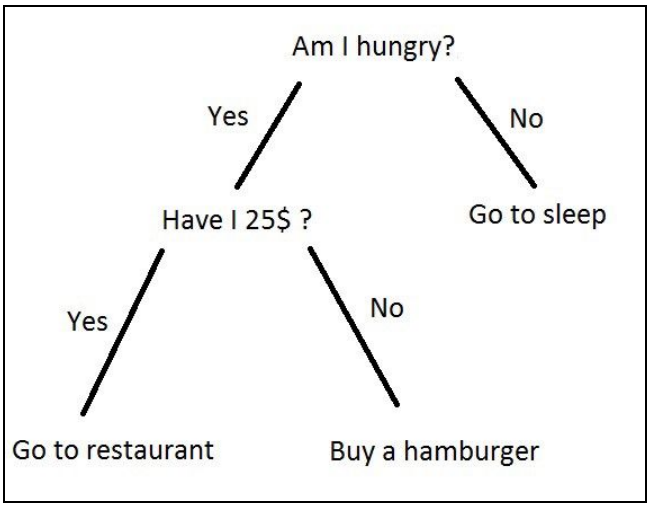
ASSESSMENT: RIDGE REGRESSION -- Alpha-tuned Model

Despite confirming through GridSearchCV that the best value for alpha is 0.1, there was no improvement in the performance of the model. At this point, I feel pretty confident in this model.

We've used cross-validation and tuned alpha, our lone hyperparameter, and from the looks of it have reached a ceiling in regards to the performance of the model. When we look at the R-squared scores, this model appears to do exceptionally well with a training score of 0.983 and a test score of 0.975 which is a good indicator that we haven't overfit the model since there was minimal drop-off between the two values.

Before we move into the Random Forest algorithm, there is one thing I want to highlight, and that is the two pretty significant outliers, one with a residual of ~-0.27 and the other with a residual of nearly -0.5! Due to their magnitude, they are probably having a significant effect on both the RMSE and R-squared scores. I guess that these outlier predictions are the result of two outlier observations from the original data set. Despite using the PER statistical threshold when we processed the data, it looks like a few outliers nonetheless snuck through.

I still feel very confident in this model. Achieving a low RMSE and high R-squared (despite these outliers), and maintaining these values after cross-validating and tuning for alpha, I believe gives further proof that this model is more than adequate at predicting WS/48.



Concept of a decision tree, simplified -- Source: Egor Dezhic

RANDOM FOREST REGRESSOR

The Random Forest algorithm is probably one of the most widely applicable machine learning models in use today according to Jeremy Howard, who is a top expert on machine learning and artificial intelligence.

Before I go further though I must first talk about decision trees. Let's begin by taking a look at the picture below:

You begin by asking yourself the question 'am I hungry?' to which you can either respond with a yes or no. At this point you to either go to sleep or ask yourself a follow-up question: do you have 25 dollars? Now if you have 25 dollars, then you can go to a restaurant, but if you don't have that much money, then you buy a hamburger.

In terms of a decision tree, the questions represent the tree's Nodes, the 'Yes/No' options represent Edges, and the resulting actions are the Leafs of the tree.

The goal of a decision tree is as follows: "to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features" (scikit-learn). Essentially what a decision tree does is take a question and break it down into its parts until you arrive at an answer.

Decision Trees are simple to understand, require little data preparation and can handle both numerical and categorical data. However, there are some downsides. Decision Trees can have a tendency to overfit and also be unstable which means even with small variations in the data an utterly different tree is created leading to entirely different results.

How do we get the benefits of Decision Trees while minimizing their downsides? By using a bunch of them, with a method called Random Forests.

Below is scikit-learn's description of how Random Forests work:

> "In random forests (see RandomForestClassifier and RandomForestRegressor classes), each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. In addition, when splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a

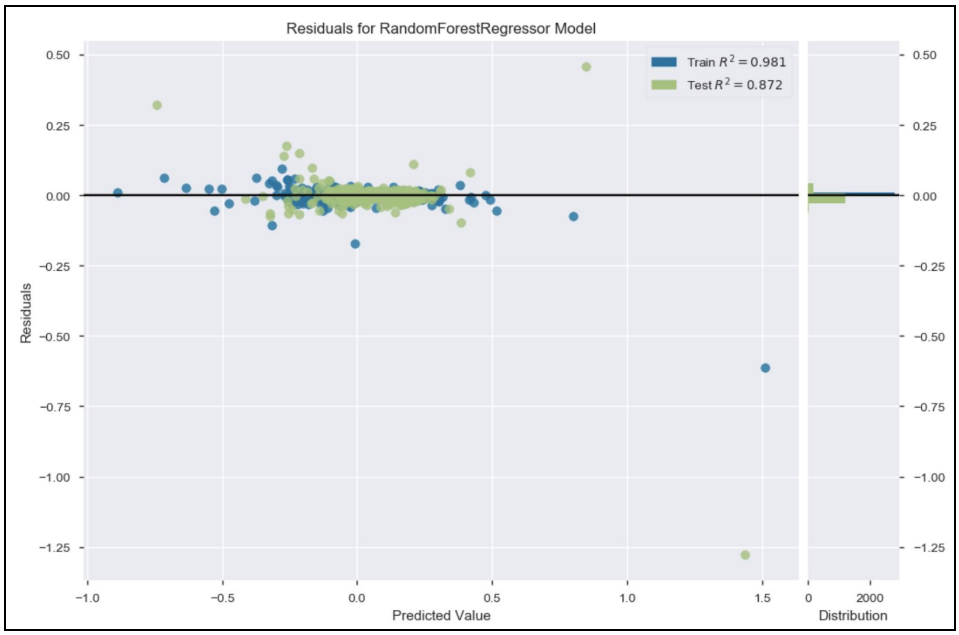
random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance also decreases, usually more than compensating for the increase in bias, hence yielding an overall better model." (scikit-learn)

In summary, multiple samples of the training data set are taken, and a unique tree is created based on each sample. Each of these trees uses a random subset of features of which it searches for the best feature amongst them. Then these multiple decision trees are merged to produce a more accurate and stable prediction as opposed to a single decision tree alone.

There are three essential hyperparameters that we can use to increase the performance of the model. They are n_estimators, max_features, and min_sample_leaf.

- n_estimators: number of (decision) trees in the forest
- max_features: number of features to consider when looking for the best split
- min_sample_leaf: minimum number of samples required to be at a leaf node

We'll tune for these a little bit later but first, let's prepare our dataset for the RandomForestRegressor and create a baseline model!



Residual plot for baseline random forest model

ASSESSMENT: RANDOM FOREST REGRESSOR -- Baseline Model

While not as good as the performance of the ridge regression model (according to the test R-squared), this model still performs admirably. An R-squared value of 0.872 is very high, but we can see that this is a rather precipitous drop-off from the training R-squared value of 0.981. A common reason for a drop off of this magnitude is that during training, the model overfits the training data and when unseen data (i.e., our test data) is fed in the performance inevitably drops.

However, when we check the plot of the residuals, we can see that the vast majority of the observations are distributed near the line at 0.0 (which is the line of best fit) and there isn't any clear pattern to them. Both of these facts are good signs. As we can see though, there are a few outlier predictions that are having a

significant impact on the results. The test set has a prediction with a residual of approximately ~-1.25. Considering that the variable we're predicting, WS/48, has a mean around 0.07 this observation is throwing a major wrench into our model.

Let's see if we can address this by further tuning the model.

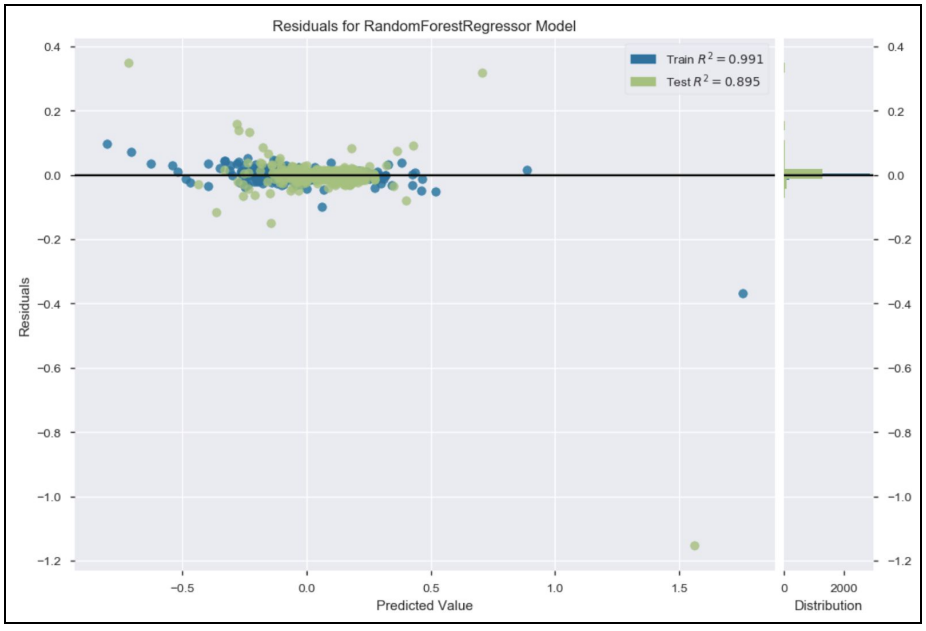
RANDOM FOREST HYPERPARAMETERS

We tune our model through the following parameters:

- n_estimators = number of trees in the forest
- max_features = max number of features considered for splitting a node
- max_depth = max number of levels in each decision tree
- min_samples_split = min number of data points placed in a node before the node is split
- min_samples_leaf = min number of data points allowed in a leaf node

Now we could use GridSearchCV to explore the above space of parameters. However, since there are quite a few combinations, this would be computationally expensive and as a result, would take quite a bit of time.

This is where RandomizedSearchCV comes in. It explores the same space that GridSearchCV but does not try all possible parameter values. Instead, it randomly selects a specified number of combinations to train and score the model. Because it is not as exhaustive, there is a slight dip in performance with RandomizedSearchCV but the tradeoff in time it takes to find the best parameters is usually well worth it.



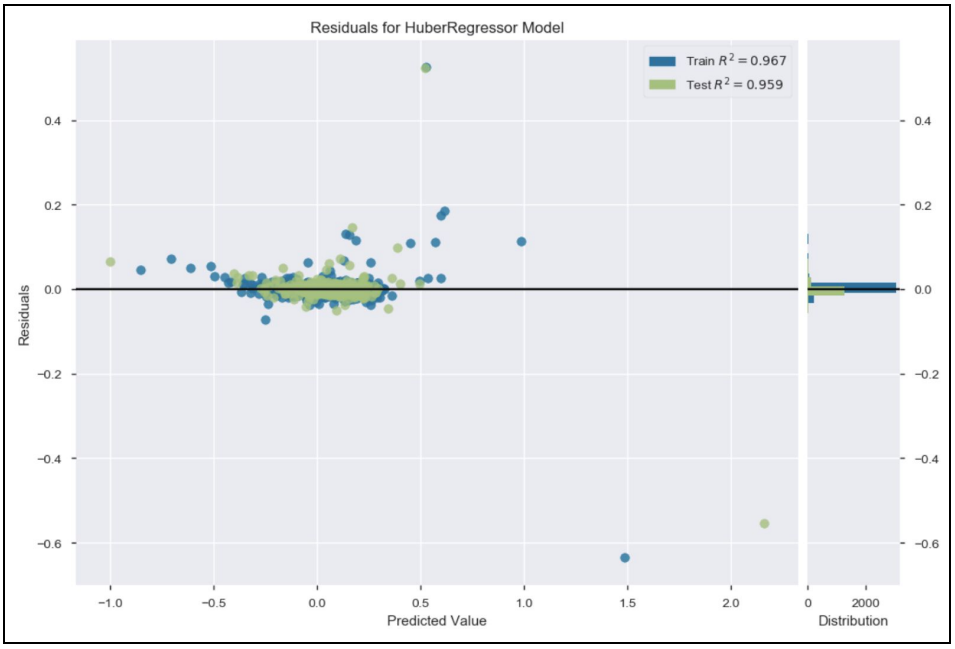
Residual plot for tuned random forest model

ASSESSMENT: RANDOM FOREST REGRESSOR -- Tuned Random Forest Model

The performance of the tuned Random Forest model is essentially the same. Additionally, it looks like that despite tuning our model we're still getting a significant outlier prediction.

I hypothesize that this may be the result of incomplete data preprocessing. When I was wrangling and cleaning the data, I instituted a cut-off using one of the variables -- PER -- that appeared to eliminate the vast majority of the outliers. However, it looks like there may have been a few observations that snuck through. This is an excellent example of the anecdote 'garbage in, garbage out'; when you feed bad data into a model, no matter how well tuned it is, you're going to get bad results.

There is one more algorithm I want to try -- the Huber Regressor -- which is a type of linear regression model that is more robust to outliers. It does through a parameter called epsilon, which 'controls the number of samples that should be classified as outliers' (source: scikit-learn).



ASSESSMENT: HUBER REGRESSOR

While the test R-squared is higher (0.959) than the random forest model, there still appears to be a few outlier predictions. This seems to be further evidence that my data preprocessing may not have been robust enough for our primary data set.

THE ALTERNATIVE DATA SET

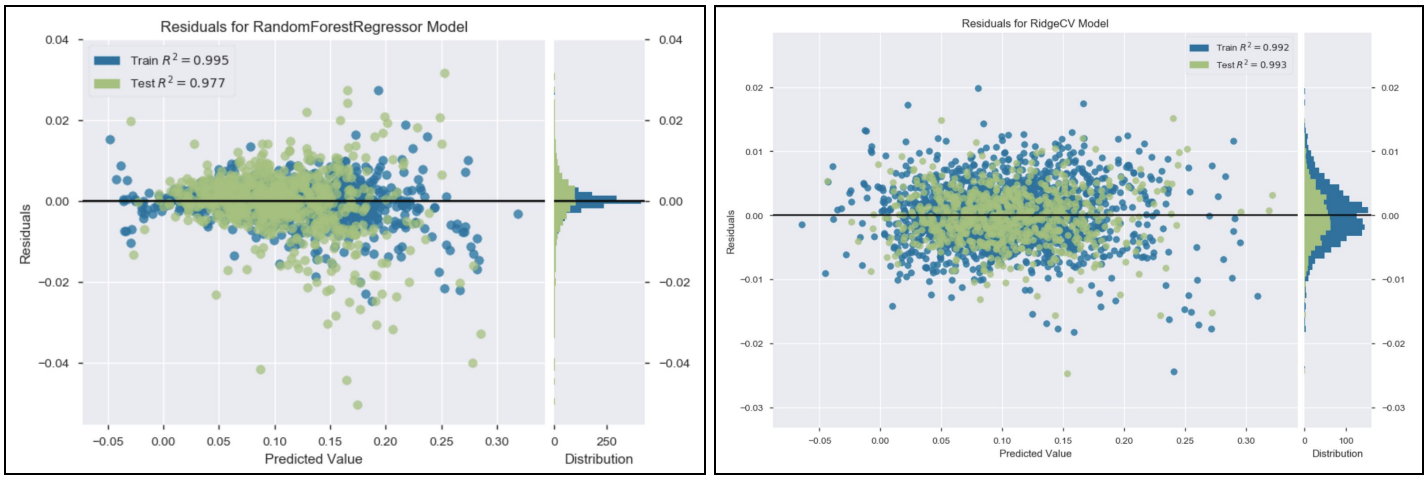
Luckily, my inner critic had a feeling that my method for addressing outliers wasn't robust enough so during the data wrangling/cleaning process I generated an alternative data set.

This data set had a stricter threshold that was per the NBA's cut-off for tracking statistical leaders. One of the prerequisites to be included amongst league leaders was to have played in 58 games. After seeing this, I filtered the observations to include only the observations (i.e., players) who met that 58 game threshold.

Now, why didn't I use it initially?

Well, it eliminated nearly 47% of the total observations that I had gathered! For my models, I wanted to try and include as many observations as possible in hopes that my model could be used to generate a prediction for WS/48 that would apply to all players, no matter if they were a starter or not.

Since we have this data readily available, let's regenerate our models from above -- namely the Random Forest and Ridge Regression models -- to see if our results differ at all.



Residual plots for random forest (left) and ridge regression (right) with new data

ASSESSMENT: RANDOM FOREST AND RIDGE REGRESSION ON ALTERNATIVE DATA

That escalated quickly! The performance of our models with the alternative data is extremely good with test R-squared values of 0.983 and 0.993 for the Random Forest and Ridge Regression, respectively.

For both models, the residuals are approximately normally distributed as shown by their respective histograms (see the graphs above). However, I do want to point out that with the Random Forest model, there does appear to be a slight fanning effect as the predicted values increase. An initial hypothesis is that this model may tend to overestimate predictions. A potential next step would be to investigate why this may be happening and account for this when training the model in an attempt to eliminate the overestimation.

That being said though, I believe that the performance of both of the models is excellent. In particular, these models highlight the fact that a model is only as good as its input data. Going forward I will be sure to be more exhaustive in the processes I use to address (and eliminate) outliers from the data.