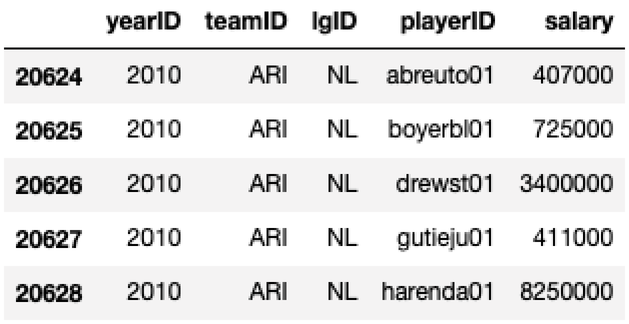
Dataset

For this project, I pulled data for Major League Baseball using the the pybaseball package in Python (https://github.com/jldbc/pybaseball). The package allows for pulling data from MLB's Statcast, Fangraphs.com, and Baseball-Refernce.com–all channels for acquiring various types of MLB data. Ultimately, I used two datasets which I cleaned and joined together to provide single dataset that includes player salaries and player stats:

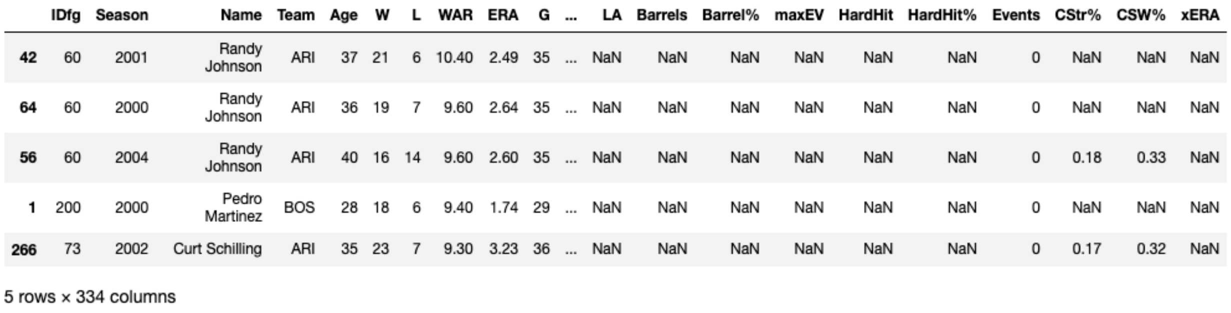
Salary



	yearID	teamID	lgID	playerID	salary
20624	2010	ARI	NL	abreuto01	407000
20625	2010	ARI	NL	boyerbl01	725000
20626	2010	ARI	NL	drewst01	3400000
20627	2010	ARI	NL	gutieju01	411000
20628	2010	ARI	NL	harenda01	8250000

The Salary dataset is simple. It contains salary data by player, by year. For this project, I decided to focus on the years 2000-2016. From this table, we'll only need the playerID, yearID, and the salary columns. Using playerID and yearID as keys to join the dataset, I merged it with Player Stats.

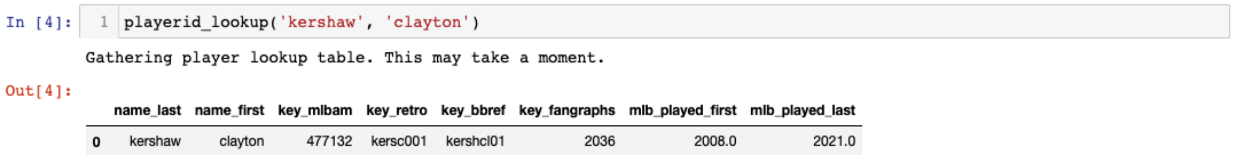
Player Stats



	IDfg	Season	Name	Team	Age	W	L	WAR	ERA	G	...	LA	Barrels	Barrel%	maxEV	HardHit	HardHit%	Events	CStr%	CSW%	xERA
42	60	2001	Randy Johnson	ARI	37	21	6	10.40	2.49	35	...	NaN	NaN	NaN	NaN	NaN	NaN	0	NaN	NaN	NaN
64	60	2000	Randy Johnson	ARI	36	19	7	9.60	2.64	35	...	NaN	NaN	NaN	NaN	NaN	NaN	0	NaN	NaN	NaN
56	60	2004	Randy Johnson	ARI	40	16	14	9.60	2.60	35	...	NaN	NaN	NaN	NaN	NaN	NaN	0	0.18	0.33	NaN
1	200	2000	Pedro Martinez	BOS	28	18	6	9.40	1.74	29	...	NaN	NaN	NaN	NaN	NaN	NaN	0	NaN	NaN	NaN
266	73	2002	Curt Schilling	ARI	35	23	7	9.30	3.23	36	...	NaN	NaN	NaN	NaN	NaN	NaN	0	0.17	0.32	NaN

5 rows × 334 columns

The Player Stats is the meat and potatoes of the dataset, providing us with various performance metrics to use as input for training the algorithm. The primary challenge with this dataset was matching the Player Names to the IDs, so that a join could be performed. Luckily, pybaseball provides a method to find playerIDs given a Last Name, First name input:



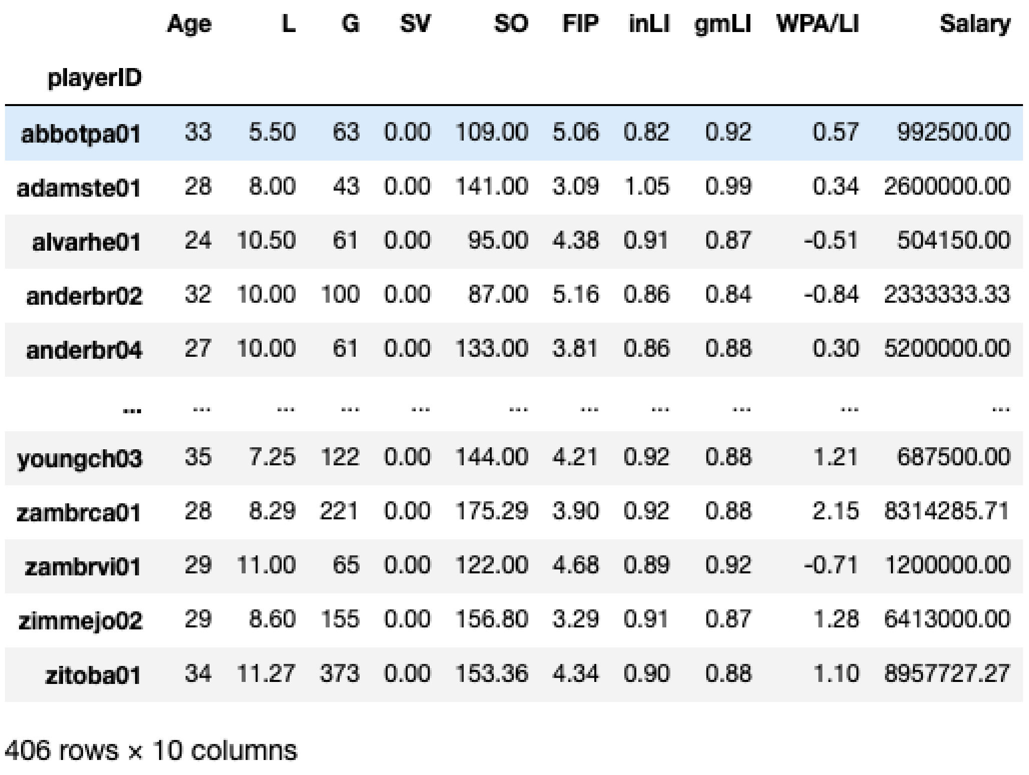
```
In [4]:  playerid_lookup('kershaw', 'clayton')

         Gathering player lookup table. This may take a moment.
Out[4]:
```

	name_last	name_first	key_mlbam	key_retro	key_bbref	key_fangraphs	mlb_played_first	mlb_played_last
0	kershaw	clayton	477132	kersc001	kershcl01	2036	2008.0	2021.0

In most cases, this was a simple task. However, many players share names, so I had to write a script to show me the list of potential matches, then manually input the playerID, like so:

```
Which ID belongs to ['Kevin Brown' 'LAD']?
 Options: ['brownke01' 'brownke04']brownke01
```

Dataset Description, Exploration



	Age	L	G	SV	SO	FIP	inLI	gmLI	WPA/LI	Salary
playerID										
abbotpa01	33	5.50	63	0.00	109.00	5.06	0.82	0.92	0.57	992500.00
adamste01	28	8.00	43	0.00	141.00	3.09	1.05	0.99	0.34	2600000.00
alvarhe01	24	10.50	61	0.00	95.00	4.38	0.91	0.87	-0.51	504150.00
anderbr02	32	10.00	100	0.00	87.00	5.16	0.86	0.84	-0.84	2333333.33
anderbr04	27	10.00	61	0.00	133.00	3.81	0.86	0.88	0.30	5200000.00
...	...	...	...	...	...	...	...	...	...	...
youngch03	35	7.25	122	0.00	144.00	4.21	0.92	0.88	1.21	687500.00
zambrca01	28	8.29	221	0.00	175.29	3.90	0.92	0.88	2.15	8314285.71
zambrvi01	29	11.00	65	0.00	122.00	4.68	0.89	0.92	-0.71	1200000.00
zimmejo02	29	8.60	155	0.00	156.80	3.29	0.91	0.87	1.28	6413000.00
zitoba01	34	11.27	373	0.00	153.36	4.34	0.90	0.88	1.10	8957727.27

406 rows × 10 columns

Note: All observations in this dataset are limited to *pitchers*. The reason for this is that pitchers are valued on their ability to pitch, and "position players" (e.g. 1st Baseman, Right-Fielder) are valued on their defense and/or batting metrics. Pitchers are not expected to bat well, nor are position players expected to pitch (sometimes...I can't make general statements like this because **Baseball is weird**. I will repeat this and handwave some explanations for the sake of brevity).

Input Variables

All of these input variables ended up scaled using a MinMaxScaler. All of them are numeric, too. After evaluating different subsets of input variables, I selected one containing the following 10 stats (we'll talk about how I settled on these stats later):

Age

- This metric was not aggregated for obvious reasons. An average career age doesn't make much sense.

L: Career Average Losses

- If a pitcher throws a ball that ends up being used to score the go-ahead run, that ultimately results in *losing* that game, they are handed a "loss".

G: Total Games Played

- This metric was not aggregated, as I wanted to include some stat that shows the importance of seniority/veteran status in baseball. Age is not a good metric for this purpose, as a player may not pitch in a Major League game until their 30s. However, a pitcher that has pitched since their 20s would have far more games than such a player–and presumably is more valuable to teams.

SV: Career Average Saves

- A save is when a lead is protected in the final inning of a game. There are quite a few details as to what constitutes a save, but the basic idea is this: when the game is on the line in the last inning, the pitcher that protects the lead is credited with a "Save". This is one of many stats that I would classify as "reliability in high leverage" descriptors of a player. Typically, you want your best reliever to be the first to answer the call in such situations. That player is typically called the *closer*. However, more recently there has been sentiment that some teams have strayed away from the idea of having a single person close out games–something that wouldn't be reflected in this data.

SO: Career Average Strikeouts

- Strikeouts are the most desirable outcomes for pitchers when it comes to getting outs. Not only are they flashy and fun to watch (some may disagree), but it completely eliminates the opportunity for the opposing team to score in that at-bat. Again: **Baseball is weird**, so don't quote me on that last part.

FIP: Career Average Fielding Independent Pitching

- FIP is a metric that aims to describe a pitcher's performance while removing outcomes of an at-bat that a pitcher has no control over. It only accounts for strikeouts, unintentional walks, hit-by-pitches and home runs. The reason for this being a preferred stat by Sabermetricians (Baseball stat nerds) over, say, the traditional "Earned Run Average" stat is because some pitchers may sport a lower or higher ERA (lower is better) than they really should. Runs can be scored on events that are simply out of the pitcher's control, such as a below average defense backing them up. Two pitchers of equal skill, with equal FIP, may not have the same ERA because Pitcher A plays with the best defense in the league, and Pitcher B plays with the worst. It's not Pitcher B's fault!

inLI, gmLI, WPA/LI: Leverage Index

- I grouped these three because they are different flavors of the same stat: Leverage Index. Leverage Index is calculated with simple arithmetic, but many calculations go into it. These leverage stats are more of what I mentioned earlier as metrics of "reliability in high leverage." To me, it was surprising just how much my model selection process favored these types of stats. Per FanGraphs:

  > Leverage Index is a measure of how “on the line” the game is at that particular moment. This allows you to determine how players perform in different situations (high, medium, and low leverage).

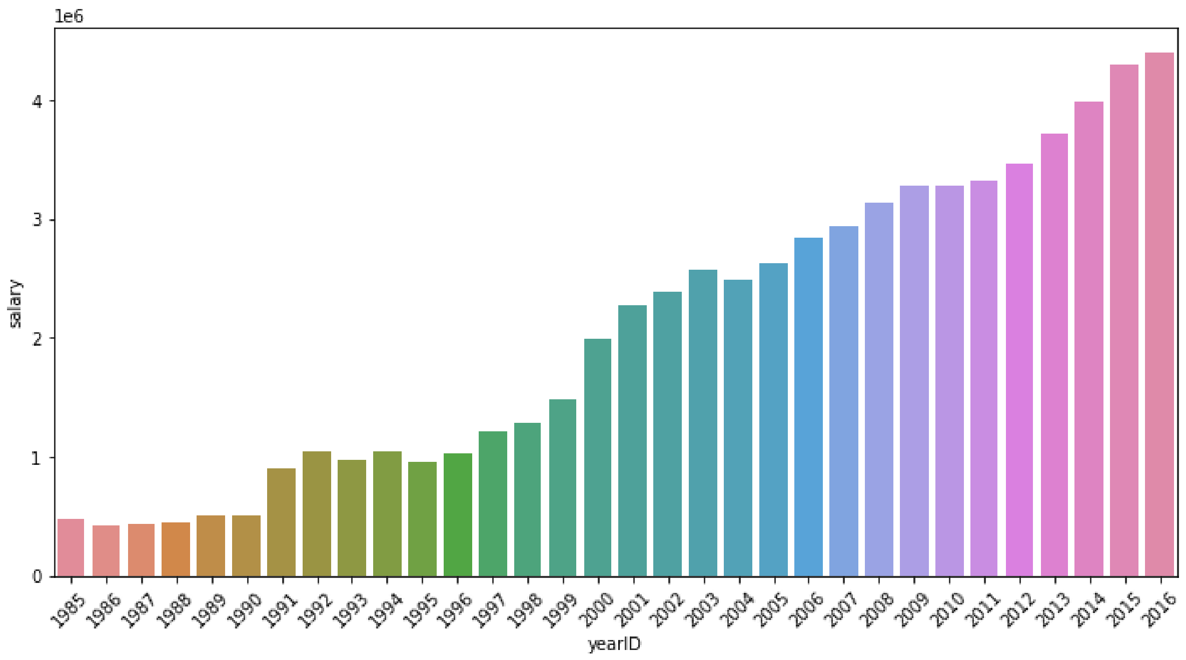
Outcome Variable: Salary

Salary is what this project aims to predict. Given all these performance metrics, can we get close to predicting a player's salary? It should be noted that this variable was deliberately *not* scaled. The reason for this is that I found the models performed better, surprisingly, when salary was left unscaled. For what it's worth, it seems to be common practice in many papers I found to simply take the natural log of salary.

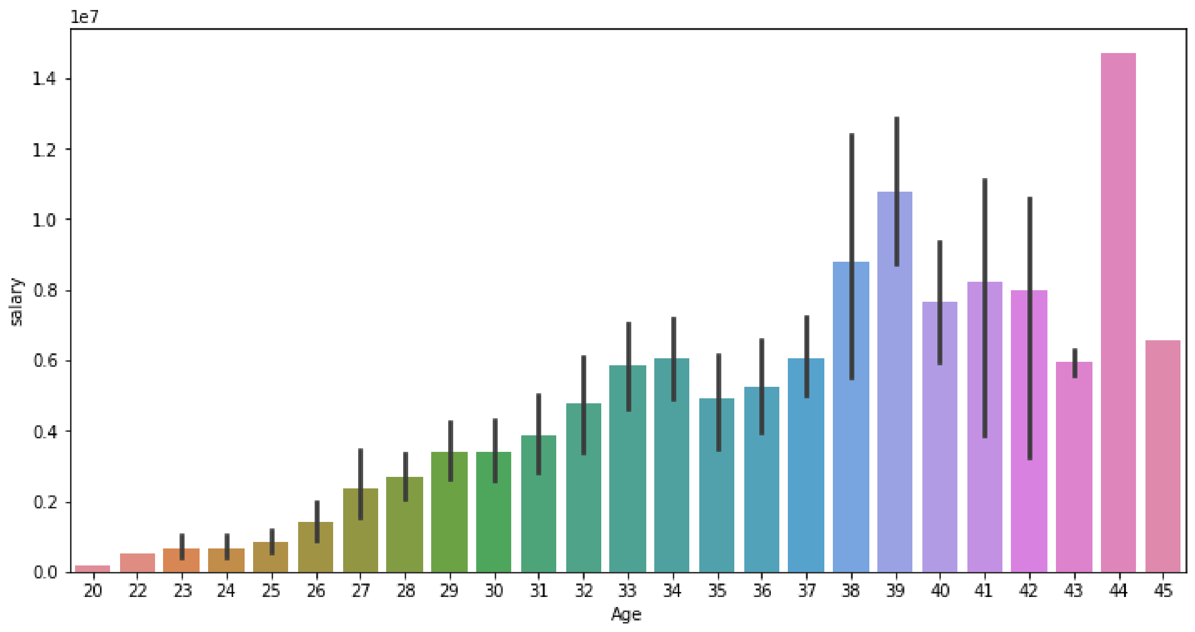
Exploration

Before cleaning the dataset, I ran a few primitive plots to explore the relationship of Salary to other variables. At first, it seemed as if there were some nice, linear relationships, such as with year and age:

Average Salary by Year



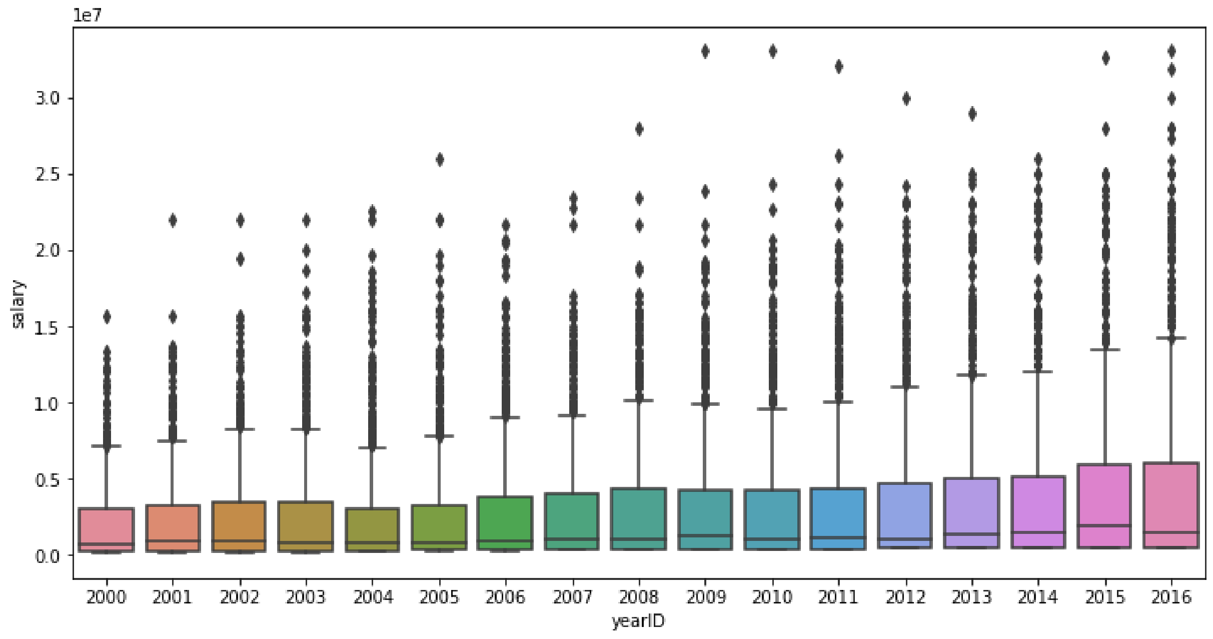
Average Salary by Age



For my dataset spanning 2000-2016, the average player earned $3,341,567 per year. However, once I looked at the distribution of salary, it became readily apparent that the mean might be a poor descriptor of MLB salaries.

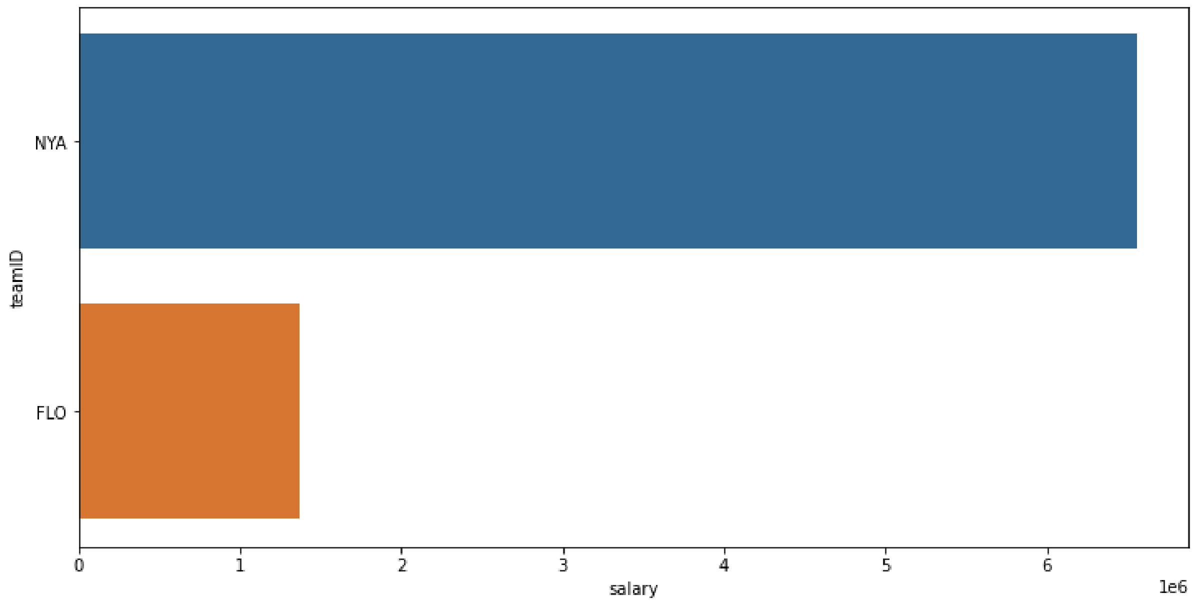
Whereas the average player earned $3.3 million, the 95th percentiel made **$9,681,105** per year. Once I looked at the boxplots, the discrepancy between the Average Joe and top earners became very apparent:

Salary Boxplots by Year

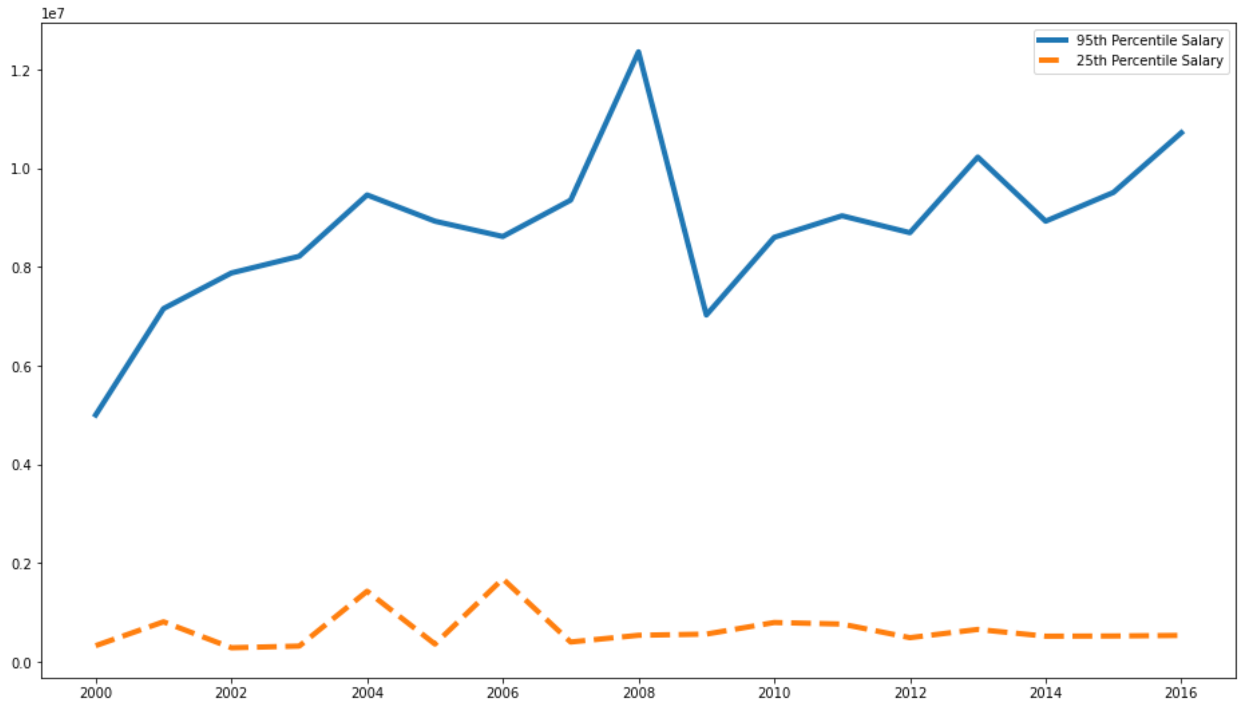


We can see from the boxplots that the IQR looks to be stagnant compared to the increase in outliers and growth of the upper whiskers. This is because while there is a salary *floor* in place, there is no *ceiling*. Some teams are less inclined to get up from that floor:

Avg. Salary for the New York Yankees and Miami Marlins (formerly Florida Marlins)



To further highlight this trend of increasing disparity, I plotted the 95th percentile earner's salary along with the 25th percentile player's salary over time:



The 25th percentile player's salary hasn't changed much in 16 years, but the top earners have nearly doubled their earnings.

EDA Takeaway:

My takeaway from this shallow dive into the data tells me two things: the rich got richer, and don't play for the Miami Marlins. The average does increase by year pretty reliably, but for most, this increase is (relatively) pocket lint.

Project Purpose

By building a model that can show teams and players what is within expectations at the bargaining table, I believe it's one of many steps needed towards bettering the league, its relations with the players, and by proxy the sport as a whole.

So, with this data I will attempt to answer the following: are performance metrics good predictors of a player's earnings? If so, which? And finally, is K-Nearest Neighbors a good method for this kind of problem? I chose K-Nearest Neighbors because the concept sounds intuitive for this type of problem. By conventional wisdom, players are often valued in relation to their peers. What is the going rate for Pitcher A, an All-Star caliber 2nd Baseman? Well, chances are he's going to look at what the other All-Star 2nd Baseman is making, and so is his team. This lends well to the idea behind K-Nearest Neighbors, which predicts the outcome of an observation by examining that observation's nearest neighbors.

Additionall, for the purpose of comparison, I'll be fitting a minimally-tuned Random Forest Regressor model to the data.

Data Preprocessing

Data preprocessing came in three steps: cleaning the data, scaling it, feature selection, and partitioning the data into training and testing sets

Cleaning

Many stats became deprecated over the years, and some stats simply don't apply to every pitcher, which results in a lot of null columns. The biggest obstacle in preprocessing was the fact that the stats dataset comes with 334 columns.

While *I* would consider what I did bad practice, as the data that is missing is not missing completely at random (or they simply stopped tracking that stat), I figured the work needed to thoughtfully impute these missing values would not be worthwhile. Besides, there are a lot of redundant stats that paint the same picture with different colors–multicolleariy abound. I ran this simple pandas filtering statement to drop those columns:

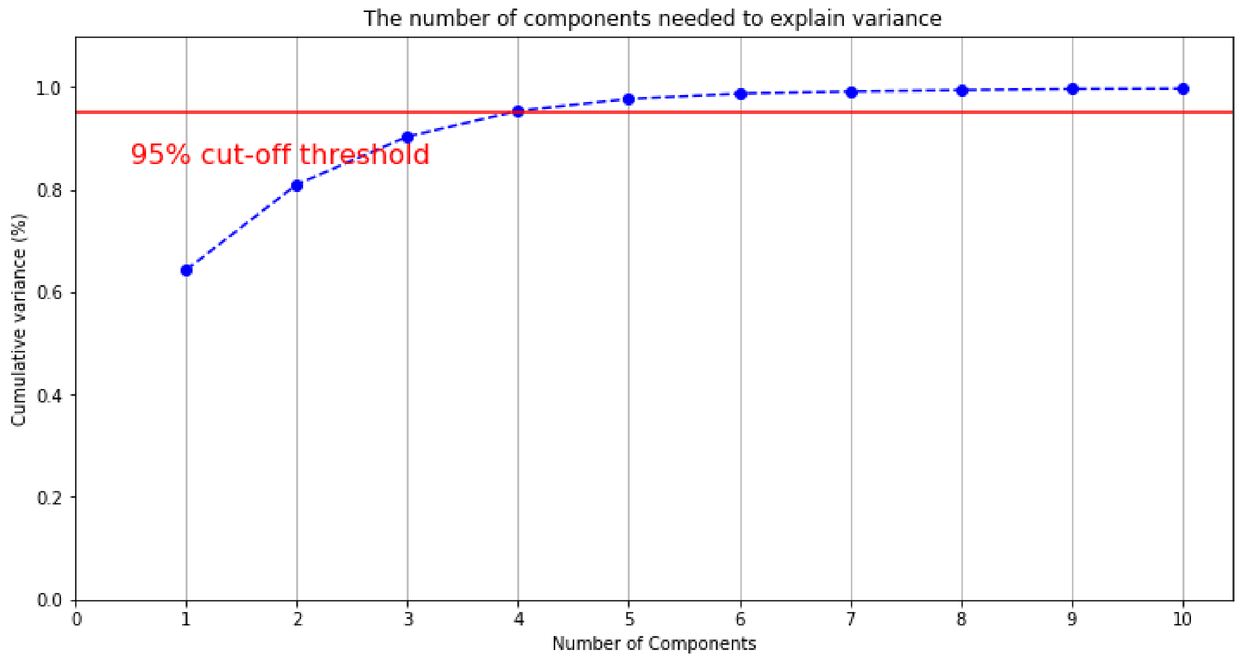
In [ ]:
```
df[df.columns[stats_df.isna().sum() == 0]]
```

This brought me down to 78 columns, which is still working in dimensions far too large to escape the curse of dimensionality.

Before attempting real dimensionality reduction, at this point I scaled the data using a Min-Max scaler.

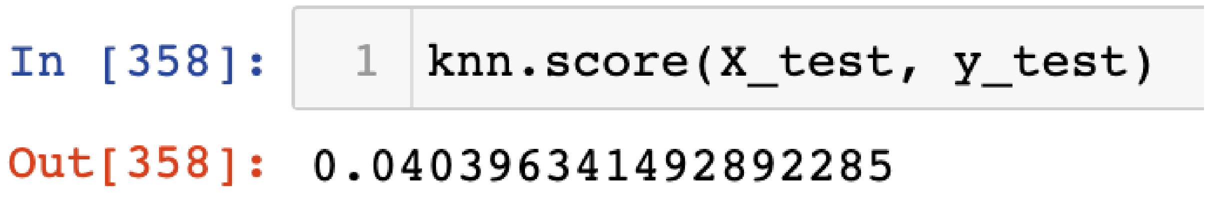
A Feeble Attempt at PCA

My first inclination was to reduce the dimensions using Principle Component Analysis. While I failed to do so successfully, it did teach me a lesson in model building: don't give up right away, and throw everything you can at the data and see what sticks.



First things first: I did *not* write original code to produce this graph. I wish I could credit the author, but I can't find the page I found it on.

Anyway, "Great!", I thought. 78 columns down to just 4, and I can still account for 95% of the variance? Sign me up. PCA has treated me well in the past, but not this time.



```
In [358]:  knn.score(X_test, y_test)

Out[358]:  0.040396341492892285
```

An abysmal R^2. At this point, I didn't even want to bother tuning the model to do well with the PCA-reduced dataset. In fact, I wanted to throw out this entire project idea. I thought that there was simply far too much unpredictable variance in the data, and nothing I had learned thus far would fair significantly better than 4%. Actually, 4% was quite generous. In another model I trained to the PCA-reduced data, I managed to get a *negative* R^2. I didn't even think that was possible.

Enter Sequential Feature Selector

Since the principle components did not do a good job at explaining the variance in our outcome variable, I decided to look at alternatives. I settled on SK-Learn's "Sequential Feature Selector", which is essentially equivalent to forward (or backward) selection in R. This required tuning a parameter that specifies *how many features to select*. I opted to tune this parameter along with the K-Nearest Neighbors Regressor in a pipeline, though I'm not sure if there are better practices for performing feature selection.

Train Test Split

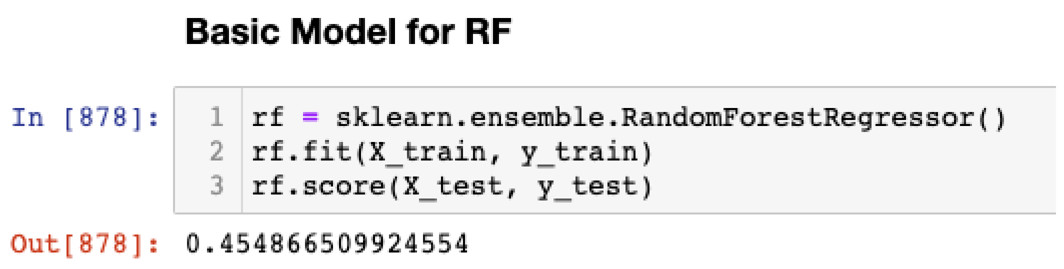
This part was straightforward: I decided on a 70:30 split for the training and test sets, respectively. After the split, I ended up with a small number of observations than I would have liked for each set (284, 122), so a cross-validated Grid Search made sense for hyperparameter tuning.

In any case, we got down to just 10 input variables after al this cleaning and feature selection. On to the training!

Model Selection and Evaluation

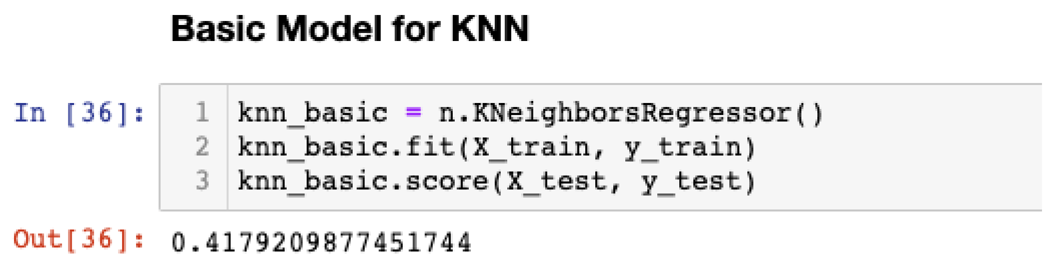
Basic, Untuned Models

As mentioned, I decided to use Random Forests as a method to compare KNN to for this problem. Here are the results of inning 1 of model tuning:



Basic Model for RF

```
In [878]: rf = sklearn.ensemble.RandomForestRegressor()
          rf.fit(X_train, y_train)
          rf.score(X_test, y_test)

Out[878]: 0.454866509924554
```



Basic Model for KNN

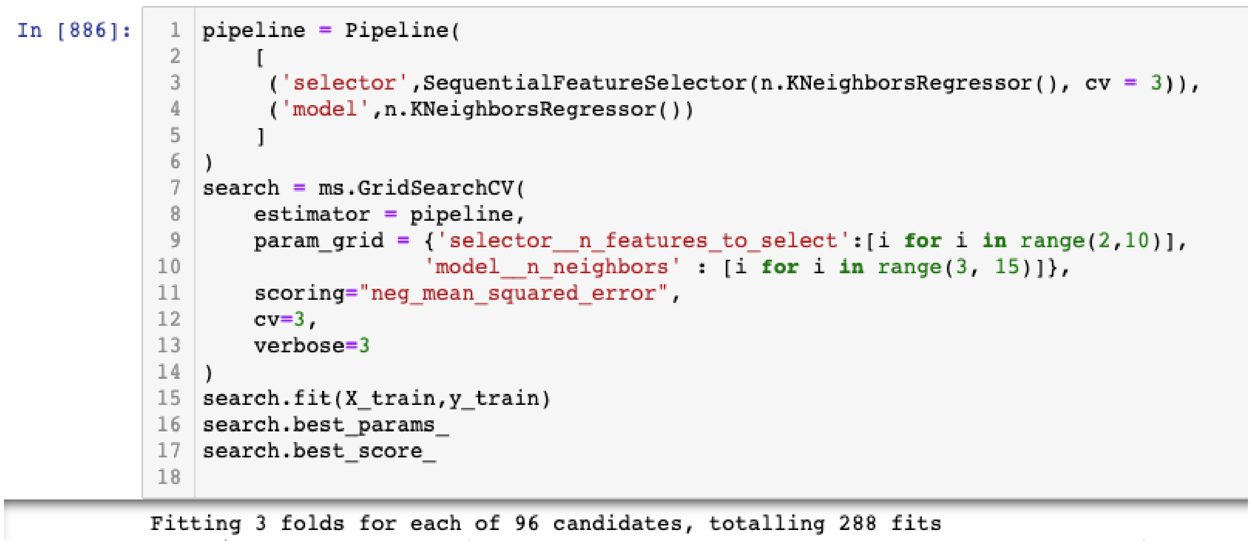
```
In [36]: knn_basic = n.KNeighborsRegressor()
         knn_basic.fit(X_train, y_train)
         knn_basic.score(X_test, y_test)

Out[36]: 0.4179209877451744
```

Random Forests wins this round. RandomForests 1, K-Nearest Neighbors 0.

Hyperparameter Tuned Models with Feature Selection

Now, let's tune the models alongside finding the Sequential Feature Selection paramater mentioned earlier (how many features to select). In both GridSearchCV'd models, the parameter "n_features" is beinged tuned along with "n_neighbors" for KNN, and "max_depth" for Random Forests. Yes, RandomForest certainly does have a lot more tunable parameters, but let's keep it fair at just one parameter each for now

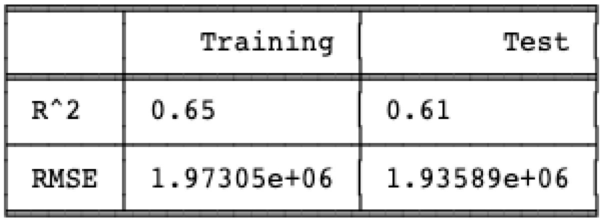
KNN, Tuned



```
In [886]: pipeline = Pipeline(
              [
               ('selector',SequentialFeatureSelector(n.KNeighborsRegressor(), cv = 3)),
               ('model',n.KNeighborsRegressor())
              ]
          )
          search = ms.GridSearchCV(
              estimator = pipeline,
              param_grid = {'selector__n_features_to_select':[i for i in range(2,10)],
                            'model__n_neighbors' : [i for i in range(3, 15)]},
              scoring="neg_mean_squared_error",
              cv=3,
              verbose=3
          )
          search.fit(X_train,y_train)
          search.best_params_
          search.best_score_

Fitting 3 folds for each of 96 candidates, totalling 288 fits
```

The GridSearch for KNN trained 96 different models, and deemed this set of parameters the best (cross-validated 3 times)–11 Neighbors and 9 Features:

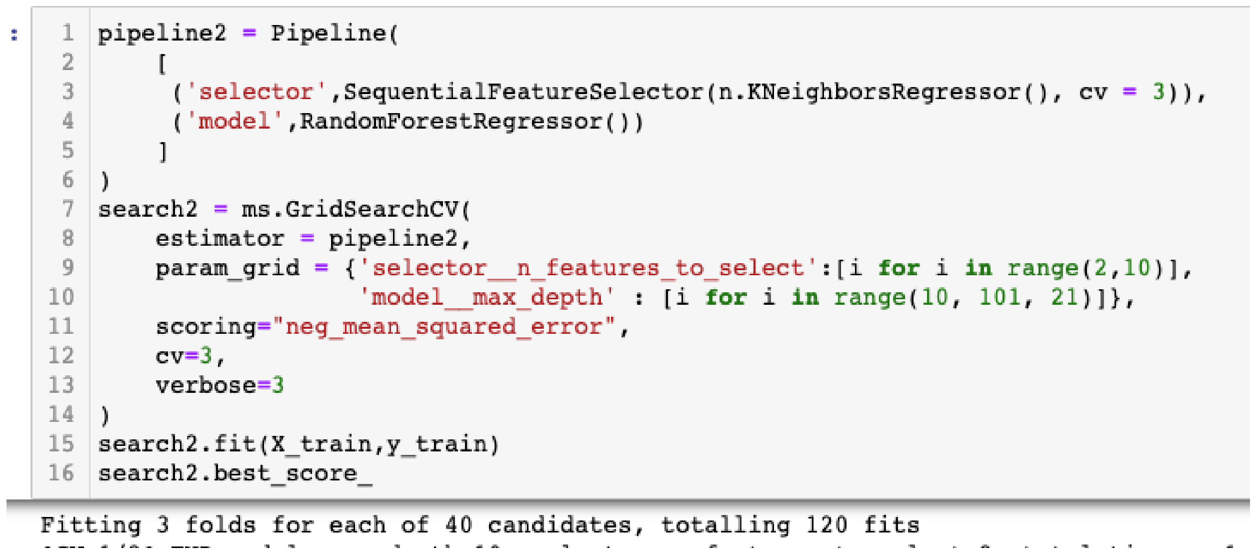
> {'model**n_neighbors': 11, 'selector**n_features_to_select': 9}
>
> ### Tuned KNN Results



	Training	Test
R^2	0.65	0.61
RMSE	1.97305e+06	1.93589e+06

Once fit to the training set and scored on both training and test sets, these were the results. Not bad! Considering other, more sophisticated models I found online ended up with a .68 R^2, I'd say we're well within the ballpark of getting somewhere. The RMSE is concerning, however. I certainly wouldn't want to potentially lose out on $1.9 million because of a lousy model. Let's see if Random Forests does better.

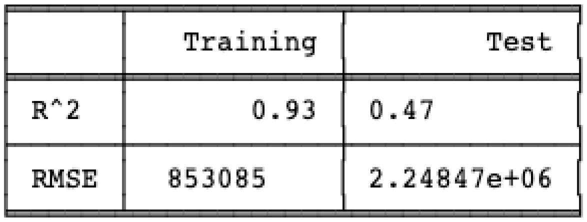
RF, Tuned



```
: pipeline2 = Pipeline(
      [
       ('selector',SequentialFeatureSelector(n.KNeighborsRegressor(), cv = 3)),
       ('model',RandomForestRegressor())
      ]
  )
  search2 = ms.GridSearchCV(
      estimator = pipeline2,
      param_grid = {'selector__n_features_to_select':[i for i in range(2,10)],
                    'model__max_depth' : [i for i in range(10, 101, 21)]},
      scoring="neg_mean_squared_error",
      cv=3,
      verbose=3
  )
  search2.fit(X_train,y_train)
  search2.best_score_

Fitting 3 folds for each of 40 candidates, totalling 120 fits
```

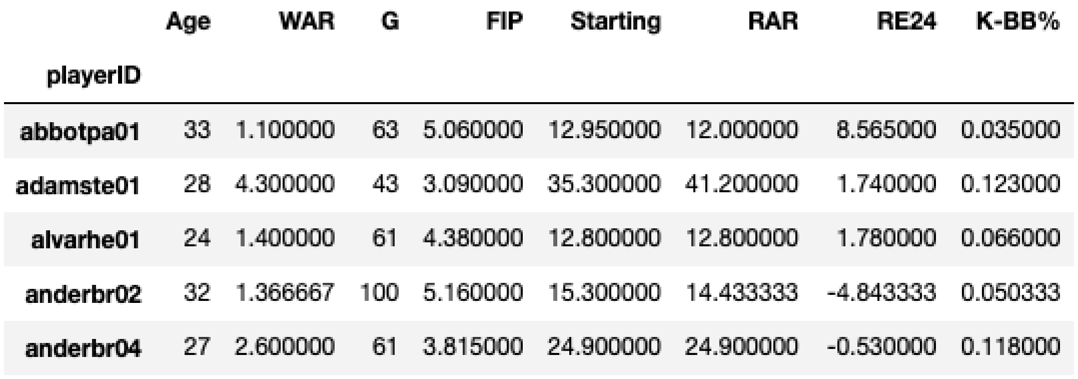
Of 40 models trained, it deemed the following parameters appropriate:

> {'model**max_depth': 10, 'selector**n_features_to_select': 8}
>
> ### Tuned RF Results



	Training	Test
R^2	0.93	0.47
RMSE	853085	2.24847e+06

Terrible overfitting to the training set, and less-than-stellar results for the test set. This level of overfitting would normally warrant some re-tuning, but K-Nearest Neighbors is the star of the show here. We're all tied up at Random Forests 1, K-Nearest Neighbors 1. Perhaps more interesting than the results is the set of variables it chose as the input:

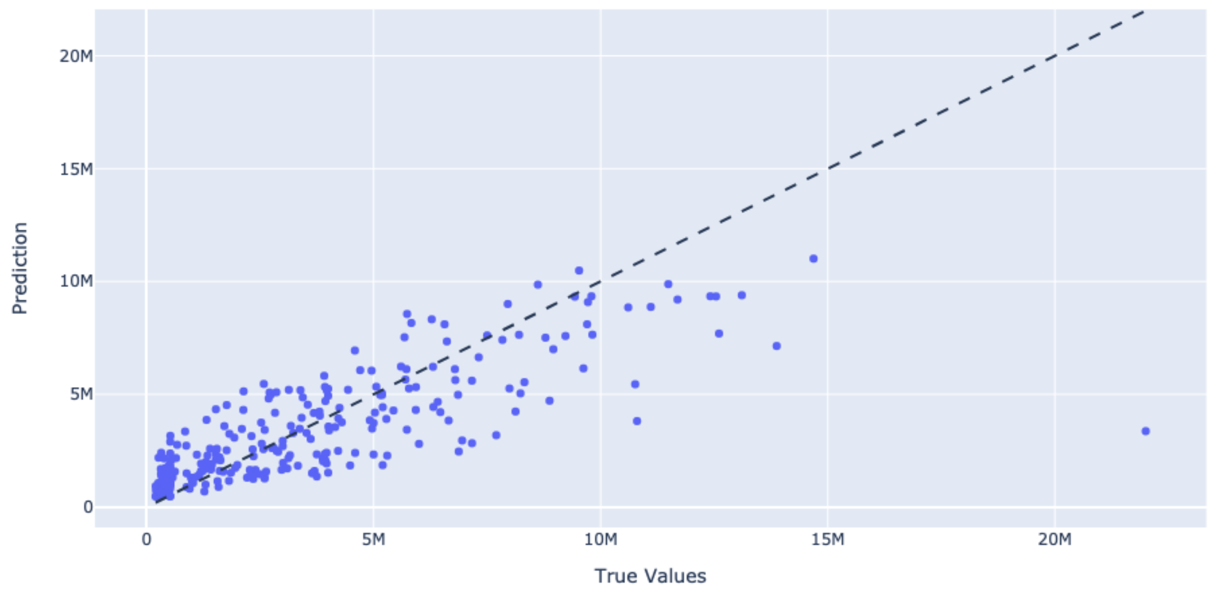


	Age	WAR	G	FIP	Starting	RAR	RE24	K-BB%
playerID								
abbotpa01	33	1.100000	63	5.060000	12.950000	12.000000	8.565000	0.035000
adamste01	28	4.300000	43	3.090000	35.300000	41.200000	1.740000	0.123000
alvarhe01	24	1.400000	61	4.380000	12.800000	12.800000	1.780000	0.066000
anderbr02	32	1.366667	100	5.160000	15.300000	14.433333	-4.843333	0.050333
anderbr04	27	2.600000	61	3.815000	24.900000	24.900000	-0.530000	0.118000

I won't get into details on what these stats mean, but know that these are far less leverage-oriented than those discussed in the dataset description section. It also includes a stat "Starting", which, as the name suggests, really only applies to pitchers who start games for the team on a set rotation. It is reassuring to know that Age, Total Games Played (G), and FIP are again chosen by a completely different algorithm.

Conclusions

As always, the answer to "Which algorithm is better?" cannot be answered. Both algorithms are definitely capable of learning and generalizing very well to this type of problem given proper tuning and input. An R^2 of 61% on a test set is far from "throw it in the garbage" performance, but again, the RMSE scores of the KNN model are what trouble me. In a practical scenario, imagine you're working for a sports agency and you are tasked with building a model to give your players a good expectation of what they can earn. Are you really going to tell your clients that their salary may be off by $1.9 million? I'm guessing no. Still, diagnostic plots showing the predicted vs. true values paint a fairly clean picture:



The massive cluster above the dashed line, close to the coordinates (0,0), show a very obvious conclusion: fresh meat in the league are *undervalued* per their performance metrics. Teams know they can typically get away with paying young players near the minimum, as there are many rules associated with salary that essentially give the teams the upper hand at the bargaining table with a young player. The values *below* the dashed line are those that the model underpredicted.

So, what are some lessons we *can* learn from this model?

Teams Desire Veterans

This is evident from the selection of two metrics in the dataset that best indicate a player's seniority in the league: Total Career Games Played (G), and Age. Now, when I say veterans in the league are desired, this doesn't necessarily mean they are *valued*. This is backed up by conventional wisdom in Baseball: even if a veteran is underperforming relative to their prime, teams still value their mentorship and will often sign them to cheap, short-term deals. On the other hand, a veteran may be *over*paid simply because they have been with a team for many years, and despite their declining performance, are still very popular with the fanbase and are well worth the money. This part is also echoed in the above graph, as the average age of players below the dashed lined (i.e. undervalued by the model) are above-average in age in the dataset.

Leverage Matters

Of course, players that have shown reliability in tough situations are always valued. It's hard to say if this should hold more weight than other statistics, as Random Forests didn't include any of them.

Further Research

Alternative Performance Metrics?

For one, the dataset used here may have been less than ideal. In Hochberg's (2011) study, they found contract year performance to be overweighted (in other words, the season leading up to their contract negotiations). Athletes are always viewed from a "What have you done for me lately?" lense. Many people write off the legends in the NBA these days simply because their age has, naturally, lessened their impact in games. In the MLB, teams simply aren't as inclined to take a chance on a player recovering from a serious injury, or at the very least they won't pay them what their career stats would warrant. On the flipside, some teams may "buy low" on such players in hope for a bargain bin steal.

Accounting for Popularity

It's no secret that superstars sell tickets. Teams across any sport are looking for marketability, as at the end of the day, professional sports are business as much as they are a game. Box Office players always get paid, if not well beyond what their performance metrics deserve. Perhaps a metric for popularity such as, say, jersey sales by player would be a good predictor of what constitutes a "Box Office" level player.

Closing Thoughts

Given what I learned in this project, and other work that has been done on the subject, I believe the ideal model would account for a handful of performance metrics, veteran status, and popularity. All are well-known desirable traits in a player, and I'd be interested to see how well a model could predict salary given that info. **Baseball is weird,** and attempting to explain it with numbers is no easy task. Teams also recognize that those who can harness even the smallest advantage by explaining that weirdness through data will perform better in the longrun–it's all probability at the end of the day.