Predicting the outcome of NBA games with Machine Learning

How we used (and you can too) machine learning to better understand the role statistics play in sports.

Josh Weiner Jan 5 · 15 min read

When deciding on a final project for our Big Data Analytics class, my partners Jack Rosener, Jackson Joffe and I looked to combine an interest in sports with the principles learned throughout the semester. After a few days of discussion, we settled on a project that would aim to predict the outcome of NBA games. While implementing our goal, we found it helpful to distill the project into the following steps with the following questions:

1. **Scraping Relevant Data** – where do we gather the relevant team and player statistics over several seasons from?
2. **Cleaning and Processing the Data** – how can we efficiently combine our scraped data so that it is both readable and usable?
3. **Feature Engineering** — what additional metrics can we append to our datasets that would help any user or ML model to better understand and predict, respectively, outcomes and trends from the data?
4. **Data Analysis** — can we determine any collinearity or other relations within the data that may better inform our predictions?
5. **Predictions** – which models and features would be most useful for us in developing an accurate prediction? Do we focus on team or aggregated individuals' statistics?

Before we get into the nitty-gritty of our workflow, let's take a moment to review and acknowledge the other work done on this exact topic. First, in 2013 Renato Torres from the University of Wisconsin-Madison set out to accomplish a similar goal as we did and predict specific season outcomes of NBA data using different machine learning models. He used several techniques featured in our project, primarily feature reduction to eliminate

techniques featured in our project, primarily feature reduction to eliminate multicollinearity from the available data, and also explored different models to explore those with the highest accuracies. Like our project, his selected features included points scored, but unlike our project had a particular focus on win-loss percentage at home and away. (We will explore our feature analysis later.)
A lot of other fantastic work on this has been done before and can be read here:

> *Cheng, Ge & Zhang, Zhenyu & Kyebambe, Moses & Nasser, Kimbugwe. (2016). Predicting the Outcome of NBA Playoffs Based on the Maximum Entropy Principle.*
> *Jones, Eri. (2016) Predicting the Outcomes of NBA Games. North Dakota State University.*
> *Fayad, Alexander. Building My First Machine Learning Model | NBA Prediction Algorithm. Towards Data Science.*
> *The Complete History of the NBA. FiveThirtyEight.*

Through our research, we found that the best published model had a prediction accuracy of 74.1% (for playoff outcomes), with most others achieving an upper bound between 66–72% accuracy. Most published research, too, focused on predicting playoff scores — which may lend towards biased data: playoff teams are more consistent in a number of stats throughout the regular season, and playoff game expected outcomes likely experience less variance as a result. Critically, note that the upset rate across the entire season in the NBA averages 32.1%. In the playoffs, the upset rate — as defined by teams with a lower regular season record winning— drops to 22% (which actually means most NBA-playoff prediction models *underperform*). Because our project looked to predict the outcome of any NBA game and is playoff agnostic, we were looking to develop a model that could reach and hopefully beat a 67.9% accuracy — and in doing so predict some upsets.

Feel free to follow along here or view our files on GitHub.

. . .

Scraping our Data

We scraped our data from available information at Basketball Reference — which has extremely detailed team and player data for each game played since the 2008–2009 season. Scraping took several days due to rate-limiting (as we had to query the results of each game over 12 seasons), and was

initially compiled into a JSON format and finally saved to csv files.

Cleaning the Data

We now had both player stats and team stats for each NBA season saved as seperate csv files. Our next step was to read in all this data and combine it into two large dataframes: one will containing the player stats for the past 12 seasons, and the other containing the team stats. Once created, we would clean the dataframes to remove invalid statistics (negative minutes) and columns that served little purpose to us (charges taken/committed, for example).

We then saved these new dataframes to the following csv files, which allowed us (and you) to skip the laborious and lengthy steps of both scraping and cleaning the data when restarting our notebook's runtime:

Search this file…

	Date	Game_ID	Season	H_Team	A_Team	H_Team_Record	A_Team_Record
0	2020-08-14	732811	2019 - 2020	Houston Rockets	Philadelphia 76ers	44 - 27	42 - 30
1	2020-08-14	732814	2019 - 2020	Toronto Raptors	Denver Nuggets	52 - 19	46 - 26
2	2020-08-14	732812	2019 - 2020	Indiana Pacers	Miami Heat	44 - 28	44 - 28
3	2020-08-13	732808	2019 - 2020	Orlando Magic	New Orleans Pelicans	32 - 40	30 - 41
4	2020-08-13	732809	2019 - 2020	Phoenix Suns	Dallas Mavericks	33 - 39	43 - 31
5	2020-08-13	732806	2019 - 2020	Los Angeles Lakers	Sacramento Kings	52 - 18	30 - 41
6	2020-08-13	732805	2019 - 2020	Brooklyn Nets	Portland Trail Blazers	35 - 36	34 - 39
7	2020-08-13	732807	2019 - 2020	Memphis Grizzlies	Milwaukee Bucks	33 - 39	56 - 16
8	2020-08-13	732810	2019 - 2020	Utah Jazz	San Antonio Spurs	43 - 28	32 - 38
9	2020-08-13	732804	2019 - 2020	Boston Celtics	Washington Wizards	48 - 23	24 - 47

NBA_Combined_Team_Stats.csv hosted with ❤ by GitHub view raw

Search this file…

	PlayerName	PlayerTeam	OpposingTeam	Date	GameID	Season	Min	SS
0	PJ Tucker	Houston Rockets	Philadelphia 76ers	2020-08-14	732811	2019 - 2020	29:03	0.
1	Hamidou Diallo	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	39:47	1.
2	Danilo Galinari	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	8:52	1.
3	Nerlens Noel	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	15:22	0.
4	Steven Adams	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	6:22	0.
5	Devon Hall	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	24:16	1.
6	Kevin Hervey	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	11:11	1.
7	Dennis Schroder	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	25:29	1.
8	Andre Roberson	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	28:07	0.
9	Terrance Ferguson	Oklahoma City Thunder	Los Angeles Clippers	2020-08-14	732813	2019 - 2020	17:28	-.

NBA_Combined_Player_Stats.csv hosted with ❤ by GitHub view raw

Feature Engineering

This is where the fun stuff begins. Our primary goal was to make all of the available data understandable: game-by-game rebounds for an entire team don't help us much unless we can use that data in a higher-level analysis that leads us to our ultimate goal – predicting wins and losses. To that end, we sought to create five different features which we would use in understanding how our teams progressed and regressed throughout each season:

1. **Elo Ratings**

This is perhaps the best existing method to relativize NBA team strength and performance over many seasons. The way Elo Ratings are calculated is simple: all teams start at a median score of 1500 and are either given or subtracted points based on the final score of each game and where it was played with weights being given to point difference, upsets, and location. In essence, it's a more sophisticated win-loss record. Most NBA-prediction models don't look at Elo Ratings but instead amalgamate a simple win-loss record with several other stats. We wanted to use Elo to appropriately weight quality wins (and losses), while also recognizing that not all teams are created equal.
The exact formula is as follows:
If *R_i* is the current Elo rating of a team, its Elo rating after its played its next game is defined as follows:

$$R_{i+1} = k * (S_{team} - E_{team} + R_i)$$

We calculate ELO for each team and each game for every season of data that we have.

Here, *S_team* is a state variable: 1 if the team wins, 0 if the team loses.
E_team represents the expected win probability of the team, which is represented as:

$$E_{team} = \frac{1}{1 + 10^{\frac{opp\_elo - team\_elo}{400}}}$$

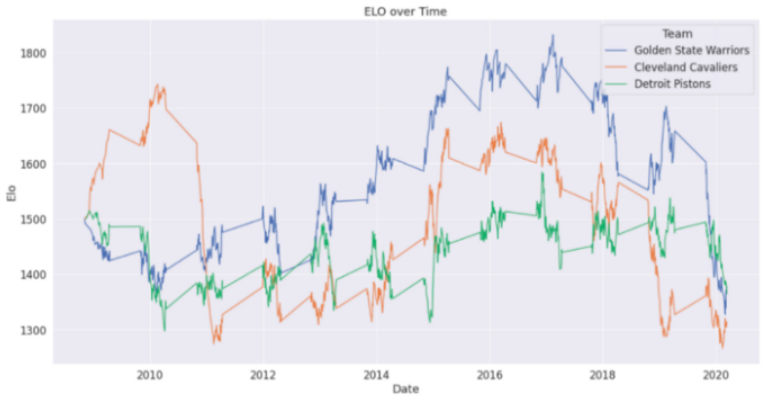
k is a moving constant, dependent on both the margin of victory and difference in Elo ratings:

$$k = 20\frac{(MOV_{winner} + 3)^{0.8}}{7.5 + 0.006(elo\_difference_{winner})}$$

It's also important to note that Elo ratings carry over from season-to-season (as all teams are not created equal, good teams tend to stay good or at least gradually decline — very rarely do teams drop onto or off of the map). If R represents the final Elo of a team in one season, it's Elo Rating at the beginning of the next season is approximately:

$$(R * 0.75) + (0.25 * 1505)$$

We can actually take a look at this metric over time, randomly selecting three teams to view, and immediately see that we can get key insights about the strength of teams throughout seasons:



Here we can actually see that Elo Ratings track quite well with teams' performances in given seasons: the years in which Golden State and Cleveland appeared and dueled in the NBA Finals is apparent by the upper peaks of their Elo Ratings. We can also see what was widely confirmed by most basketball analysts at the time: that the Western Conference was much tougher than the East — as exhibited by the influence of quality wins on Elo for the Warriors versus the Cavaliers. We can also see how far these teams fell quickly after their championship seasons and as they all suffered from roster losses and injuries. (Image by Author)

2. Recent Team Performance (Avg. stats over 10 most recent games)

These are pretty self-explanatory, we are simply looking to average the stats for each team over their last 10 games. To do this we wrote a simple function that would calculate a sliding average for a given team's stats and a window of n games:

```
1  #given a team and a date, this method will return that teams average stats over the previous n g
2
3  def get_avg_stats_last_n_games(team, game_date, season_team_stats, n) :
4    prev_game_df = season_team_stats[season_team_stats['Date'] < game_date][(season_team_stats['H_
5
6    h_df = prev_game_df.iloc[:, range(3, 43, 2)]
```

```
    h_df.columns = [x[2:] for x in h_df.columns]
    a_df = prev_game_df.iloc[:, range(4, 44, 2)]
    a_df.columns = [x[2:] for x in a_df.columns]

    df = pd.concat([h_df, a_df])
    df = df[df['Team'] == team]
    df.drop(columns = ['Team'], inplace=True)

    return df.mean()

recent_performance_df = pd.DataFrame()

for season in team_stats['Season'].unique() :
    l = ['Date', 'Game_ID', 'Season', 'H_Team', 'A_Team']
    other = list(team_stats.columns[9:47])
    cols = l + other

    season_team_stats = team_stats[team_stats['Season'] == season].sort_values(by = 'Date')[cols].

    season_recent_performance_df = pd.DataFrame()

    for index, row in season_team_stats.iterrows() :
        game_id = row['Game_ID']
        game_date = row['Date']
        h_team = row['H_Team']
        a_team = row['A_Team']

        h_team_recent_performance = get_avg_stats_last_n_games(h_team, game_date, season_team_stats,
        h_team_recent_performance.index = ['H_Last_10_Avg_' + x for x in h_team_recent_performance.i

        a_team_recent_performance = get_avg_stats_last_n_games(a_team, game_date, season_team_stats,
        a_team_recent_performance.index = ['A_Last_10_Avg_' + x for x in a_team_recent_performance.i

        new_row = pd.concat([h_team_recent_performance, a_team_recent_performance], sort=False)
        new_row['Game_ID'] = game_id

        season_recent_performance_df = season_recent_performance_df.append(new_row, ignore_index=Tru
        season_recent_performance_df = season_recent_performance_df[new_row.index]


    recent_performance_df = pd.concat([recent_performance_df, season_recent_performance_df])
```

get_stats.py hosted with ♥ by GitHub view raw

After saving this data into a new dataframe, we sought to separate each game (which contains stats for both home and away teams) into its own rows by team, which allows us to group-by and aggregate team stats much more easily and simplifies existing features. Finally, we added a win state-variable column to include the most critical measurement to our project: wins and losses.

3. **Recent Player Performance (Avg. stats over 10 most recent games)**

We create our player_recent_performance dataframe using similar methods to the above section, this time with individual players as opposed to teams. This created a dataframe of each player's performance over the past 10

games.

4. **Player Season Performance**

We also sought to include the average player stats over the entire season: unlike teams, players themselves get injured or fall in and out of the rotation and it's perhaps more critical for us to understand how player performances in individual games track with their averages. We will use this later in our models to see if it will allow for accurate predictions on the team-level.

5. **Player Efficiency Ratings (PER)**

Critically, just as we had done with teams via Elo Ratings, we wanted to be able to relativize player performance using a metric that combines seemingly unrelated statistics. Our hope was that we could use Hollinger's Player Efficiency Ratings to compare and predict team performance by the aggregated PER scores of their players. In the NBA, it is easy for players to experience wildly inflated or deflated statistics (such as points per minute) simply by virtue of the amount of playing time they get, against bench players or versus starters, number of games played, or even from outlier performances. We did not want to rely solely on player averages simply because of their ability to skew. PER solves that problem by weighting certain in-game statistics by the inverse of number of minutes played, which creates a metric that defines player performance relative to the number of minutes played.
Thus for each player, we added a column for PER in a given game according to the following formula:

> *PER = (FGM x 85.910 + Steals x 53.897 + 3PTM x 51.757 + FTM x 46.845 + Blocks x 39.190 + Offensive_Reb x 39.190 + Assists x 34.677 + Defensive_Reb x 14.707 — Foul x 17.174 — FT_Miss x 20.091 — FG_Miss x 39.190 — TO x 53.897) x (1 / Minutes)*

Data Analysis

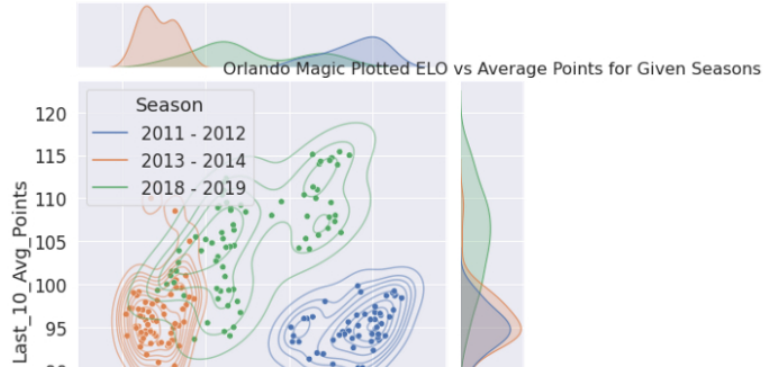
Our data analysis was centered around the use of Elo Ratings as our test metric. Essentially, could we be confident that Elo correlates with and correctly aggregates other statistics? Furthermore, would it be more appropriate for us to predict game outcomes using team stats (Elo Ratings) or averaged player stats (PER Ratings)?

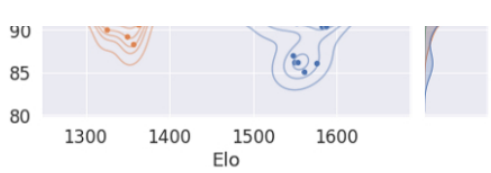
First, let's explore the density of Elo Ratings across the NBA on a per-season basis. This tells us a little about the level of parity across the league: if we can see Elo Ratings approach a normal distribution that would suggest the league's teams are relatively well-matched. Otherwise, we see large disparities and the development of "super-teams".

Twelve seasons of league Elo densities. (Image by Author)

Moving away from an understanding of Elo Ratings from a league-perspective, we endeavored to see how Elo Ratings tracked against an individual team's performance in other statistics.

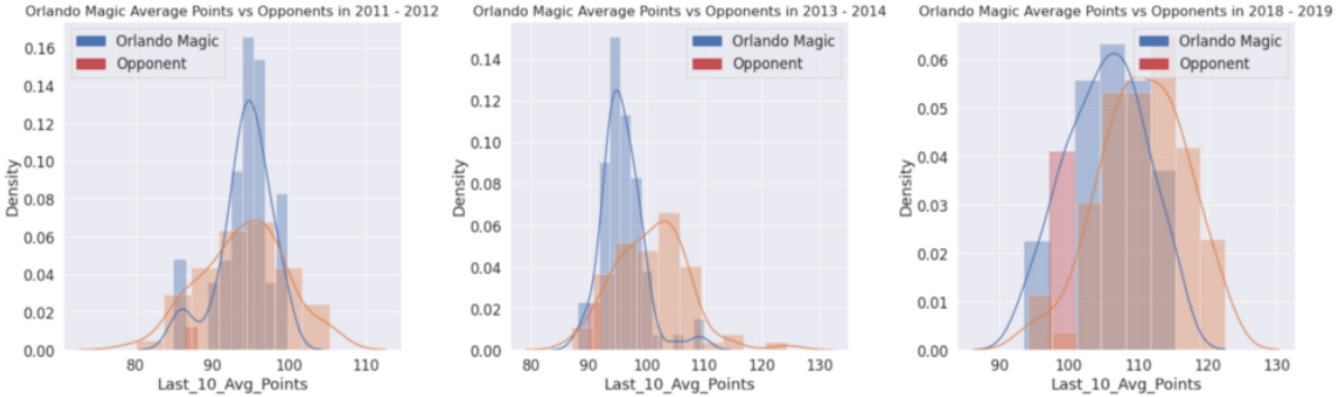
First, we looked to plot the distribution of Elo for a random team against the average number of points scored in recent games:





(Image by Author)

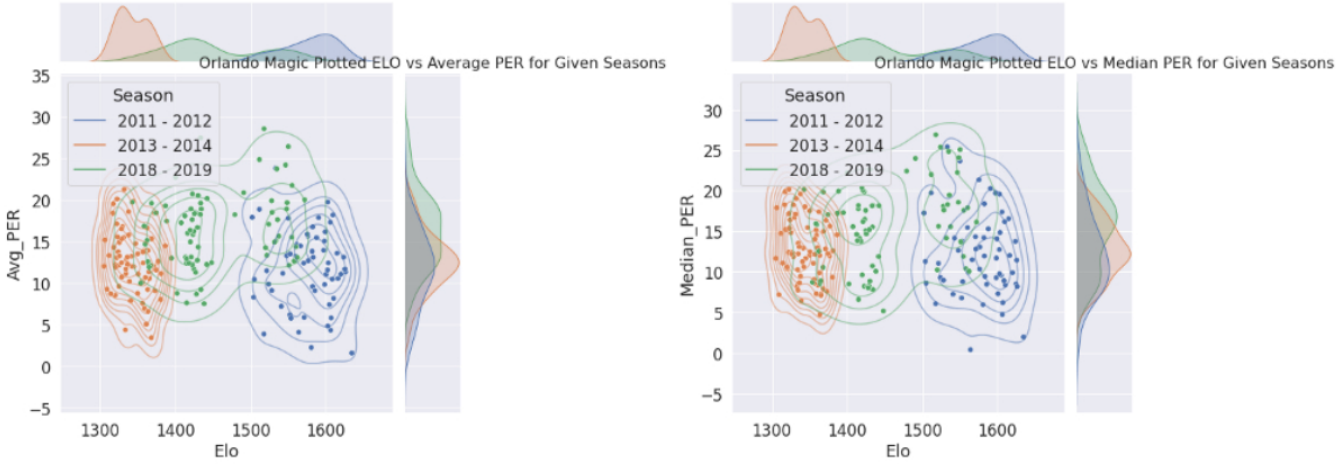
We can actually see from this that there is some correlation between the average number of points a team scores versus its Elo Rating — the higher the average points scored across a window of games the higher the Elo Rating seems to climb. However, we can also see that the Elo may also exhibit a high variance across similar scoring figures. So, to better understand how Elo Ratings track with points scored, we examine how the average points scored compares to season averages across the league – from there we can determine if points scored improves ELO, provided that high scoring is relative to the rest of the league. To do this, let's look at that same team for the same seasons and plot the distribution of points scored against its opponents.



(Image by Author)

This confirms our suspicions, as we can see that when the distribution of average points is greater than those of its opponents, or is more concentrated at an equal or higher level, the Elo is higher for those seasons. When the groupings approach an even or lesser value, those seasons' Elo ratings for the given team are lower. Therefore, average points scored is a alone a solid determinant of predicting game outcomes, but better when ***relativized***. This demonstrated for us that Elo would be a much better determinant in predicting wins for us than points, as it is by design a relative statistic.

Shifting away from team statistics, we sought to understand if Elo tracks better with player performance than it does team performance. To do this, we took a similar approach to how we plotted Elo Ratings with average points scored for the same random team, this time with PER.

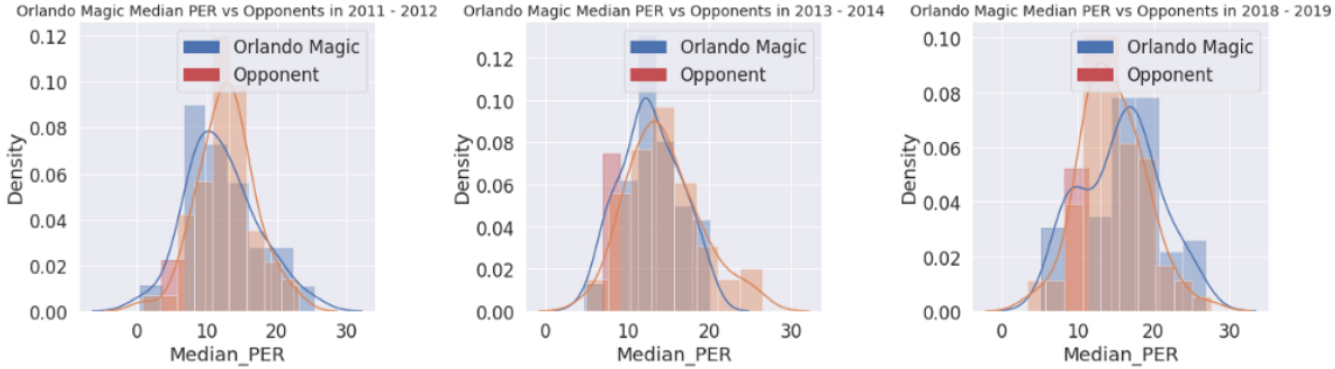


(Image by Author)

From the plotted data, we can see that aggregated PER as compared to opponents doesn't show much of any correlation with the strength of a team as determined by Elo Rating. Instead, points scored translates better — which makes some sense as a player's efficiency isn't necessarily tied to scoring the most points — and points scored against opponents is the determinant of winning a game and therefore impacting Elo.

We can see this further by mapping the Orlando Magic's mean and median PER ratings against its opponents for the same given seasons, and find that there is almost no relation between team-PER averages or medians and team strength.

From these and the above plotted distribution, we see that Average PER — while having a slight correlation to Elo Ratings throughout the season — generally show us little about how individual player efficiency impacts team strength when tracked against opponents. (Image by Author)



Median PER ratings show even less of a correlation to Elo Ratings throughout a given season. Here we can observe that in winning seasons (2011–12), the Orlando Magic had a lower median PER than its opponents for most games, yet they had their highest Elo Ratings and best record in recent years. (Image by Author)

From all of our analysis of relativized team versus aggregated player statistics, it looks clear to us that our Elo Rating and its determinants would be better features to train our models on when it comes to predicting the outcome of NBA games.

Predicting the Outcome of Games Based on Team Statistics and Elo Ratings

Our first step here was to split our data into features and columns. Reading from our dataset, once split, we then used sklearn to randomly split our data into train and test sets with an 80:20 ratio.

```
from sklearn.model_selection import train_test_split

final_team_stats = pd.read_csv('/content/drive/MyDrive/CIS545 Final Project/Final_Team_Stats.csv

cols = [0,1,2,3,4,7,8, 18, 37]
final_team_stats.drop(final_team_stats.columns[cols],axis=1, inplace=True)

features = final_team_stats.drop(columns = 'Label')
label = final_team_stats['Label']

x_train, x_test, y_train, y_test = train_test_split(features, label, test_size = 0.2)
```

model_prep.py hosted with ❤ by GitHub view raw

The first model we aimed to use to predict the outcome of an NBA game was a Logistic Regression model. Unlike a Linear Regression model which

predicts outcomes on a range of values between (and sometimes outside) 0 and 1, Logistic Regression models aim to group predictions into binary outcomes. Since we are predicting wins and losses, this type of classification suits us perfectly.

To begin, we used a simple non-parameterized LR model with our team stats and Elo Ratings as parameters using sklearn:

After playing around with some hyperparameter tuning, we found that using max_iter=131 and verbose=2 slightly improved our initial testing accuracy to 66.95%. Definitely not bad for a non-parameterized model and very close to our desired prediction accuracy. However, we sought to see if we could better tune our hyperparameters to improve our overall accuracy. Essentially, we would try out many combinations of possible hyperparameters on our data to give us the absolute best weights for our LR model.

```python
from sklearn.linear_model import LogisticRegression
from sklearn import metrics

# create a simple, non-parameterized Logistic Regression model
model = LogisticRegression(random_state=42)
model.fit(x_train, y_train)

y_pred = model.predict(x_test)
print(metrics.accuracy_score(y_test, y_pred))

from pprint import pprint
# Look at parameters used by our current forest
print('Parameters currently in use:\n')
pprint(model.get_params())

# create complex Logistic Regression with max_iter=131
log_model = LogisticRegression(max_iter=131, verbose=2, random_state=42)
log_model.fit(x_train, y_train)
y_pred_log = log_model.predict(x_test)
print(metrics.accuracy_score(y_test, y_pred_log))
```

team_logistic_regression.py hosted with ❤ by GitHub view raw

We accomplished this using cross-validation: because we only have a vague idea of the parameters we might want to use, our best approach is to narrow our search is and evaluate a wide range of values for each hyperparameter.

Using RandomizedSearchCV, we searched among 2 * 4 * 5 * 11 * 3 * 3 * 5 * 3 = **59,400** possible settings – and so the most efficient way to do this would be to take a random sample of the values.

Running our model with the best parameter values of the random samples actually decreased the accuracy of our model to 66.27%, which showed us that while random sampling helped us narrow down our hyper parameter tuning within a distribution, we would have to explicitly check all combinations with GridSearchCV.

```python
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of grid search
# Penalty type
penalty = ['l1', 'l2', 'elasticnet', 'none']
# Solver type
solver = ['lbfgs', 'liblinear']
# Maximum number of iterations
max_iter = [int(x) for x in np.linspace(start = 80, stop = 120, num = 5)]
# Multi class
multi_class = ['auto', 'ovr']
# Verbosity
verbose = [0, 1, 2]
# l1 ratio
l1_ratio = [0, 0.8, 0.9, 1]
# C
C = [0.5, 0.75, 1.0, 1.25, 1.5]

# Create the param grid
param_grid = {'penalty': penalty, 'solver': solver, 'max_iter':max_iter,
    'multi_class':multi_class, 'verbose':verbose, 'l1_ratio':l1_ratio,
    'C':C
}
pprint(param_grid)

# Instantiate the grid search model with 2-fold cross-validation
log_grid_search = GridSearchCV(estimator = LogisticRegression(random_state=42), param_grid = par

# Fit the grid search to the data
log_grid_search.fit(x_train, y_train)
best_log_grid = log_grid_search.best_estimator_
best_log_grid.fit(x_train, y_train)
y_pred_best_log = best_log_grid.predict(x_test)
print(metrics.accuracy_score(y_test, y_pred_best_log))
```

grid_search.py hosted with ❤ by GitHub view raw

In this case, implementing GridSearch only marginally increased our accuracy with our LR model.

The second model we looked to implement was a RandomForestClassifier, which can be efficiently used for both regressions and classifications. In this case, we will see if the Classifier can build a proper decision tree to determine wins from the given team-stats.

```python
from sklearn.ensemble import RandomForestClassifier
```

```
rf = RandomForestClassifier(random_state=42)
rf.fit(x_train, y_train)
y_pred_rf = rf.predict(x_test)
print(metrics.accuracy_score(y_test, y_pred_rf))

from pprint import pprint
# Look at parameters used by our current forest
print('Parameters currently in use:\n')
pprint(rf.get_params())
```

team_random_forest.py hosted with ❤ by GitHub view raw

Immediately, we get that the RandomForestClassifier reaches an initial accuracy of 66.95%, which again is pretty good. Like with the LR model, we attempted to tune the hyperparameters to give us more accurate results — first using RandomizedSearchCV.

```
from sklearn.model_selection import RandomizedSearchCV
import numpy as np

# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 0, stop = 2000, num = 11)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(0, 100, num = 6)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
pprint(random_grid)

# Use the random grid to search for best hyperparameters
# First create the base model to tune
rfc = RandomForestClassifier(random_state=42)
# Random search of parameters, using 2-fold cross validation,
# search across 100 different combinations, and use all available cores
rfc_random = RandomizedSearchCV(estimator = rfc, param_distributions = random_grid, n_iter = 100
# Fit the random search model
rfc_random.fit(x_train, y_train)
y_pred_rfr_random = rfc_random.predict(x_test)
print(metrics.accuracy_score(y_test, y_pred_rfr_random))
```

randomized_search.py hosted with ❤ by GitHub view raw

Unlike with the LR model, we find that RandomizedSearch improves our

hyperparameter tuning, giving us a better accuracy of 67.15%.

Running GridSearchCV in a similar manner to what we did above, we also sought to explicitly test 2 * 1 * 6 * 2 * 3 * 3 * 5 = 1080 combinations of settings instead of randomly sampling a distribution of settings. GridSearch also gave us an improvement from the base RandomForestClassifier, with an accuracy of 67.11%.

Overall, when running both a LinearRegression and RandomForestClassifier on the team stats and Elo Ratings, we achieved a win-prediction accuracy of **66.95%–67.15%**. For basketball games, which as we established earlier are quite variable in their actual versus predicted results, this is a significant result.

Predicting the Outcome of Games Based on Individual Player Statistics and Scoring

We then took a different approach to predicting the outcome of a game to see if we can achieve any better perfomance. Using the larger dataset of individual player statistics that we've collected, we will train a model to predict how many points a player will score in a given game. We will predict this based on a players average season stats up until the game we are trying to predict as well as their average performance over the past 10 games. We already created this data in the feature engineering section above. We will also make use of Elo ratings in our prediction as well, as presumably the higher rating of the opposing team the less points a player will score. Once we have this model we can predict how many points a team will score in a game by summing the predicted number of points of each individual player will score. With this information we will be able to predict which team will score more points and thus win the game.

Before we run our models, we need to clean our data slightly. For some games in this dataset, we have the statistics for one teams' players, but not for the other team — generally only for the first game that other team plays in the season. Thus, we will remove all these games from the dataset.

```
game_id_value_counts = final_player_stats['GameID'].value_counts()
valid_game_ids = [x for x in final_player_stats['GameID'] if game_id_value_counts[x] > 16]

final_player_stats = final_player_stats[final_player_stats['GameID'].isin(valid_game_ids)]
```

cleaning_player_stats.py hosted with ❤ by GitHub view raw

Unlike with the above games, we can't randomly split our data into train

and test sets. We are looking to use individual player statistics to predict the final score of a team, thus we must keep all players playing in the same game together. To do this, we will split up our train and test sets by game so players playing in the same game stay together. About 80% of the games will be in the train set and 20% will be in the test set:

```python
import random
games = set(final_player_stats['GameID'].unique())

train_set_game_ids = random.sample(games, 9700)

x_train = features[features['GameID'].isin(train_set_game_ids)].drop(columns = 'GameID')
y_train = label[label['GameID'].isin(train_set_game_ids)].drop(columns = 'GameID')

x_test = features[features['GameID'].isin(train_set_game_ids) == False].drop(columns = 'GameID')
y_test = label[label['GameID'].isin(train_set_game_ids) == False].drop(columns = 'GameID')
```

train_test_split.py hosted with ❤ by GitHub view raw

Instead of using a Logistic Regression model, for player scoring we will use a Linear Regression model as we are looking to predict a range of possible values (points scored) instead of simply predicting a win or a loss. Our RMSE (Root Mean Squared Error) for all players was 5.56, or the equivalent of each player making or missing around 2–3 baskets game around their averages.

On the test set, we grouped each team's predicted scoring for each game and compared it with their actual scoring numbers. Computing the numbers of games won versus the winner based on predicted scoring gave us a ratio of 1483/2528, or an accuracy of **58.66%**. Clearly, and as we realized earlier when looking at PER distributions of teams versus their opponents, aggregated player performance is too variable of a determinant to accurately predict the outcome of games — especially when compared to team performance which tends to be more consistent across games.

Conclusions and Future Considerations

As avid NBA fans, we felt that creating a model to predict the outcome of NBA games would be an interesting project and taught us a lot about building classifiers for professional sports game outcomes. We were able to utilize many of the concepts learned in our Big Data Analytics class for this project — including scraping, data cleaning, feature analysis, building models and hyperparameter tuning — and want to thank Professor Ives for his fantastic work in teaching throughout the semester.

Our Random Forest Regression model, with parameters optimized through RandomSearchCV, gave us the highest testing accuracy of 67.15%. It is slightly higher than the Logistic Regression model, and it is much higher than the Linear Regression model based on individual player statistics. Optimizing parameters using GridSearchCV and RandomizedSearchCV was time consuming and computationally costly, and it resulted in only marginal changes in testing accuracy. If we had more time, we'd likely spend less time optimizing parameters and more time selecting a model.

The best NBA game prediction models only accurately predict the winner about 70% of the time, so our logistic regression model and random forest classifier are both very close to the upper bound of predictions that currently exist. If we had more time, we would explore other models and see just how high of a test accuracy we could get. Some of those candidates include an SGD Classifier, linear discriminant analysis, convolutional network, or a naïve Bayes classifier.

Hopefully, you enjoyed reading about our work as much as we enjoyed making it — and learned something from it too.

Machine Learning Data Science Python

7 Must-Know Visualizations for Better Data Analysis

A practical guide for ggplot2 package in R

Photo by Lucas Benjamin on Unsplash

Data visualization is a very important part of data science. It is quite useful in exploring and understanding the data. In some cases, visualizations are much better than plain numbers at conveying information.

The relationships among variables, the distribution of variables, and underlying structure in data can easily be discovered using data visualization techniques.

In this post, we will learn about the 7 most commonly used types of data visualizations. I will use the ggplot2 library in R programming language. I also wrote an article that contains same visualizations created with Seaborn, a statistical data visualization library for Python.

We will use the data.table package for data manipulation and ggplot2 for visualizations. I prefer R-studio to use R and its packages. Let's first load the…

Read more · 5 min read

103

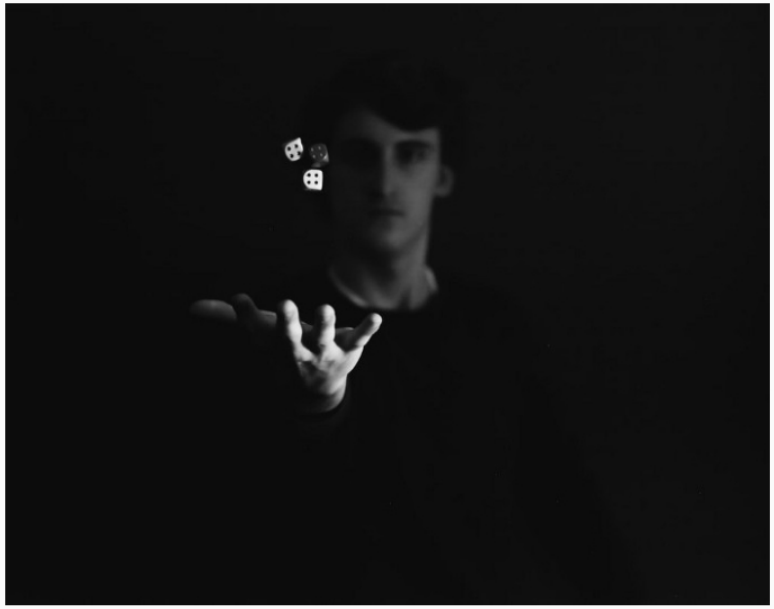
Betty Le Dem · Jan 5 ★

The Simple Intuition Behind Confidence Intervals

Improve your understanding of confidence intervals and their fundamentals.

I wrote this article to:

- Give some intuitions about confidence intervals.
- Clear some classic misunderstandings about confidence intervals (such as 95% of the data is contained by intervals of confidence, or unclear distinction between the population and samples)



Catching the uncertainty. Credits @maxfelner from Unplash.com

Why should you care about confidence intervals?

You want to rely on your guess

When you look for a value of a characteristic of a population, you first collect the data from a sample of the population and guess the parameter from this sample data (ie: the mean value, the coefficients of some models, etc). Because you could not collect *all* the data, an obvious question comes up:

> *"How reliable is my estimation"?*

I want to highlight the word **reliable** here. You can compare a confidence interval with a lasso and your parameters with a fence post, if you aim at the fence, sometimes you catch the fence, sometimes you miss. …

Cierra Andaur · Jan 5

Using the US Census API and PUMS for Data Analysis

A beginner's guide including mini-tutorial in Python.

Photo by Markus Winkler on Unsplash

It's project week here at the Flatiron School data science bootcamp and we're pulling US Census data!

There are about a gajillion (give or take) websites and links and resources and webinars and raw data regarding census data and it will take you days to go through it. Lucky for you, I spent all that time for you and have compiled my favorite resources to get you started! My goal is make the process as simple as possible so you can get to the good stuff: playing with some data!

As a newbie (literally new-born) Python programmer and data science student, I was intimidated by the thought of using an API to create and clean my own home-grown dataset, so I scoured the internet for days looking for one without needing to use an API. I knew what I wanted to focus on (marginalized and/or underrepresented communities, thank you

for asking), but every data set with the variables I was looking for was too small, too old, too sparse, or WAY too convoluted (read: complicated) for my newborn coding brain. …

Read more · 8 min read

717

Michaël HOARAU · Jan 5 ★

Having hundreds of time series to explore and not knowing where to start?

In this article, I will lay out the foundation for a framework dedicated at massive time series exploration.

Photo by Ant Rozetsky on Unsplash

In this article, I'm continuing my exploration of multivariate industrial time series. In case you need a refresher about some of the particular challenges you may meet when dealing with these, you can have a look at one of my past articles about these:

All multivariate time series are not born equal

An introduction to different families of multivariate time series and how this can impact your exploration and AI/ML…

towardsdatascience.com

Context

Let's say you are a process engineer or a data scientist working for an energy company. Some of the industrial assets you work with might be compressors, pumps, gas turbines, expanders… Any of these pieces of equipment can collect hundreds if not thousands of time series signals.

In addition, these are critical assets: any unplanned downtime can have impacts amounting to millions of dollars of damage (maintenance and repair works, replacement parts, penalties, missed revenues…). Such events are rare, and if you want to improve your predictive maintenance practice you may have years of data to explore with each signal sampled at the minute-level (or even at the second-level). …

Read more · 6 min read

35

Rod Castor · Jan 5 ★

Why is DAX Difficult to Learn?

The thorn in the side of every Power BI developer

Photo by Hans-Peter Gauster on Unsplash

You open Power BI Desktop, connect to an excel file, and drag your first visualization on to the canvas. Your eyes light up with the giddy excitement that comes from creating something useful from this powerful tool at your disposal. Now you choose to calculate something for your next chart. The first couple of formulas you add are not much different than excel formulas,

and you bristle with excitement. The world is your oyster.

Then you notice it. Wait! The numbers are wrong. Maybe a different calculation would work? Let me google it. Finding someone with the same problem online, you click on the link to their solution. Suddenly you're plummetting down a rabbit hole that feels a little like Alice in Wonderland. The solution you find is hard to understand. The enjoyment is gone. The giddiness is gone. …

Read more · 4 min read

36

Read more from Towards Data Science

More From Medium

9 Distance Measures in Data Science

Maarten Grootendorst in Towards Data Science

Are You Still Using Pandas to Process Big Data in 2021?

Roman Orac in Towards Data Science

18 Git Commands I Learned During My First Year as a Software Developer

Ahmad Abdullah in Towards Data Science

Creating Automated Python Dashboards using Plotly, Datapane, and GitHub Actions

Hakkı Kaan Simsek in Towards Data Science

Stylize and Automate Your Excel Files with Python

Nishan Pradhan in Towards Data Science

8 Fundamental Statistical Concepts for Data Science

Rebecca Vickery in Towards Data Science

You Should Master Data Analytics First Before Becoming a Data Scientist

Matt Przybyla in Towards Data Science

Building a Map of Your Python Project Using Graph Technology—Visualize Your Code

Kasper Müller in Towards Data Science