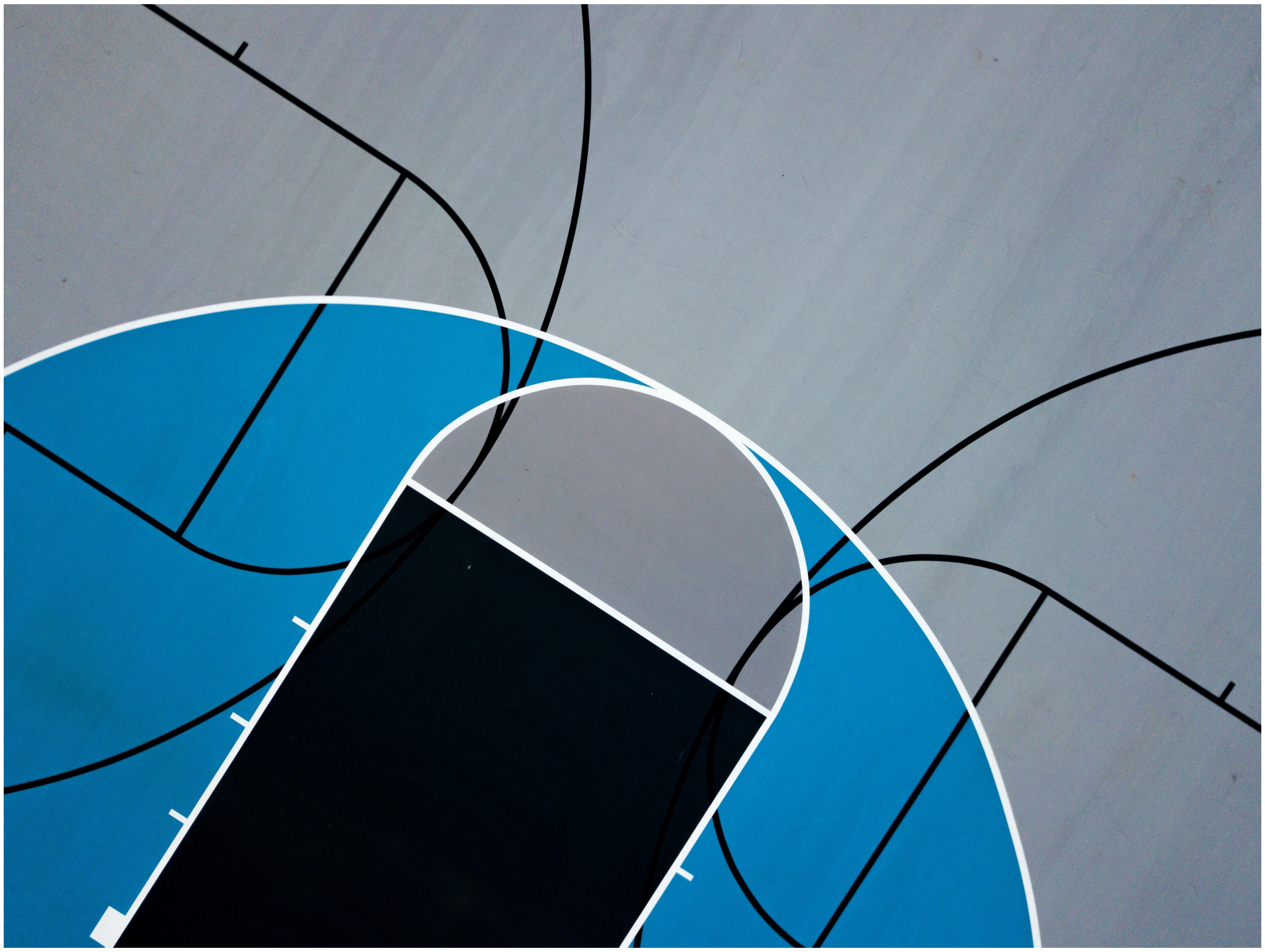


Photo by Martin Sanchez on Unsplash

Guide to Building a College Basketball Machine Learning Model in Python

Let's compare neural net performance to Vegas line accuracy.

Blake Atkinson Follow
Dec 12, 2019 · 13 min read ★

Introduction

Over the Thanksgiving holiday, I had some free time and stumbled upon a great Python public API created by Robert Clark. The API allows users to pull about any statistic for major American sports very easily from sports-reference.com. Often the hardest part of any data science work is gathering and cleaning data. While some work still has to be done, I'm naturally attracted to any project where the hardest parts have been made easy. My goal is to quickly create a model that will approach the quality of well-known publicly available systems. Gambling markets are really hard to beat, and they will provide a good measuring stick for accuracy. Although early season college basketball is often cited as a soft market, it's very unlikely this model would be profitable in Vegas — there are a few key elements outside the scope of it. My code for this project can be found on Github.

Outline

My plan is to find the most important statistics relevant to predicting college basketball outcomes. Then I can leverage two powerful models, one Light Gradient Boosting Machine (LGBM), and one neural network to predict college basketball spreads. Both a neural network and LGBM models have different strengths and weaknesses, and usually an ensemble of the two outperform each individually. They are both extremely popular in machine learning competitions on Kaggle. If I'm about to lose you, here is a great explainer video on neural networks. An LGBM model, on the other hand, has the strengths of a linear regressor yet can handle non-linear data and null values well. Here is an in-depth explanation of why it is popular on Kaggle. These models, while much more complicated than a simple linear regressor, might not significantly outperform a linear regressor (we'll test it!).

Gather Data

We need the most predictive features we can find before a college basketball game is played. This can be tricky, because sample size changes over the course of a season. Going into the season, we can use only preseason features. Late in the season, preseason features are unlikely to be important and we can rely on a sample size of 20+ games.

In order to assemble these features, we need game logs for every game. That way, we can aggregate season statistics up to the point the game is played. I compiled a list of box score URLs with the following code:

```
import numpy as np
import pandas as pd

from sportsreference.ncaab.teams import Teams
from tqdm import tqdm

seasons = range(2011, 2020) # python is odd, this is 2011 - 2019

master_array = []

for season in tqdm(seasons): #tqdm shows progress bar
    # a couple of teams don't have links for every season, thus
#"try/except"
    try:
        season_team_list = Teams(str(season))
        for team in season_team_list:
            team_schedule = team.schedule
            for game in team_schedule:
               boxscore = game.boxscore_index
               master_array.append([str(season),team.name,boxscore])
    except:
        continue

schedule_df = pd.DataFrame(master_array, columns=
['Season','TeamName','BoxscoreIndex'])

schedule_df.to_csv('./output/schedule.csv',index=None)
```

All these API calls take about 30 minutes total. For that reason, I posted the dataset on Kaggle so that you can just download it instead.

Now that we have our boxscore links, we can iterate through the boxscores and capture pretty much everything. Since I don't yet know what is useful, I will just download it all. Luckily the amazing sports-reference API has an easy command to do that. I ran this about 4 seasons at a time, and it still took multiple hours. So, again, you'll probably just want to download it from Kaggle. Roster data can be collected separately (instructions to do so can be found in the API docs).

```
from sportsreference.ncaab.boxscore import Boxscore

season = 2011 # will run once for each season
box_df = None

schedule_df = pd.read_csv('./output/schedule.csv')
```

```
season_df = schedule_df.loc[schedule_df.Season==season]
for index, row in tqdm(season_df.iterrows()):
    box_link = row['BoxscoreIndex']
    _df = Boxscore(box_link).dataframe

    if box_df is not None:
        box_df = pd.concat([box_df,_df],axis=0)
    else:
        box_df = _df

box_df.to_csv('./output/{}_boxscores.csv'.format(season),index=None)
```

Data Cleaning

In total, from 2011–2019 we have about 50,000 games and box scores. Data cleaning can be an awful mess, but it's also extremely important. Let's pray to the data science gods for good fortune and then first check for null values:

```
print(df.isnull().sum(axis=0).sort_values(ascending=False))

away_ranking                                 95468
home_ranking                                 92529
location                                     14172
away_offensive_rebound_percentage             2329
home_minutes_played                           2329
away_turnover_percentage                      2329
away_true_shooting_percentage                 2329
away_total_rebound_percentage                 2329
away_steal_percentage                         2329
away_assist_percentage                        2329
away_offensive_rating                         2329
away_free_throw_attempt_rate                  2329
away_effective_field_goal_percentage          2329
away_defensive_rebound_percentage             2329
away_defensive_rating                         2329
away_block_percentage                         2329
away_three_point_attempt_rate                 2329
home_offensive_rebound_percentage               81
home_assist_percentage                          81
home_defensive_rating                           81
...
date                                             0
away_wins                                        0
away_win_percentage                              0
away_two_point_field_goals                       0
away_two_point_field_goal_percentage             0
away_two_point_field_goal_attempts               0
away_turnovers                                   0
home_total_rebounds                              0
```

```
home_offensive_rebounds                         0
away_three_point_field_goals                    0
winning_name                                    0
```

First of all, a lot of important columns don't have null values! That's exciting considering we are relying on a public API and website to collect 100,000 boxscores (the above code collects each boxscore twice, once for each team).

Let's move along to the nulls. Because only 25 teams are ranked at a time, it makes sense that many teams have null rank columns. If we didn't have much data, I would rely on the ranking columns as an input. However I think we can safely drop those columns. Location must also be hard to come by in some cases. I'm not going to incorporate travel into this model, and so I'll ignore that column. Travel, or cumulative travel over a given time period, would be an interesting addition!

Upon further inspection, it is apparent that advanced stats are only tracked for Division I teams. That's why there are way more nulls in the away advanced stats than the home advanced stats — Division II teams usually have to play in Division I arenas. We don't really care about Division II advanced stats anyway, and so it isn't a problem. This data cleaning is going a lot smoother than I expected.

There are other steps to data cleaning than just checking for null values. I went through some key columns (points, assists, wins, etc.) and checked their min values and max values as well as used seaborn's distplot to check if they made sense. Everything seems good for now!

Feature Creation

Simple box score stats will help us predict future success of an NCAA basketball team. There are different ways to input team quality into our model. We could use an average points scored over the past 10 games. We could use average point scored over the past 5 seasons. We could just input top 25 ranking, and the model would learn that teams ranked 1–5 are really good teams. I'm trying to reasonably limit the scope of this project, but there are near infinite combinations to try.

As an aside, I want to talk about exponential weighted average (EWA). Instead of averaging stats over the past X games, it's sometimes better to weigh more recent games (even slightly) heavier than older games. I will be using both EWA and season-long average versions of box score statistics. Stats with high game-to-game variance (3 point shooting percentage, turnovers) will benefit from long term samples.

Also, I have to come clean. I've already done a lot of feature selection work. If the following statistics seem oddly specific, it's because I've already eliminated similar but less useful features. There were so many combinations to try, and I didn't want to waste valuable reader time going through the whole process. I'll add in a handful of untested, likely-not-useful features to show how I'd test for importance.

I'll start with long term features. We're talking very long term. For the features below, I took an exponential weighted average over the past 5 seasons:

1. Win %
2. Offensive Rating (Points scored per 100 possessions)
3. Offensive Rebounding Percentage (percent of available offensive rebounds that result in an offensive rebound)
4. Opponent FG% Allowed
5. Pace
6. Percent of points from free throws (FTs)
7. FT shooting percentage
8. Percentage of points from 3-pointers
9. 3PT shooting percentage
10. Turnover Percentage (percent of offensive possessions that result in a turnover)
11. Steal Percentage (percent of defensive possessions that result in a steal)

Already, we have 11 features. However, we'll need the same 11 features for the opposing team. So we're up to 22 features. That's a lot! You can see how this can easily get out of hand.

Okay, so we also need more recent samples than just the past five years. The following statistics I'll use a season-long sample.

1. Defensive Rating
2. Win %
3. Opp 3PT Shooting %
4. Fouls Drawn
5. Percentage of points from 3PT shots
6. Pace
7. Free Throw Shooting %

And last but not least, we need our statistics weighted heavily over the past few games:

1. Offensive Rating
2. Steal percentage
3. Assist percentage
4. Block percentage
5. Two point shooting percentage
6. Assists allowed percentage

So that's 24 features per team, or 48 features per game. It's also very important to tell our model how far along in the season we are and who the home team is.

Context Features:

1. Day of season
2. Home/away

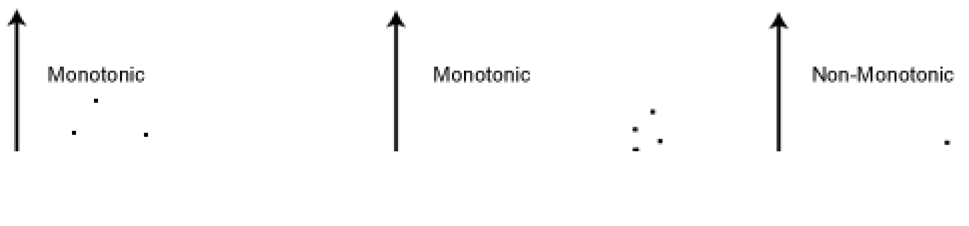
Now we have a nice, even 50 features. Can you find features that would add to the model? Absolutely! This model is yours to play with. For demonstration purposes, I'll add in a couple that probably aren't useful. I actually haven't tested them yet, so I can't rule them out!

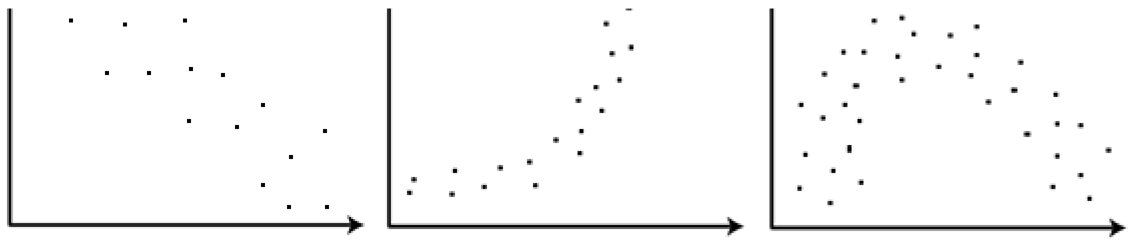
1. Average personal fouls this season
2. Season free throw percentage
3. Season turnover percentage

Feature Importance & Selection

There are 3 primary ways I like to test feature importance. In order, from most lazy to least lazy:

1. LGBM gain — our LGBM model can spit out what features it finds most important. While this gives an idea of rough importance, only using gain has drawbacks too. I will touch on the drawbacks of all these methods throughout the article.
2. LOFO importance — "Leave one feature out", this method tests the model by leaving one feature out each run. The obvious drawback is that it takes a lot of time. This drawback is offset by the fact that it is very widely applicable to different models and input data. I only recently learned of this method, and I want to give credit to Ahmet Erdem for open sourcing it and making it easy to use.
3. Pearson's, Spearman's, and Hoeffding's D correlations— Hypothetically, in the real world, sometimes you have to defend your work. In such cases, it's not enough to say "I threw it in a black box machine learning model and this is what I got". Thankfully, statisticians have done the heavy lifting for us. Pearson's correlations measure linear relationships, Spearman's correlations measure monotonic relationships, and Hoeffding measures non-linear relationships. We can run these correlations for each feature in the model vs. the target and figure out relative usefulness. These are also very useful in finding certain parameters. For example, if I want to know how much to decay my exponential weighted average for assist percentage, I could optimize Spearman's correlation to find the best decay value.




Pearson's would estimate the first well, Spearman's the first and second, and Hoeffding's D the third. You can read more about these correlations here or here. If I had to guess, Spearman's would work on almost all of these college basketball features.

Baseline Results

How do we grade our models? Well, the model is trying to predict the Vegas spread. So if a good team plays a bad team, the model will give an expected winning margin. Let's say the model predicts the good team wins by 10, but they actually win by 8. Then our mean absolute error is 2. Likewise, if the good team wins by 12, and our model again predicts 10, our mean absolute error is also 2.

Here is my model code. You'll notice I'm using five fold cross-validation to help give a reliable estimate of model error and I'm also recording feature importances.

```
from sklearn.model_selection import KFold, cross_val_score, train_test_split, cross_validate
from sklearn.metrics import mean_absolute_error
import lightgbm as lgb

n_folds = 5
feat_names = list(X)

def mae_cv(model, lin_reg=False):

    kf = KFold(n_folds, shuffle=True, random_state=17).get_n_splits(X.values)
    output = cross_validate(model, X.values, y, cv=kf, scoring = 'neg_mean_absolute_error', retu
    all_estimators = []
    for idx,estimator in enumerate(output['estimator']):
        if lin_reg:
            all_estimators.append(list(estimator.coef_))
        else:
            all_estimators.append(list(estimator.feature_importances_))

    all_estimators = pd.DataFrame(all_estimators)
    avg_imp = all_estimators.values.mean(axis=0)
    feature_importances = pd.DataFrame(avg_imp, index = feat_names,
                                        columns=['importance']).sort_values('importance', ascendi

    feature_importances.columns = ['feats','importance']
    return output, feature_importances

model_lgb = lgb.LGBMRegressor(objective='regression',learning_rate=0.09,importance='gain')
output, feat_imp = mae_cv(model_lgb)

print("LGBM score: {:.4f} ({:.4f})\n" .format(-1*output['test_score'].mean(), output['test_score
```

cbb-model-embed hosted with ♡ by GitHub   view raw

The mean absolute error, using the 56 features above:

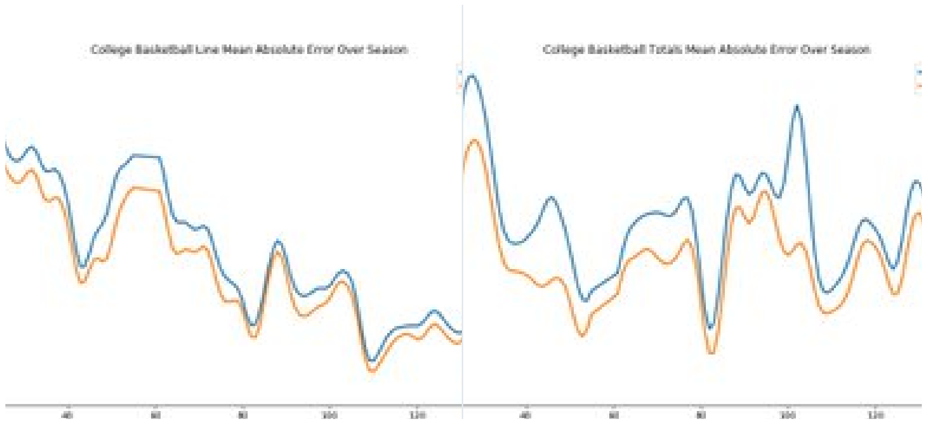
```
Sklearn's Linear Regressor: 9.17

LightGBM model: 9.09
```

That means on average, our prediction is about 9 points off. Is this good? Who knows! Let's compare it to Vegas spreads. A few days ago, I found historical lines and results. Apologies for linking my own twitter thread, but I talk about season to season error and error over the course of the season:

Blake Atkinson
@BlakeTAtkinson

With the NFL Big Data Bowl over, I had some time to get back to CBB and answered some questions on sportsbooks. Do college basketball lines get sharper over the course of a season? Yes, until the latter stages of the tournament:



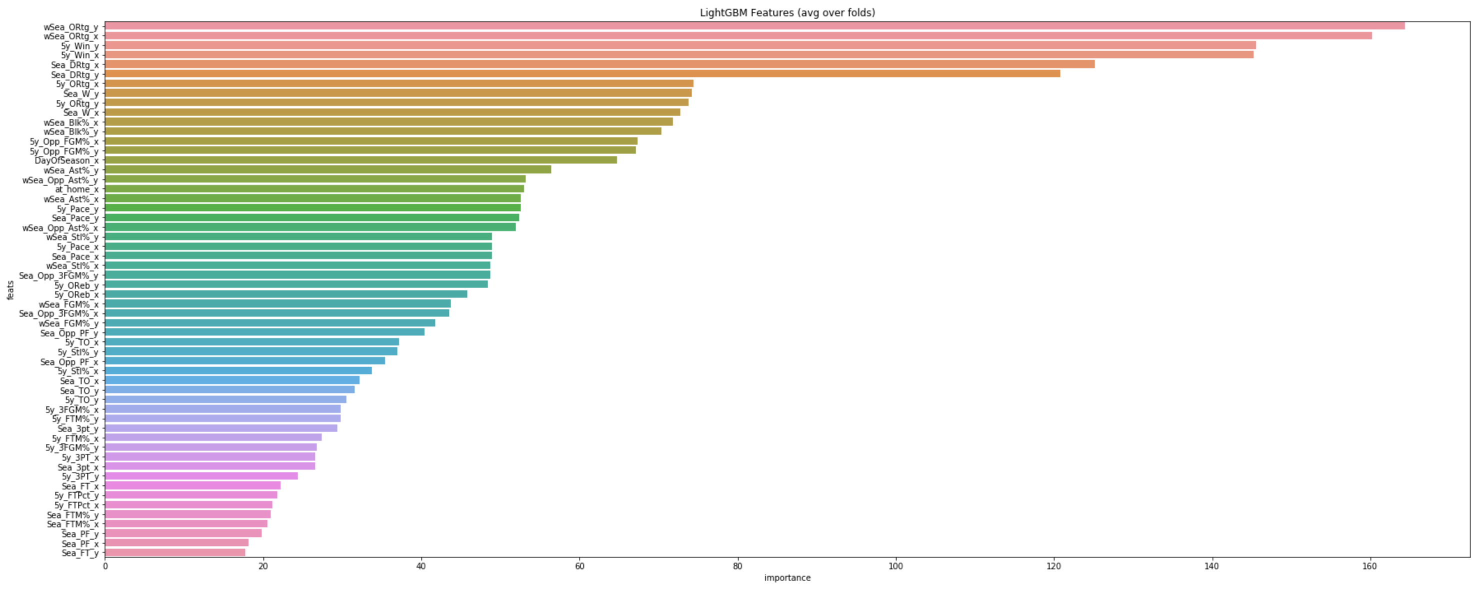
2 12:00 PM - Dec 3, 2019

See Blake Atkinson's other Tweets

TL;DR: Vegas has an opening line error of about 8.48 and a closing line error of about 8.41. On one hand, we're pretty far off. That's to be expected. If a linear model would beat Vegas, I wouldn't be writing Medium articles. I'd be off the coast of Italy probably. On the other hand, I do think it shows the power of simple linear regression to already come within .75 points of Vegas on average. This also demonstrates the power of the LightGBM. It shaves off a tenth of a point of error with similar speed as the linear regressor.

Feature Selection

Let's take a closer look at what our features are contributing, using the laziest method I mentioned above:

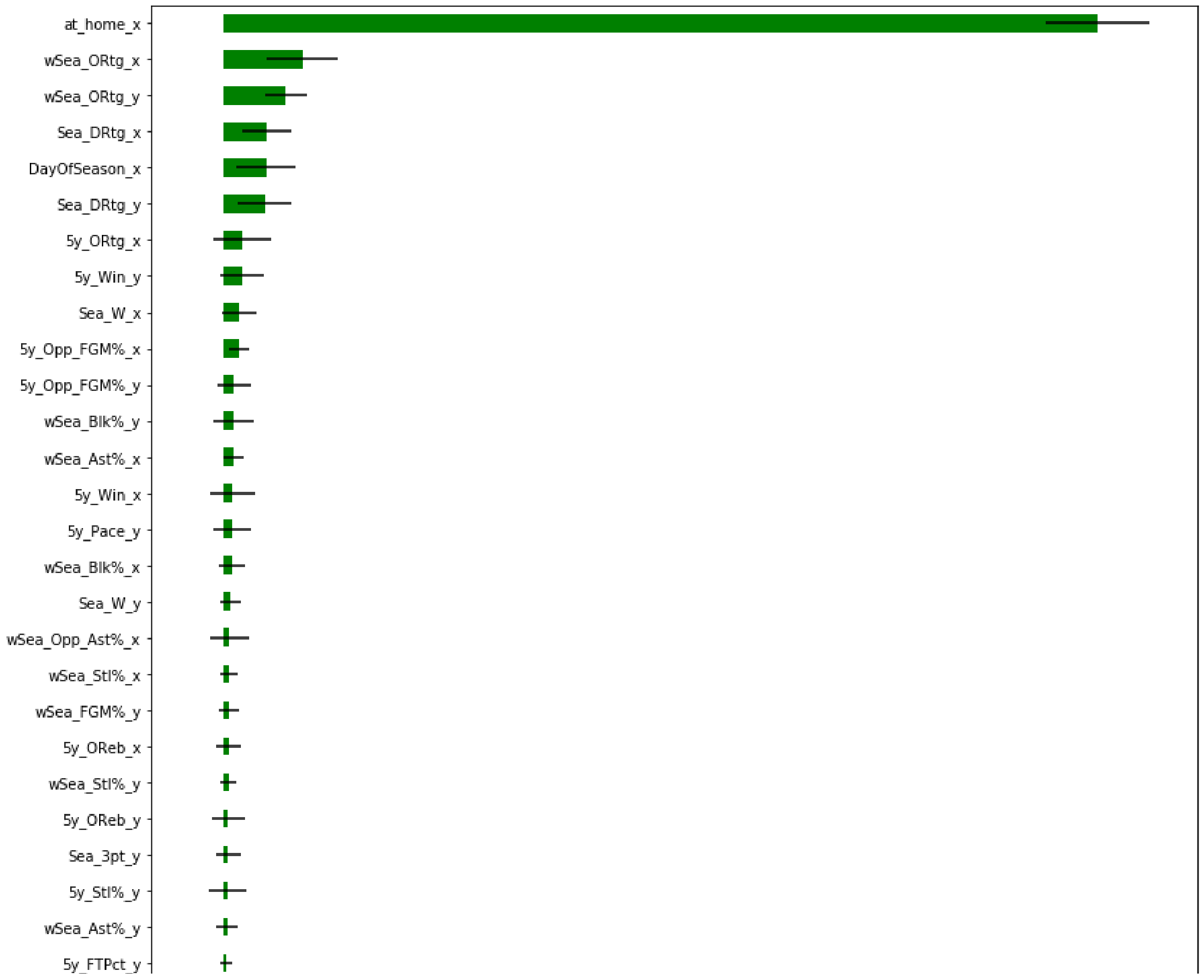


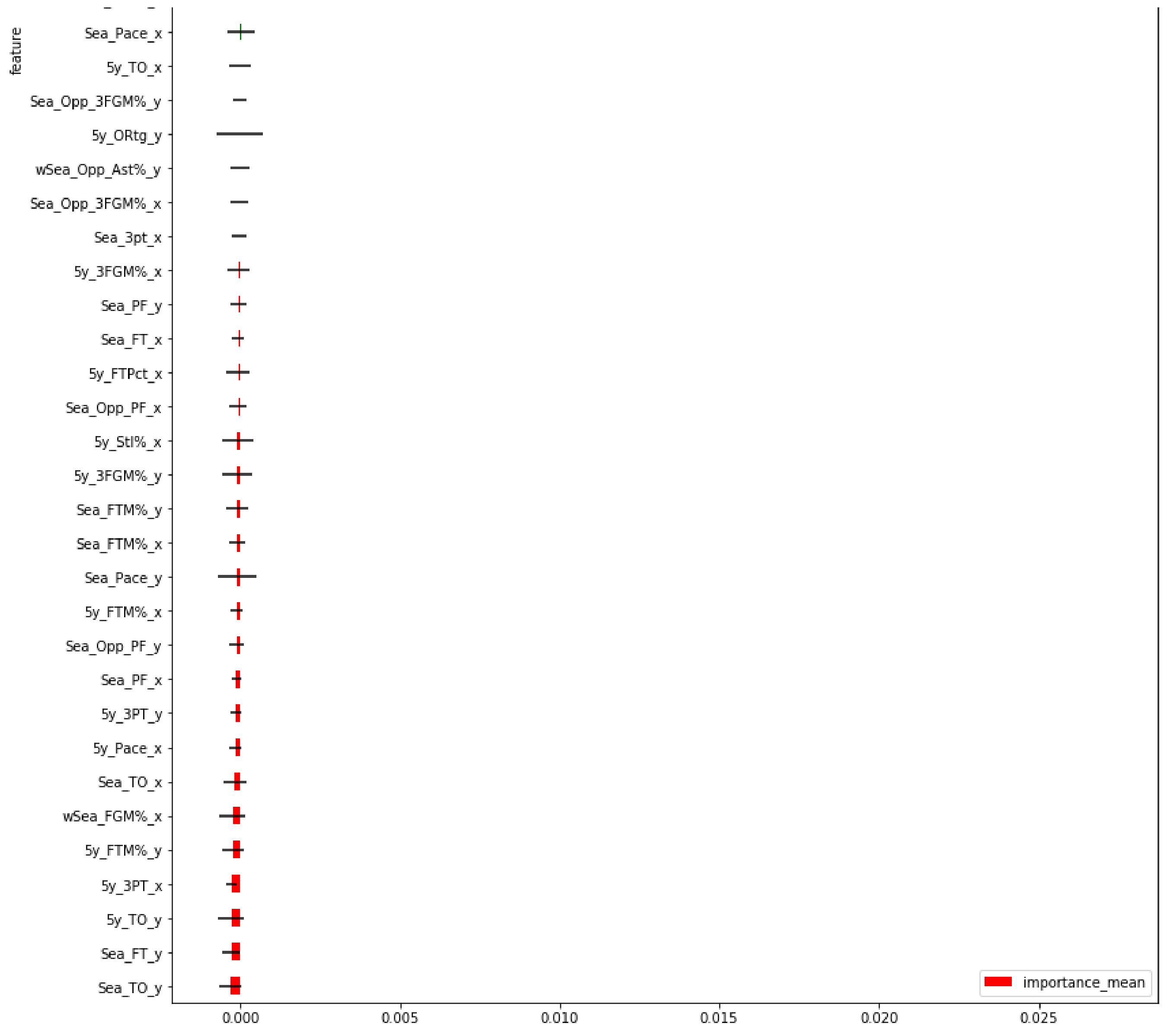
LGBM Feature Importance, using the 'gain' setting

I apologize, the above y-axis is kind of hard to read since there are over 50 features. The two sets of features that the LGBM model loves are weighted season offensive rating and five year weighted win percentage. It also really likes season-long defensive rating. This makes sense to those familiar with basketball.

One thing to note is that every feature contributes a little bit. This isn't always the case. Sometimes there are obvious features from this graph that don't contribute anything. If I make a column of random noise, it's going to show up as useless (or else we have a problem). I've also accidentally given an LGBM two copies of the same column — in that case one of the pair will show up as having not contributed anything. Those sorts of features are safe to drop.

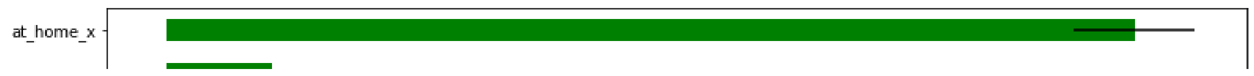
Earlier I referenced drawbacks to only using LGBM gain. It's not necessarily safe to start dropping the features that show up at the bottom of the gain graph. "At home" is one of the most clear-cut, useful features there is. However it's in the middle of the gain graph. Sometimes features can be collinear, and it can confuse or even hinder the model. "Season free throw percentage" is very similar to "Percentage of points from free throws on the season" — i.e., teams that do one well probably do the other well. The model might have trouble differentiating between which one to use. Let's see if the LOFO method can help us. Below is a graph of importance using the LOFO method:

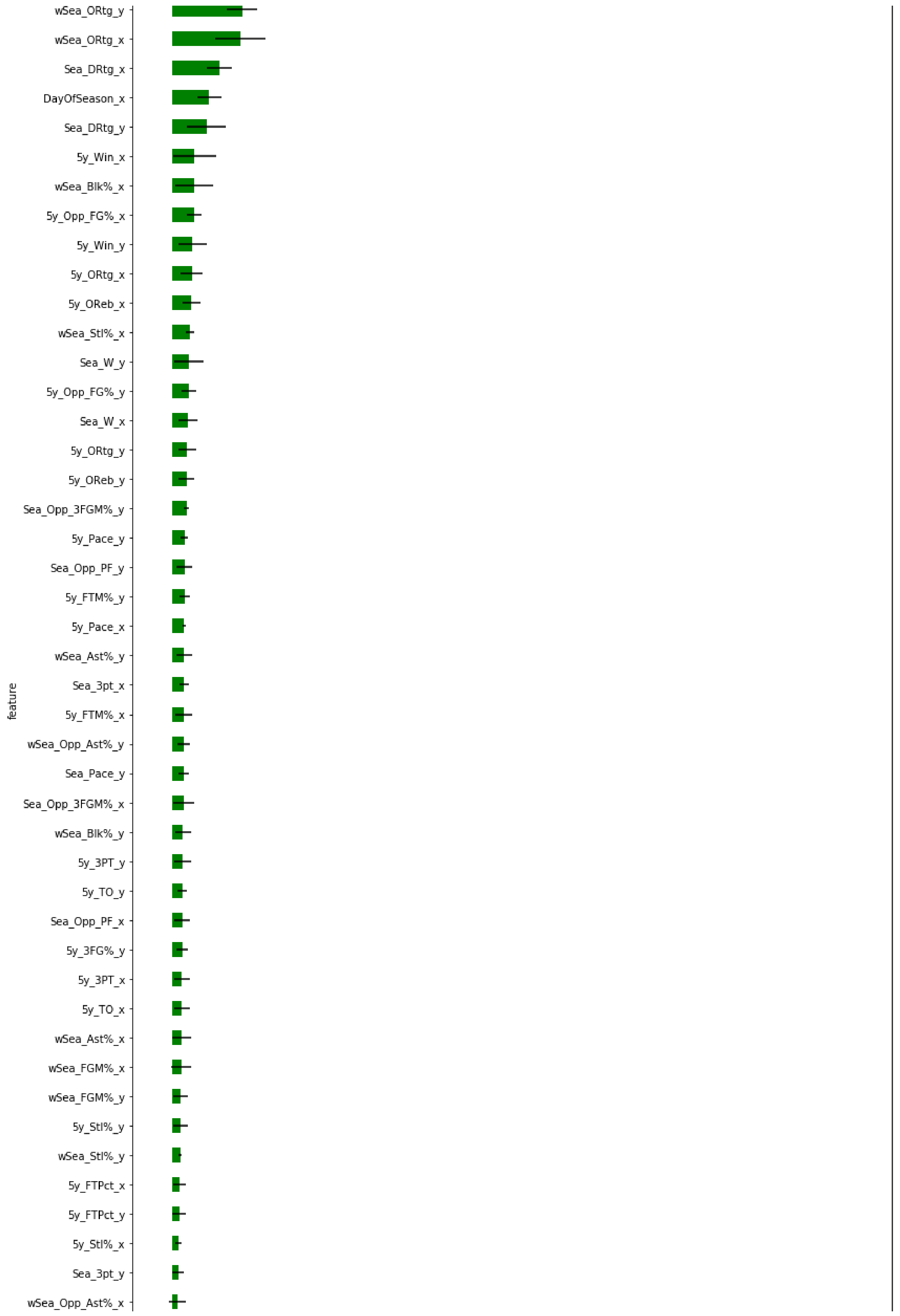


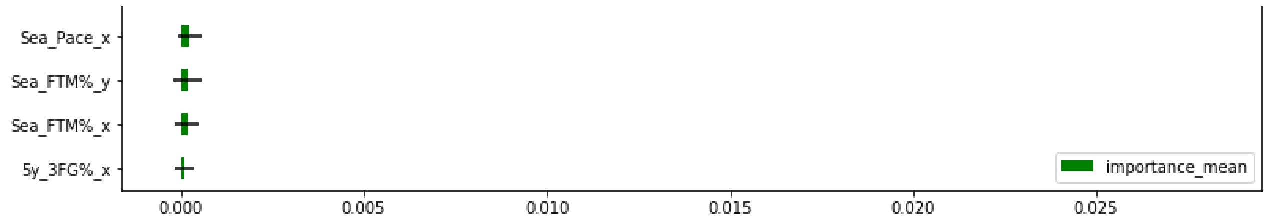


LOFO highlights "At home" — because it's telling us something no other feature is!

That's a lot of red! The author of the repo states that it's safe to drop any red features. In this case however, the error bars extend into green territory. There's probably only two certain drops — Season FT%, and Season TO%. Those are both features I added to demonstrate my feature selection process. You'll notice that some of the red features are similar to the ones we added. For example, five year FT% and five year TO%. Those might be red because I added collinear features. Lastly, season average personal fouls is only slightly red, but definitely isn't helping. I wouldn't necessarily drop this feature in this batch, but it's also last on the LGBM gain graph. Therefore I think it's safe to drop too. Let's try again, this time with only the 50 original features:




feature
wSea_ORtg_y
wSea_ORtg_x
Sea_DRtg_x
DayOfSeason_x
Sea_DRtg_y
5y_Win_x
wSea_Blk%_x
5y_Opp_FG%_x
5y_Win_y
5y_ORtg_x
5y_OReb_x
wSea_Stl%_x
Sea_W_y
5y_Opp_FG%_y
Sea_W_x
5y_ORtg_y
5y_OReb_y
Sea_Opp_3FGM%_y
5y_Pace_y
Sea_Opp_PF_y
5y_FTM%_y
5y_Pace_x
wSea_Ast%_y
Sea_3pt_x
5y_FTM%_x
wSea_Opp_Ast%_y
Sea_Pace_y
Sea_Opp_3FGM%_x
wSea_Blk%_y
5y_3PT_y
5y_TO_y
Sea_Opp_PF_x
5y_3FG%_y
5y_3PT_x
5y_TO_x
wSea_Ast%_x
wSea_FGM%_x
wSea_FGM%_y
5y_Stl%_y
wSea_Stl%_y
5y_FTPct_x
5y_FTPct_y
5y_Stl%_x
Sea_3pt_y
wSea_Opp_Ast%_x



Much better! All green!

But, did we sacrifice performance? No, both models actually score the exact same with 6 less features:

```
Sklearn's Linear Regressor: 9.17

LightGBM model: 9.09
```

Less data used, and a trivial difference in performance. I'll take that trade every time. Using too many features can result in overfitting, which means your model won't generalize to future basketball games. In fact, I could probably add about 20 features similar to the existing 50 and maybe improve the CV score .02. The question is whether or not that's worth it. To me, it's not in this case.

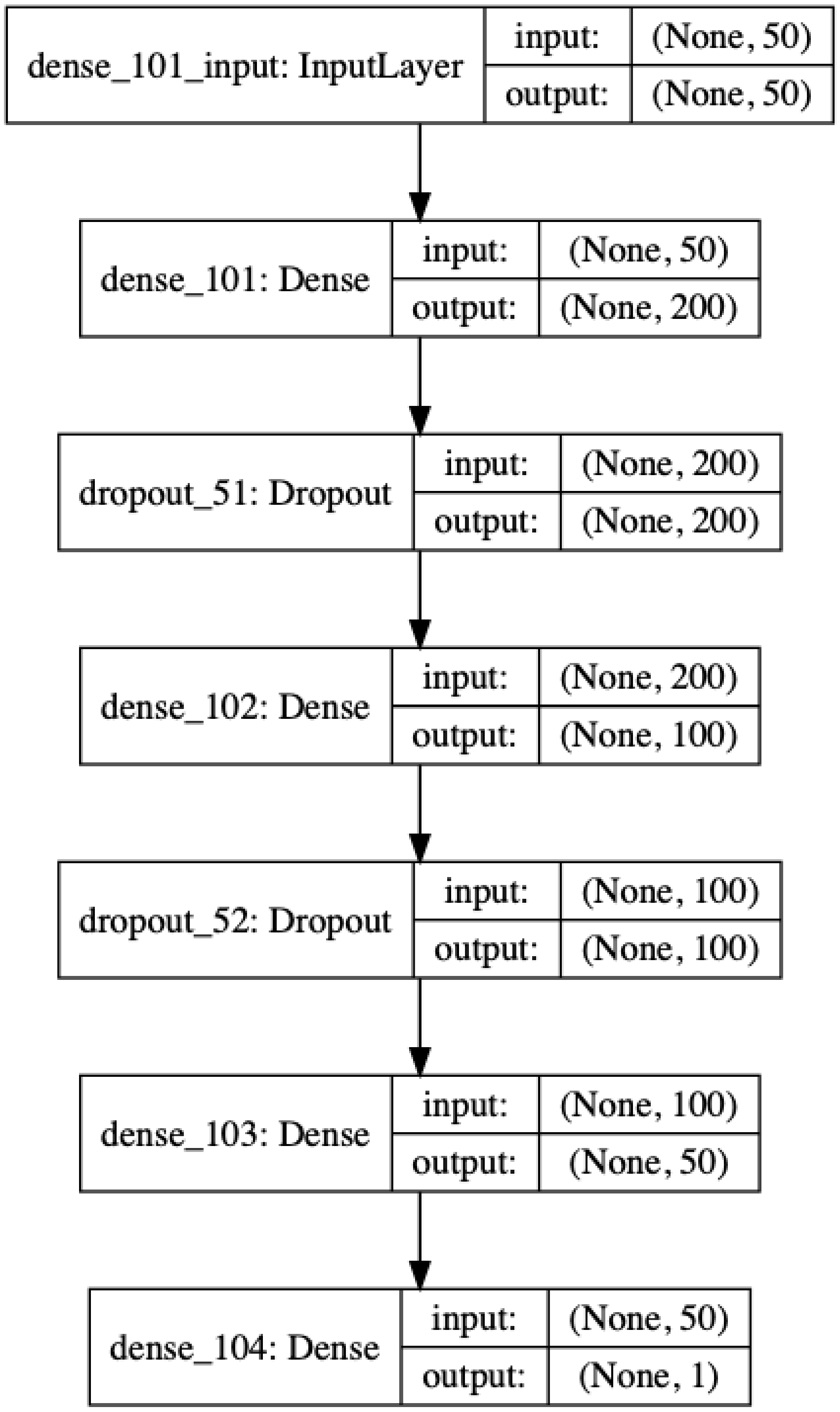
With that being said, there are features that would significantly boost performance. More on that later.

Neural Net

Now that we've cut out features that could lead to overfitting, it's time to give the neural network (NN) a shot. I actually have low expectations here. We're not dealing with a lot of non-linear features, and we can't benefit from neural-network specific layers like convolutions. Therefore a LightGBM is more than sufficient. We've also optimized our feature selection to boost LGBM performance. It would be worthwhile to play with adding and subtracting some features from the NN separately.

In addition to the typical dense neural network layers, I'll utilize dropout. Dropout layers randomly "drop out" a certain percentage of neurons. This prevents over reliance on single neurons, which prevents overfitting and local minima. Since almost all of our features are useful, and we have a basic neural network, we don't need a ton of dropout.

If you create your own network however, I'd recommend playing with dropout values (especially after adding some features). Here's the plain vanilla feed-forward network I'm using:



Using five fold cross validation repeated five times, this took 15 minutes to train! The added complexity and performance isn't free. The final results:

```
Neural Network: 9.22

Sklearn's Linear Regressor: 9.17

LightGBM model: 9.09

Vegas Open: 8.4833

Vegas Close: 8.4110
```

Wait a second, a NN took 15 minutes to train and couldn't beat a linear regressor? That's right. Part of this is due to optimizing features for the linear models. Part of this is also due to the linear nature of our features. Lastly, our neural net could be even deeper, and that would boost performance. Because I love my readers, I tried this model:

```
def get_model():
    x = keras.layers.Input(shape=[X.shape[1]])
    fc1 = keras.layers.Dense(units=450, input_shape=[X.shape[1]])(x)
    act1 = keras.layers.PReLU()(fc1)
    bn1 = keras.layers.BatchNormalization()(act1)
    dp1 = keras.layers.Dropout(0.45)(bn1)
    gn1 = keras.layers.GaussianNoise(0.15)(dp1)
    concat1 = keras.layers.Concatenate()([x, gn1])
    fc2 = keras.layers.Dense(units=600)(concat1)
    act2 = keras.layers.PReLU()(fc2)
    bn2 = keras.layers.BatchNormalization()(act2)
    dp2 = keras.layers.Dropout(0.45)(bn2)
    gn2 = keras.layers.GaussianNoise(0.15)(dp2)
    concat2 = keras.layers.Concatenate()([concat1, gn2])
    fc3 = keras.layers.Dense(units=400)(concat2)
    act3 = keras.layers.PReLU()(fc3)
    bn3 = keras.layers.BatchNormalization()(act3)
    dp3 = keras.layers.Dropout(0.45)(bn3)
    gn3 = keras.layers.GaussianNoise(0.15)(dp3)
    concat3 = keras.layers.Concatenate([concat2, gn3])
    output = keras.layers.Dense(units=1)(concat2)
    model = keras.models.Model(inputs=[x], outputs=[output])
    return model
```

This was a modification of a competitive neural net I found on Kaggle by Bruno Mello. I also added a couple of features I had previously eliminated back into the model. It took over 4 hours (!) to train, and scored 9.05.

Future Direction

Astute readers, or at least those with extensive basketball knowledge, will point out there are very important features that I've left out. Strength of schedule is among the most obvious and most useful. In fact, there's a great explainer on how to incorporate strength of schedule by Alok Pattani and Elissa Lerner. Injuries, recruiting rankings, and coaching changes would also help contribute. Lastly, basic roster features like height or offseason features like "percent of last season's rebounding lost to graduation" could boost performance as well.

I'm very optimistic that using this groundwork and a combination of some of the features above, you could make headway toward that coveted Vegas error value. Will I update this article adding some of those features? Will I share my code if I start approaching Vegas error? No one knows, least of all me.

Machine Learning Python NCAA Basketball Keras Data Science