Predicting NFL Score Differentials

Joseph Habel

habeljw@gmail.com

Overview

The goal of this project is to attempt to predict NFL scoring differentials using publicly available historic game data. I acknowledge that betting point spreads' main goal isn't to necessarily accurately predict the exact outcome of a game, but rather accurately predict a spread that will incentivize an "equal" distribution of bets on both sides. Maximizing profits from setting spreads acts more as a market than a score predictor. However, if I've learned anything, finding good free sources of historical data can be hard. Historical, moderately detailed NFL data however is very easy to come across. The mass adoption of fantasy sports has also driven more advanced game stats to be officially tracked and made widely available.

The goal of this project is to provide a worked example from data collection up and through the modeling process. While the data might not be a perfect source for the project at hand, and with how the football works, score differential might not be such an easy thing to actually predict, the domain allowed me to go through a in-depth worked example without having to sink time into aggregating data from multiple sources. Again the goal of this project is to provide an insight into how I would approach a similar problem in a time frame of a couple weeks.

I realize that you might not be interested in all of the intimate details of this project from top to bottom, so feel free to skip to any of the following sections using the links below. On top of that each section has both a condensed and detailed explanation.

Data Collection

Detailed Explanation

Pro-football-reference

While there do exist some nicely detailed NFL historical data sets out there, (the nflscapR being a really good project for play by play data), this doesn't always tend to be the case for other data sets. As opposed to taking an already made, cleaned, tabular set, I decided to scrape a set off of `https://pro-football-reference.com`. Pro-football-reference is a really nice data aggregator for historical NFL data. For more recent seasons they provide more informational stats such as pass and rush directions, as well as snap count percentages. The site has easily accessible historical data, a robots.txt that allows us to a large majority of their statistics with their permission, and a majority of the data is populated into tables server-side, so we don't have many JavaScript barriers or API calls to reverse engineer to get the data. Pro-football-reference allows us to access data on a team, player, or game basis. Using the individual game data should give us the most detail, but the player sheets provide us a way of not having to parse each table of game stats to curate individual player career data. We'll go ahead and scrape out every game from 2003-2017. 2003 is when the NFL first had a reliable 21 week season, so we'll choose that as a starting point. We'll also get the entire career history of every player who appeared in any of these games.

Scraping Game Data

Let's start with how we can first scrape all of the game data. First off, pro-football-reference provides us nicely named routes to individual weeks from each season. For example if we wanted the games for week 7 of 2009, the route is /years/2009/week_7.htm, with the playoffs and the SuperBowl simply being weeks 18-21 respectively. Unfortunately this easily predictable route schema goes out the window when it comes to the individual games. Clicking on the links to a boxscore will just direct you to a
/boxscores/YYYYMMDD<team abbreviation>.htm route. Two problems with this, the first is that we now need to know the date the games happened on, and the second is that we'll also need to know the home team. From here I decided it's best to just visit every week route between 2003 and 2017, find the all the links to the boxscores on the page, and store those. Then instead of trying to iteratively generate the naming schema for boxscores, we can just visit the stored links. On the initial scrape this will force us to make $21 \times N$ more requests, with N being the number of seasons, but after that, to update any data, it only means a single extra request. We could eliminate the need for these requests if we had the dates of all the games on hand though.

We can nicely wrap those http requests to scrape the game links in parallel. All we have to figure out is how much time we want to give the server between requests, and then we have all of the game links. The problem comes when we access the boxscores. Boxscores nicely have at least 20 tables of data on them depending on the year. This is great for us in terms of getting the data, but this is a lot of data to populate server side for them, hence they populate the tables dynamically with JavaScript on the client side. A simple http request will only serve us the very first two tables, which are the quarterly scoring representation and a scoring table which tells us what the scoring plays were. We really have two options moving forward. The first is to dig into their JavaScript and see if the data is hidden in the initial request somehow, or if they are making API calls and reverse engineer their API to be able to just call for the table data directly. The second is to spawn up some automated browser interfaces, such as phantomjs or selenium, and just let each page render. The first option is the more efficient and salable option, but it also includes the work of trying to figure out the API calls for 20+ tables. The second option isn't very scalable for an initial scrape, but when it comes to updating the data every week, it only comes down to making 16 browser requests. In this case, I opted to go with the second option, just so I could have a dataset at the expense of a few hours of load on my workstation.

To now scrape the data, I used Python in conjunction with the standard requests module, selenium, and the multiprocessing module. The biggest technical challenge is going to be making selenium functional for scraping over 4000 games of data. Normally I would say that running the web driver headless should allow us to not have to worry too much about page load times. This wasn't the case with pro-football-reference, as they serve a pretty substantial amount of ads on their page. When running the full web driver with AdBlock installed, and running the web driver headless (Chrome in this case), headless mode often spent more time in what I would assume is waiting for a myriad of ad supplier requests. I could probably have waited for certain features to be detected and then stop page loading at that point when in headless mode. There also exists a headless driver called Splash that allows you to utilize AdBlock which you can hook in through Python's scrapy. I'm normally more familiar with the selenium environment along with BeautifulSoup for scraping, but I've been looking into Splash since this. What I ended up doing is just loading up N-1, (N being the number of available CPU threads), Chrome web drivers with AdBlock installed.

As each game request took somewhere between 4 and 6 seconds, with 7 available cores, I was able to get all of these pages scraped in about an hour. Looking at scalability, assuming you have a processor with

4 cores and hyper threading, it should take more time for all of the web drivers to spawn up than it would to actually render a week's worth of game pages. This could cause memory issues though as Chrome is notorious for being a memory hog. I've only tested this on my machine and my memory consumption stayed below 40%, but I also do have 24 Gb of memory. On an 8 Gb machine 7 chrome browsers might be an issue.

After scraping the games, I've chosen to output them into individual Excel sheets for my local convenience. Since the data is tabular, it would be just as easy to dump the data into a relational database which would eliminate some of the lazy loading that I've been using. I've simply been generating training data off of the entirety of the data set, but that isn't always ideal. Being able to query over a portion of the data set before it is loaded in would provide the option for smaller and more targeted data sets, without having to load all the data in first to query over it.

The process for scraping the games is included as game_scraper.py. As each table included a two row header, I couldn't use an out of the box parser like pandas' read_html. Initially, I wrote the process with a function for each individual table that I came across on the page. In the name_scraper.py script, I have a more general method for scraping tables on the pro-football-reference site, and this could be implemented instead of all of the individual table parsing functions. If you wish to run this script specifically, it's going to require that you have Chrome's web driver installed and in your local path variable in addition to the required packages.

Scraping Name Data

While the game data is very sufficient, we're going to work off of individually influenced stats as well as team/coach influenced stats. We could take our scraped data and calculate game by game performance for each player, but pro-football-reference has done that work for us. I will say that pro-football-reference does prohibit the /gamelog/ route in their robots.txt.

Now when actively scraping a site, it's important to consider first and foremost a site's robots.txt. So generally when coming across routes in the robots.txt file, I would normally say that those routes are off limits. However according to the Terms and Service, they ask to not "aggressively spider" the site. And that robots are banned from routes in the robots.txt with the exception of utilizing bots that operate in a given time frame that resembles that of a human accessing the site using a standard browser to access the site. Web scraping is complicated to say the least. I tend to err on the side of caution as I would rather not have my IP banned from accessing a site. In this case, I'd recommend using a similar request delay that utilizing selenium caused for scraping the game data, or even possibly the time frame that it would take a human to browse to that specific players game log and click the link that would allow you to import the table into your own sheet. Again there is nothing illegal about violating a robots.txt, but it is a very easy way to get banned from accessing that site, and more often then not, sites include routes in their robots.txt for a reason.

After dealing with the semantics of web scraping, we don't actually have to worry about a JavaScript call in this case. Requesting the individual players gamelog will serve you all of the data initially. Now there's two options for scraping players, we can either opt parse through all of our game sheets and scrape any name that comes up, or when generating training data that takes into account a player's career, we can scrape that player's data then. I've included both methods in the name_scraper.py script. How it's written, scraping all of the players at once is rather inefficient, as it searches for each player's link before scraping. It would be far more efficient to group players by last name, gather links, and then scrape all of the player links at once. It would eliminate a few thousand requests, but it's in place to simulate searching for a name in compliance with the ToS, as the route is disallowed by the robots.txt.

Again all of the requests for the scraping player data are wrapped to run in parallel, and to save to Excel sheets locally. Putting this data into a relational database with each of the players in their own table would definitely speed up processes down the line by not by preventing more lazy loading. At the same time, writing to local sheets allows me to not have to worry about configuring a database locally. As I was the only one accessing my data, using Excel sheets allows me to nicely simulate a database and table format, without forcing me to have to spend time configuring such a set-up.

Moving Forward With Scraping

What I've laid out in the name and game scraper, I'd describe as very initial work. My next steps to better these scrapers would probably be establishing a database structure for them to write to, adding an argument parser to allow these to be CLI tools as opposed just scripts, refactoring the game scraper table parsers into more general methods, and exploring using Splash as an alternative to a non-headless Chrome web driver with AdBlock installed.

Condensed Explanation

We need to scrape both individual game and player data. We're going to use `https://pro-football-reference.com` as it's a comprehensive and easy to navigate site. We're going to start scraping in 2003 since that's the first year that the NFL had a reliable 17 week season, (21 total counting postseason). We're going to pull every table of data that was recorded for each game, and a history of every player's individual game performance. For the game data, we have to use an automated browser to be able to scrape the data since the tables are loaded in dynamically. For the player data we have to scrape their data at a "human speed" to obey the site's robots.txt and Terms of Service. We're currently saving the data to Excel spreadsheets for local convenience, but could easily dump it to some form of database on a server. The code for both scrapers is in the project directory under the names game_scraper.py and name_scraper.py; they scrape the individual game data and player career respectively. To run those scripts you'll need Python 3.5 installed as well as the script dependencies. To run the game_scraper.py script you will also need the Chrome web driver installed and accessible from your path environment variable.

Data Wrangling

Detailed Explanation

Generating Historical Team Data

After scraping the data, by this point we should at least have the Games directory populated with sheets of individual game data. If we don't have the player data, we can opt to scrape that on an as needed basis, at a ToS complaint speed, but that data is conveniently tabular. Our end goal is to be able to take certain contributing game factors and attribute their significance to either the players or the team/coaching staff as a whole. The player data is already in a form that is convenient for those manipulations, but the game data isn't. We're going to need to utilize the game by game data we have to generate cumulative team stats. We could probably utilize the team data that pro-football-reference has, but again, like our reasoning for scraping the data from pro-football-reference to demonstrate a data collection process, we're going to generate our own team data here to demonstrate a data wrangling process.

The stats that we are opting to calculate are going to be defined as defaults in the function calls of the make_team_sheets.py script. The majority of these stats are going to be what I would describe as "allowed" stats. These are generally defensive stats, pass yards allowed, rush yards allowed ect., but some of these allowed stats are from special teams, like punt return yards allowed and kick return yards allowed, so it's not accurate to call these strictly defensive stats. The rest of the offensive stats will still be aggregated by player so we know which player's history to reference. The only team based offensive stats will be offensive fumbles (since for some reason the fumbler has only started being recorded in the last few years), conversion rates, and total amount of first downs.

After getting our desired stats as game by game player and team totals, we can write these to individual team sheets. Again like the scraping process we're going to write these to Excel sheets for local convenience, but these are just as appropriately formatted for a database representation where each team has their own table. Generating the sheets for each individual team is done by running the make_team_sheets.py script. Like the scraping scripts, this script has the ability to utilize all of the processing cores via Python's multiprocessing module. Overall there probably could be some performance upgrades by better handling Pandas' data flow processes, but this script is decently optimized to deal with it's workflow.

Training Representation

We opted to generate team based representations to allow us an easy method to generate training sets. When generating training sets, we're going to separate our data into player influenced and team influenced. The reasoning for this, is it should allow us to better predict performance in the case of a traded player or backup starting. Stats such and rushing attempts we can attribute to coaching calls, and stats such as conversion rates and sacks allowed we can attribute to the offensive line and offense in general as opposed to a single player. However, stats such as catch percent, yards per carry, and interceptions thrown we can attribute more to single player's ability.

Since we're attributing certain influences to individuals as opposed to the whole team, we need to understand how to combine individual performance into a single metric for each game. Let's take the receiving core as example. Receiving cores tend have more players than any other aspect of an NFL team. All star receivers can define a team, but at the same time, other receivers can very little impact overall. Take Percy Howard for example, while he did contribute a touchdown for the Cowboys in Super Bowl X, it was his only reception in his entire career. At the same time receivers such as Odell Beckham Jr, tend to be the defining player of their team. Most teams utilize one Quarterback, maybe two Backs, but their receivers tend not to be as cut and dry. And while we can measure Odell Beckham Jr's success rate, how do we measure how that relates to the Giant's overall performance. What we can do, is take Odell Beckham Jr's career performance in stats such as Yards per Catch, and Catch Percentage, and weight that with respect to how often he is a passing target. So if OBJ gets targeted 60% of the time, we can attribute 60% of the team's receiving stats to him. We can extend this analogy to the rest of the positions.

Since we're calculating both team influenced statistics, as well as individual influenced statistics, we're going to have two different processes for generating training features. If we're weighting performance based on overall team performance we can access a past N games for the team as a whole, and if we're weighting performance based on an individual's ability we we access a past M games for the player. We'll weight player determined stats from a player's career performance over the past M games, with the player's weight being determined by the amount of plays the player the player contributed over the team's past N games. We'll pull the player's performance from the individual player sheets in the Players directory, and the team based performance from the Teams directory. For each team we'll calculate the stats specified in the make_training.py script, as well as gathering the actual point differential and the Vegas line for that particular game. Currently the make_training.py script is set to calculate 45 feature variables for each team, totaling 90 feature variables. The training set can then be specified as to how far the team data should be taken back, how far back the player's career data should be accessed, and whether or not we wish to include playoff performance.

Condensed Explanation

While we've scraped what we need for both game and individual player performance data, the current representations aren't in the most useful states. The historical player data is in a nice tabular form, but the game data is spread over at least twenty tables for each game. We're going to opt to represent the data by separate performance statistics in two categories, player influenced and team/coaching influenced statistics. Stats such as field goal made percentage, rushing yards per carry, and catch percentage, are stats that we'll attribute to individual players. Stats such as 3rd down conversion rate, total number of passing attempts, as well as the majority of defensive stats we can attribute to either overall team performance or coaching decisions. The player influenced statistics will then be represented as the weighted average of every player's career data who contributed to that position over the specified history of team games. So if a running back made 40% of rushing attempts over the team search period, his career running stats would attribute 40% to the team's rushing stats. Breaking up individual and team influenced stats should allow us to not have to worry as much about how injuries and trades contribute back to performance. Aggregating the data into team stats takes place in the make_team_sheets.py script, and generating the training data sets takes place in the make_training.py script.

Data Modeling

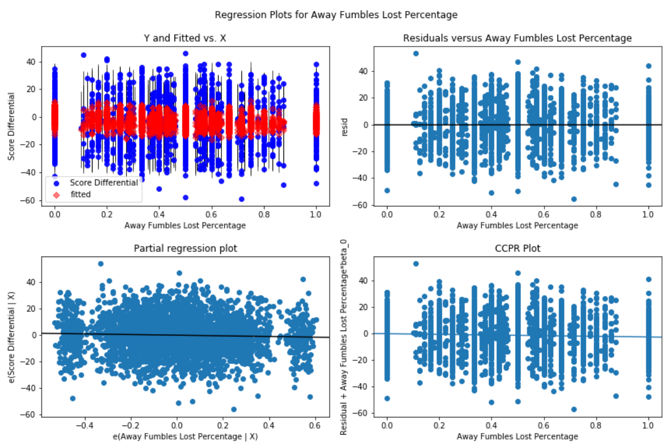
Detailed Explanation

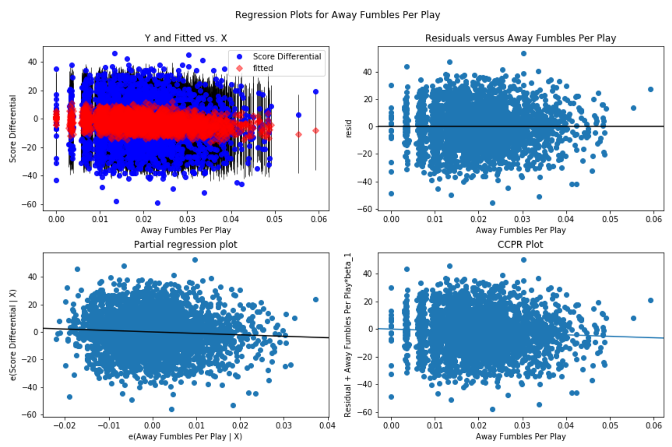
Using Regression

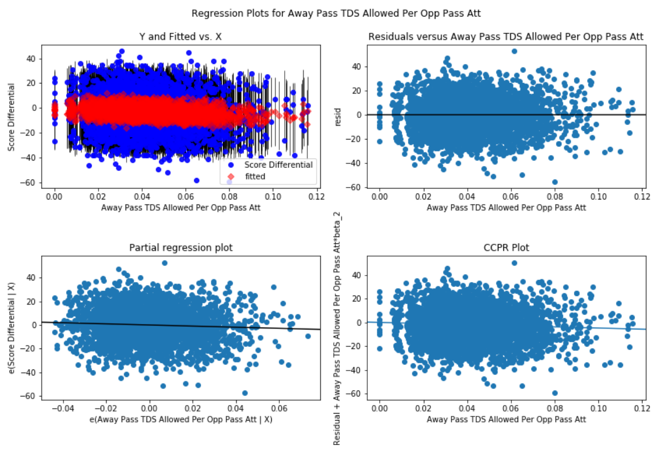
We're going to attempt to use regression to predict what the scoring differential will be. If we wanted to give a binary answer, such as whether a team will beat a specified spread or simply if a team will win, we could use classification models. Since we'd rather try to specify our own spread representations, a regression might be better suited for this case. Although a regression isn't going to be a perfect solution, as scoring differential isn't a completely continuous variable. The issue comes down to measuring how "close" a spread prediction is. With sports such as baseball, soccer, and hockey, it's easy since each score is strictly one. So in principle losing by 8 points in those sports is much worse than losing by 2. Basketball also doesn't suffer as much from this problem. Even though Basketball as 1, 2, and 3 point scoring opportunities, the scores are much higher multiples of these scoring intervals that it doesn't make as much of a difference when looking at differentials. Football's scoring however is more erratic. A 1 point loss in football is a close game, but is it hard to argue a last minute touchdown that gives a 7 point loss still also isn't just as close of a game. When using a regression model, since we have to assume our predictor is a continuous variable, a prediction being off by 7 points would be seen as significantly worse than one being 2 off. So while we're going to use regression to attempt to predict point spreads, we're going to start to see errors that feel relatively high, and these high errors can contribute to modeling us further from the actual differential.

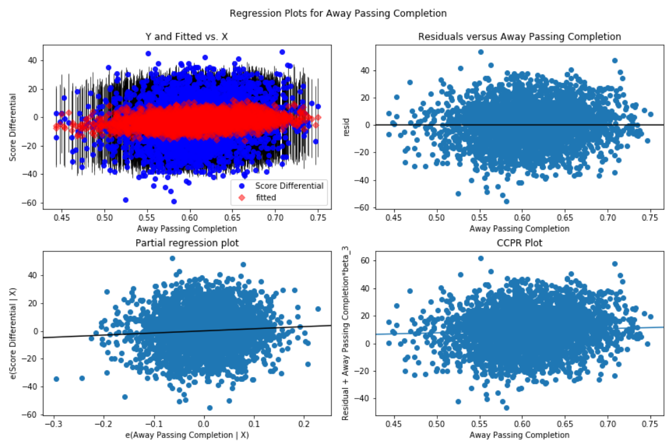
Ordinary Least Squares Regression Model

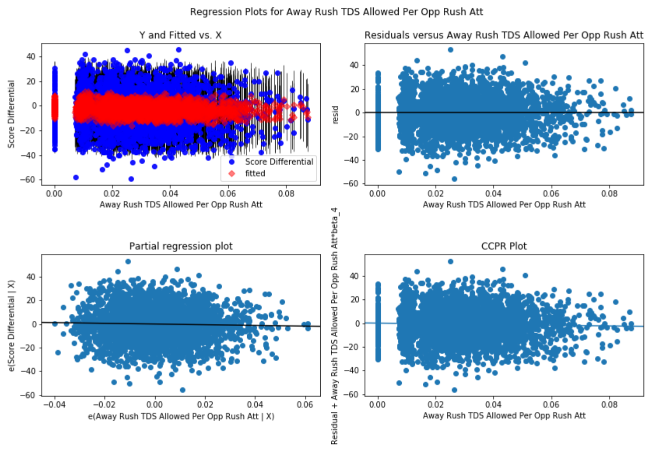
We're first going to attempt an ordinary least squares model as a starting point. OLS has the advantages of being easy to interpret and explain, and often proves to be a rather robust starting point. In this case we're going to first model our entire training set. This training set can be found under the Training directory. After we model the entire set, we're going to then use the significance values of each of the features to eliminate features that don't appear to be significant to the model. This training set has features that are going to have some collinearity with each other. Highly collinear features should be avoided in modeling using OLS. We can look for collinearity via the eigenvalues of the correlation matrix of our features. If any of the eigenvalues of the correlation matrix are close to zero, then we know there exists collinearity in our data. We can look at each of the corresponding eigenvectors of the near zero eigenvalues to figure out which features appear to be collinear.

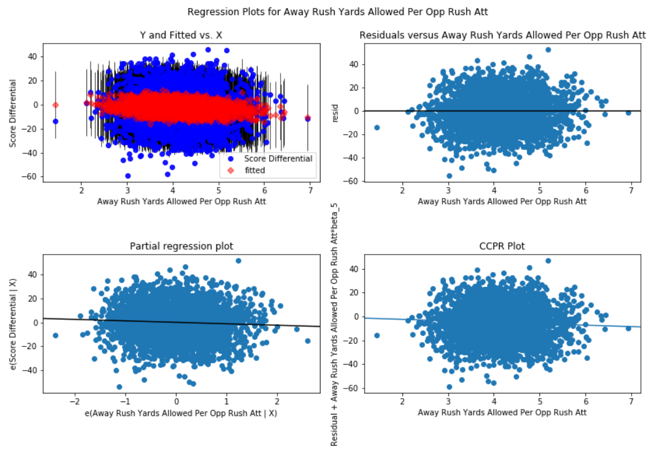

Regression Plots for Away Fumbles Lost Percentage
Y and Fitted vs. X
Residuals versus Away Fumbles Lost Percentage
Partial regression plot
CCPR Plot
Score Differential
fitted
Away Fumbles Lost Percentage
resid
e(Score Differential | X)
e(Away Fumbles Lost Percentage | X)
Residual + Away Fumbles Lost Percentage*beta_0

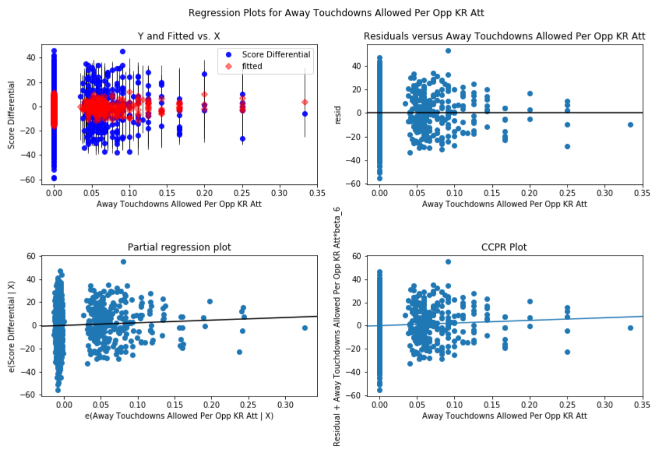

Regression Plots for Away Fumbles Per Play
Y and Fitted vs. X
Residuals versus Away Fumbles Per Play
Partial regression plot
CCPR Plot
Score Differential
fitted
Away Fumbles Per Play
resid
e(Score Differential | X)
e(Away Fumbles Per Play | X)
Residual + Away Fumbles Per Play*beta_1

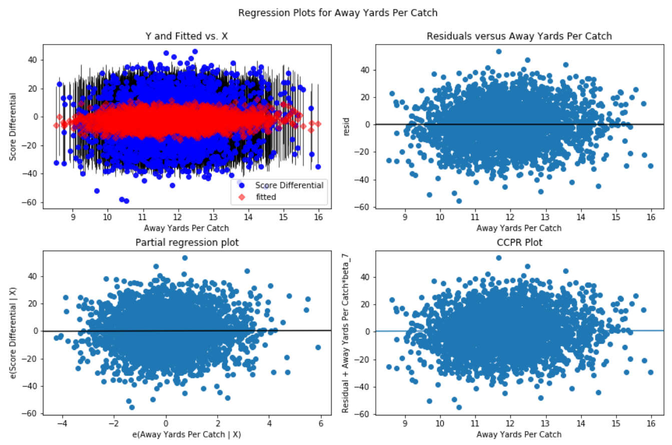

Regression Plots for Away Pass TDS Allowed Per Opp Pass Att
Y and Fitted vs. X
Residuals versus Away Pass TDS Allowed Per Opp Pass Att
Partial regression plot
CCPR Plot
Score Differential
fitted
Away Pass TDS Allowed Per Opp Pass Att
resid
e(Score Differential | X)
e(Away Pass TDS Allowed Per Opp Pass Att | X)
Residual + Away Pass TDS Allowed Per Opp Pass Att*beta_2

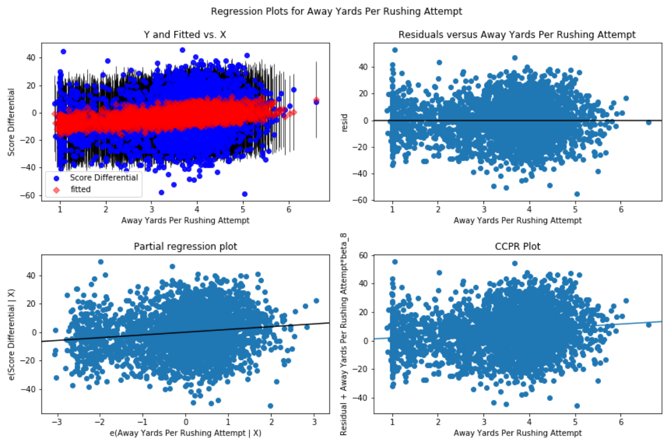

Regression Plots for Away Passing Completion
Y and Fitted vs. X
Residuals versus Away Passing Completion
Partial regression plot
CCPR Plot
Score Differential
fitted
Away Passing Completion
resid
e(Score Differential | X)
e(Away Passing Completion | X)
Residual + Away Passing Completion*beta_3

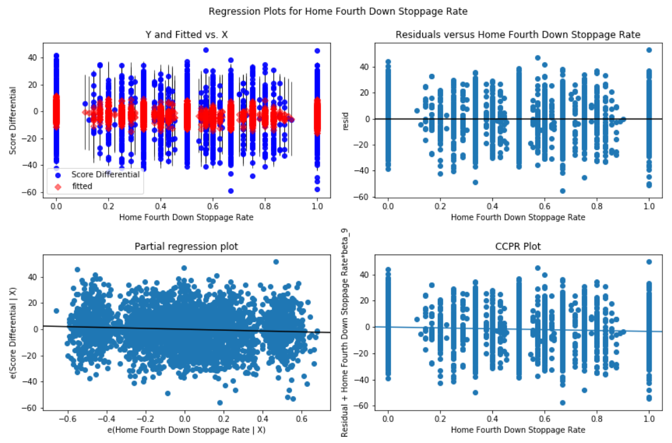

Regression Plots for Away Rush TDS Allowed Per Opp Rush Att
Y and Fitted vs. X
Residuals versus Away Rush TDS Allowed Per Opp Rush Att
Partial regression plot
CCPR Plot
Score Differential
fitted
Away Rush TDS Allowed Per Opp Rush Att
resid
e(Score Differential | X)
e(Away Rush TDS Allowed Per Opp Rush Att | X)
Residual + Away Rush TDS Allowed Per Opp Rush Att*beta_4

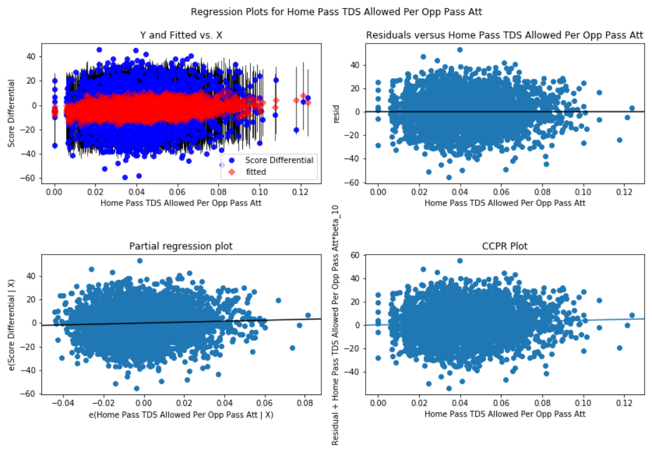

Regression Plots for Away Rush Yards Allowed Per Opp Rush Att
Y and Fitted vs. X
Residuals versus Away Rush Yards Allowed Per Opp Rush Att
Partial regression plot
CCPR Plot
Score Differential
fitted
Away Rush Yards Allowed Per Opp Rush Att
resid
e(Score Differential | X)
e(Away Rush Yards Allowed Per Opp Rush Att | X)
Residual + Away Rush Yards Allowed Per Opp Rush Att*beta_5

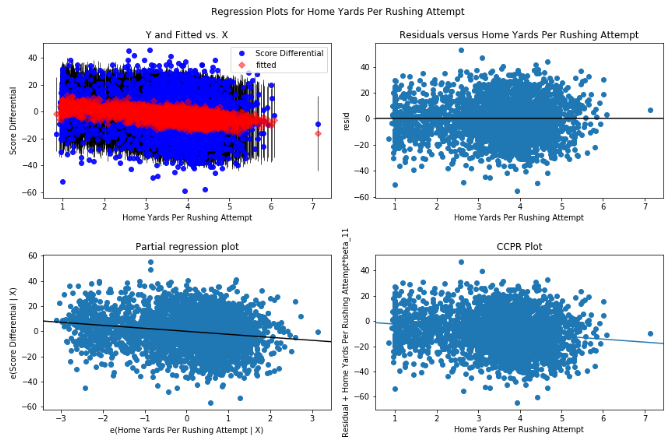

Regression Plots for Away Touchdowns Allowed Per Opp KR Att
Y and Fitted vs. X
Score Differential
fitted
Away Touchdowns Allowed Per Opp KR Att
Residuals versus Away Touchdowns Allowed Per Opp KR Att
resid
Partial regression plot
e(Score Differential | X)
e(Away Touchdowns Allowed Per Opp KR Att | X)
CCPR Plot
Residual + Away Touchdowns Allowed Per Opp KR Att*beta_6


Regression Plots for Away Yards Per Catch
Y and Fitted vs. X
Score Differential
fitted
Away Yards Per Catch
Residuals versus Away Yards Per Catch
resid
Partial regression plot
e(Score Differential | X)
e(Away Yards Per Catch | X)
CCPR Plot
Residual + Away Yards Per Catch*beta_7


Regression Plots for Away Yards Per Rushing Attempt
Y and Fitted vs. X
Score Differential
fitted
Away Yards Per Rushing Attempt
Residuals versus Away Yards Per Rushing Attempt
resid
Partial regression plot
e(Score Differential | X)
e(Away Yards Per Rushing Attempt | X)
CCPR Plot
Residual + Away Yards Per Rushing Attempt*beta_8


Regression Plots for Home Fourth Down Stoppage Rate
Y and Fitted vs. X
Score Differential
fitted
Home Fourth Down Stoppage Rate
Residuals versus Home Fourth Down Stoppage Rate
resid
Partial regression plot
e(Score Differential | X)
e(Home Fourth Down Stoppage Rate | X)
CCPR Plot
Residual + Home Fourth Down Stoppage Rate*beta_9


Regression Plots for Home Pass TDS Allowed Per Opp Pass Att
Y and Fitted vs. X
Score Differential
fitted
Home Pass TDS Allowed Per Opp Pass Att
Residuals versus Home Pass TDS Allowed Per Opp Pass Att
resid
Partial regression plot
e(Score Differential | X)
e(Home Pass TDS Allowed Per Opp Pass Att | X)
CCPR Plot
Residual + Home Pass TDS Allowed Per Opp Pass Att*beta_10


Regression Plots for Home Yards Per Rushing Attempt
Y and Fitted vs. X
Score Differential
fitted
Home Yards Per Rushing Attempt
Residuals versus Home Yards Per Rushing Attempt
resid
Partial regression plot
e(Score Differential | X)
e(Home Yards Per Rushing Attempt | X)
CCPR Plot
Residual + Home Yards Per Rushing Attempt*beta_11

After we filter out any collinear features, and select features with a p-values below our given significance level, we can plot individual regression plots of the features used in predicting the model, along with residuals plots for these features. Ideally residuals should be randomly distributed in a circular pattern and should be evenly distributed above and below the x axis. Looking at our residual plots we can see that there are a features that highlight some potential issues. With the Away Fumbles Lost Percentage and Home Fourth Down Stoppage Percent, we see some issues where our X values aren't evenly distributed throughout and are heavily represented at 0 and ± 1. On top of that Away Touchdowns Allowed Per Opponent Kick Return Attempt, Away Rush TDs Allowed Per Opponent Rush Attempt, Away Yards Per Rushing Attempt, and Home Yards Per Rushing Attempt all have the minimum X value over represented.

After filtering out the features with extremely skewed X distributions (Away Fumbles Lost Percentage, Home Fourth Down Stoppage Percent, and Away Touchdowns Allowed Per Opponent Kick Return) we arrive with a final model. The full summary of this model is available under the Model Results directory under the name of OLS.txt. We're going to use the Vegas lines given from pro-football-reference as a baseline to compare our model against. In this case we're going to calculate and compare the root mean square error (RMSE), mean absolute error (MAE), and the spread accuracy.

Error	OLS Model	Baseline
RMSE	13.777	13.360
MAE	10.989	10.445
Spread Accuracy	52.49%	50.61%

In the case of this OLS model, our RMSE and MAE are both worse than the baseline by about a half a point, and the spread accuracy performs slightly better than the baseline. While I wouldn't argue that this can serve as a be all end all model, it's ability to set spreads of similar performance to our baseline is surprisingly close for a simple linear regression. The script that we used to generate the OLS model is ols_model.py

Augmenting the Data

Data Augmentation is a process that should allow us to get more data from the 4000 games of data that we currently have. Some methods of data augmentation include adding random noise around features, or in the case of image classification problems, applying affine transformations to the image. Since what we currently have in our training representation is an equal number of features for both teams, we can take notes from the image recognition book and "reflect" our data. By this I mean we can swap the home and away values for each of the corresponding statistics and then negate our scoring differential. This will in term give us twice as many unique data points to work with.

Random Forest Regression Model

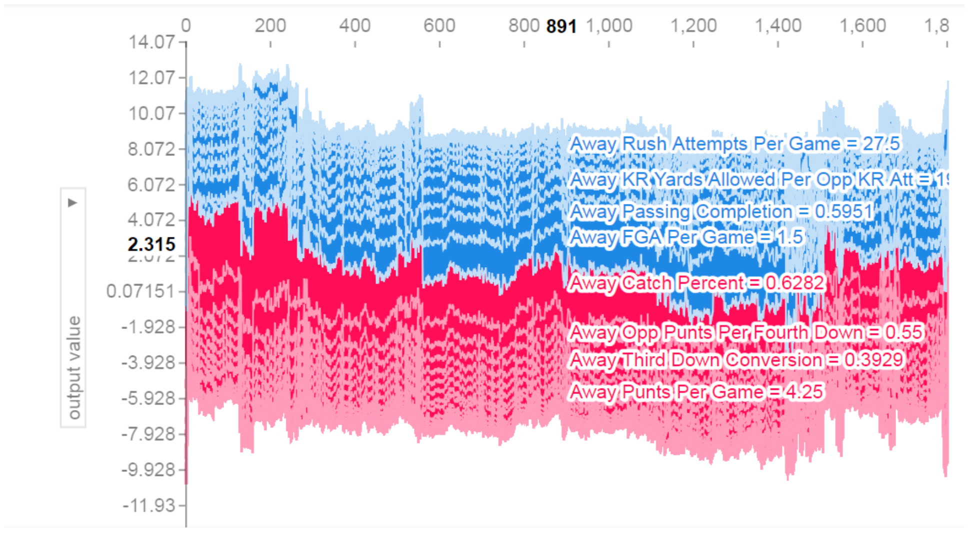
Random Forest is a nice starter for introducing machine learning into predictions. Off the shelf we only really have one hyper-parameter to tune, which would be the number of estimators, and that itself can be a relatively forgiving parameter. Some other off the shelf starting models to consider would be a support vector machine or a gradient boosting model. SVMs and GBMs tend to slightly outperform RFs, but they have more hyper-parameters to worry about tuning. While machine learning is often tagged as a black-box, there exists a framework for better understanding how ensemble based models (SVMs, GBMs, RFs, as well as others) make their predictions. Python provides a nice module for utilizing these SHAP values for explaining machine learning models. Here is the link to the module's github page, as well as the link to the original NIPS paper.

We'll first load in the data and augment it by mirroring the home and away values and negating the scoring differential. From here we'll fit a baseline model that encompasses all all of the features. In the case of a random forest model, we can tune the number of estimators hyper-parameter. It's helpful to have more estimators than the number of features. As the number of estimators scales though, as will the model's complexity, and hence require an increasing amount of time. I cycled through various values ranging from 100 to 10000, and found that 1500 seemed to perform the best while still not being too computationally expensive.

After having a base model fit, it's important to begin to eliminate features that are rather insignificant to the model's performance. Using a linear regression we were able to find the significance value via a t-test. The random forest however is not a linear model, so we have to find another way of measuring the significance of a feature to the model. There are two types of importance we can use for this, scikit-learn comes with a built in importance attribute that measures what percentage of that feature ends up at the end of the decision tree. There is another method that gives a better idea of what influence a feature has on a model though. We can randomize each feature and observe how much it changes the model's error. The more the error is affected, the more important the feature is to the model. We can measure all of these importances, and then recursively eliminate the cumulatively smallest $n\%$ of features, until we optimize the model. With respect to our base model, our optimal feature arrangement was eliminating the cumulatively smallest 10% of the feature importances. This left us with 50 features. We can access the scripts used to build the regression model in RandomForest.py. Our corresponding metrics for this model are as follows:

Error	Random Forest Model	Baseline
RMSE	14.292	13.360
MAE	11.127	10.445
Spread Accuracy	51.47%	50.61%

The problem is while we have our error estimates for this decision tree, the question arises why. With the OLS model we can look at the coefficients of each of the features to understand how much impact each feature has on the prediction. With a random forest we don't have that luxury. We do have the opportunity to observe SHAP values though. SHAP provides us with an intuitive means to observe how each feature of each training point contributes to that individual prediction. As of now SHAP plots can only be accessed via an IPython notebook. The benefit however is the SHAP module provides hands on visualizations that allow you to explore how each feature variable contributes to the predictions. The following plot is referred to as a force plot. Each point on this force plot represents an individual training point, the red sections are the individual features that are influence the prediction to be higher, and the blue sections are the individual features that influence the prediction to be lower. The larger the section, the more influence that feature has in that direction.



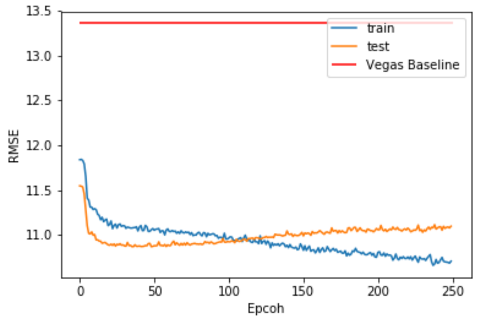
As of now any SHAP plot is built using React and D3 to generate an SVG file, and the project is still rather young. The force plots are generalizable, but not optimized for non ensemble methods. With a webspace to host SVGs however, the force plots become a tool that makes engaging with machine learning predictions more intuitive and understandable. The SHAP plotting is available in the notebook RandomForestShap.ipynb.

Deep Feed Forward Regression Model

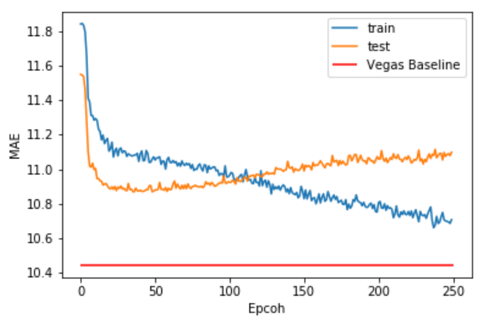
The final model is going to be an attempt at building a deep learning model to predict the scoring differential. Often a deep learning model can provide little advantage over a ensemble based machine learning model, except when there is a true abundance of data. Some instances of this would be imagenet classification problems, natural language processing problems, and potentially a large abundance time series data, think multiple years of intraday stock data. With only a total of around 4000 games, I wouldn't make an argument that this is a good problem for modeling with deep learning. One of the benefits we can find from building a deep learning model is that we're not as restricted to our data being strictly tabular as we were with the random forest. In our case we've been feeding the data through as a single row, while it could also make sense to send the data through as a $N \times 2$ array. Where the first column would be the away team and the second column would be the home team, and each row would be the same corresponding statistic.

The current network has 9 fully connected layers, with two dropout layers, and a layer to flatten out the two dimensional data. It utilizes an Adam optimizer so it can automatically determine learning rate, and is optimized with respect to the mean square error. For each training epoch we'll calculate the training and test's root mean square error (RMSE), mean absolute error (MAE), mean absolute arc-tangent percent error (MAAPE), and the spread accuracy. We utilize MAAPE as a replacement for mean absolute percent error, as it gives as a similar metric for low values, and allows us to handle the actual value being 0. The spread accuracy is how often the predicted score differential would have been a a correct spread for the favored team.

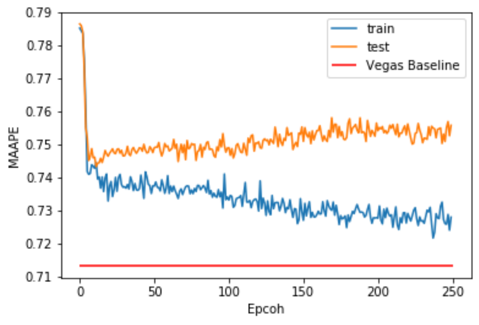
We can then augment our data, and then split it into training and test sets, and observe how it performs. We'll run over 250 epochs to start with a batch size of 40. We can then look at our errors and accuracy with respect to our Epochs to get an idea of how our model is evolving.



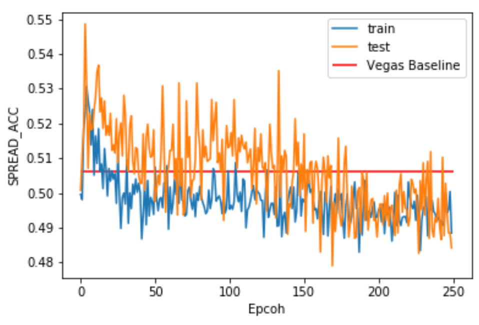
(a) Root Mean Square Error



(b) Mean Absolute Error



(c) Mean Absolute Arctangent Percent Error



(d) Spread Accuracy

We can see that after only about 100 epochs both our test RMSE and MAE begin to grow while our training RMSE and MAE continue to shrink. This is generally a sign that we're over training our model. We can also note that while we do seem to outperform the baseline with respect to RMSE, our performance against the baseline for MAE and MAAPE are worse, and our spread accuracy is pretty volatile. Overall we probably don't have enough data to really harness the power of a deep network for this problem. Similarly to the SHAP analysis we did for our random forest model, we could just as well do a form of SHAP analysis over our Deep Learning model, however, being that the model appears to be over-fitting with not many training epochs, and SHAP isn't well optimized for deep learning applications yet, this feels like a waste of computational resources.

Due to how fast the network overfits, it's hard to identify an appropriate number of epochs to cut the training and and be able to measure how this model stacks up against our baseline. We can see how it stacks up against the baseline as it evolves over the Epochs, but overall I feel like we don't have enough data to make a proper model in this case to draw conclusions. The architecture of the deep learning model is in the NNRegressor.py script.

Condensed Explanation

We've used three different regression models to attempt to predict the scoring differential. Regression doesn't fit predicting an NFL score differential perfectly because of how NFL games happen to be scored. A one point difference and a three point difference are both games that came down to a field goal, but a regression model would weight the three point difference as a larger loss than the one point difference. At the same time we want more than a simple win/loss prediction, and adapting a classifier to distinguish the difference between a 5 and 7 point spread is not a straightforward task, so we'll see how the issues of regression play into our models. We used three different kinds of regression models, an ordinary least squares model, a random forest model, and a deep feed forward neural network. We used the Vegas spread that pro-football-reference records as a baseline to compare our models' performance.

Ordinary Least Squares Model

Our first model choice, an ordinary least squares (OLS) model, was chosen due to it's simplicity to both model and understand. We'll first start by filtering out our data for any collinear relationships, and then fitting a initial model. We'll then set a significance cutoff value and remove any features whose p-values are not less than that cutoff. After this we'll look at the residual plots of each feature to see if there are any issues with using that feature in our OLS prediction. We'll then remove any problematic features based off of the residuals and fit a final model. Using the root mean square error (RMSE), mean absolute error (MAE), and the percent of the time in which the underdog does not beat the spread (spread accuracy), we can compare our final OLS model to the Vegas spread baseline. Our final model is available under the /Model and Results/OLS.txt directory. Against the baseline our model stacks up as such:

Error	OLS Model	Baseline
RMSE	13.777	13.360
MAE	10.989	10.445
Spread Accuracy	52.49%	50.61%

Both the model's RMSE and MAE perform slightly worse than the baseline by about half a point, and the model's spread accuracy slightly out performs the baseline's.

Random Forest Model

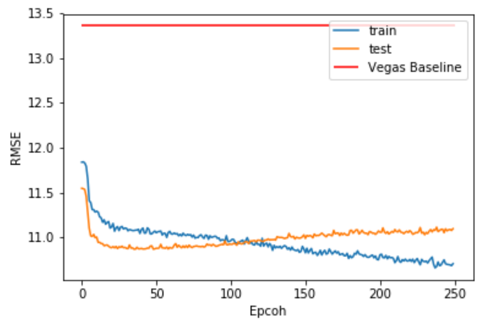
Our second model choice of a random forest was selected for it's robustness as a supervised machine learning model. While machine learning models can prove to be advantageous to linear models, they also can be a little more mysterious on how they arrive at their results. What we can do is fit a random forest model to our full data set, and then recursively select the best features to then fit the model to. Here we randomize each feature and see how that changes our errors, and weight each features importance based upon that. Once we have a final model we can then use a SHAP framework to begin to intuitively understand how the model is reaching its predictions. These SHAP visualizations are interactive and provide a similar intuitive understanding to machine learning models as we would see with an OLS model. Again against our baseline, our model performs as such:

Error	Random Forest Model	Baseline
RMSE	14.292	13.360
MAE	11.127	10.445
Spread Accuracy	51.47%	50.61%

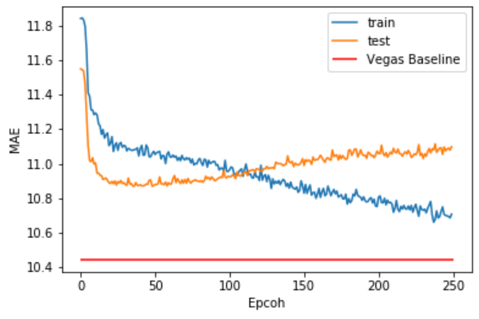
The model's RMSE and MAE perform slightly worse than the baseline by about a point each, and the model's spread accuracy again slightly outperforms the baseline.

Deep Feed Forward Neural Network

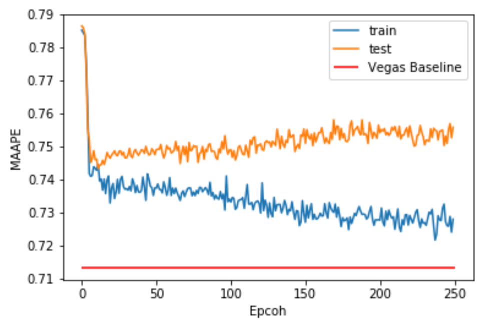
Our third model choice will be a deep feed forward neural network. Deep models can prove to be advantageous when there is an abundance of data at hand. These models tend to be much more computationally expensive, in this case we utilized GPU computing over CPU computing for training, and the results only tend to improve with large amounts of data, think tens of thousands of rows. Just like the random forest model, understanding how models like these reach their predictions can be hard to explain. In the case of our current application the network began to overfit very quickly, and unfortunately doesn't allow us to specify a stable model. The overfitting is very likely related to a lack of data to train a deep network on. Compared to the OLS and Random Forest models however, even with fast overfitting, the deep learning model appears to significantly outperform the other two with respect to the root mean square error, and performs similarly in the MAE, and spread accuracy. We'll compare the values at where the test and training curves appear to intersect against our other models.



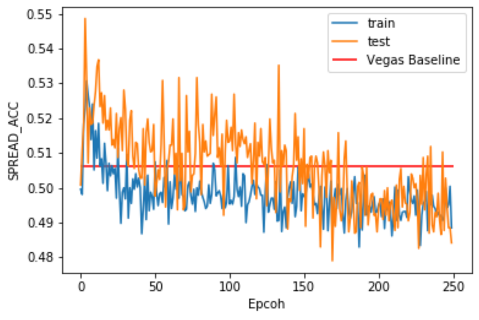
(a) Root Mean Square Error



(b) Mean Absolute Error



(c) Mean Absolute Arctangent Percent Error



(d) Spread Accuracy

Moving Forward With These Models

All of our models had a mean absolute error that hovered around 11 points. We can likely relate this to the fact that football scores are not ideal for regression problems. Even though we had an mean absolute error exceeding 11 points, compared to our baseline, we still performed within 7 percent at the worst. As well as being around the same error metrics, we performed similarly for how often the spread is "correct." Again it's important to note that when setting spreads it should be important to make that accuracy as close to 50 percent as possible, as to incentivize equal betting on both sides and hence hedge house risk. What we could do to potentially improve these models is set up better cutoffs. As of now our predictions are still continuous variables, rounding close games to either a pick or 3 point spread might help reduce our error. On top of that we can pivot our regression model to a classifier to potentially be able to identify advantage bets. These advantage bets would be scenarios where the spreads reached by a market consensus might be overvaluing certain metrics that don't necessarily translate to game performance. It would also be interesting to pivot this player/team influence data representation into sports where regression is a better fit for modeling scoring.