Predicting Over/Under results for MLB games

New York University, Center for Data Science

Authors: Guido Petri, Nathan Griffin, Jonas Peeters, Stephen Roy (Team More Coffee)

Repo: https://github.com/nyu-ds1001/ds1001-project

Abstract

We use several supervised learning models such as RandomForests, Support Vector Machines, Neural Networks, Gradient Boosts, and Logistic Regressions in order to predict the over/unders in MLB games. The data used to train the models was scraped from BaseballReference, OddsPortal, and SportsBookReviewsOnline. In addition to the raw data, we created a vanilla ELO-rating, moving averages, and a pitcher-matchup benchmark. We find that the Neural Network performs best when you want to bet on every single game, but the RandomForest model provides the best accuracy when you restrict betting only to games that meet a specific probability threshold.

Keywords: baseball, betting, over/under, O/U, sports

Business Understanding

In early 2018, the Supreme Court overturned a law from 1992 and made sports betting legal in the US. In our term project, we aim to create a machine learning model that can predict over/under results to MLB games. The purpose behind this is to achieve a positive expected value for a bet in this new market and thus to outperform the gambling public. Since there are only two options on over/under bets, this is a simple binary classification problem.

In order to assess games accurately and create meaningful predictions, we required three types of data: team, player, and global data. Under the umbrella term of "team" falls data such as specific matchup information, as well as aggregate team offensive and defensive stats like OPS, BA, or unearned runs per game. Under "player" data we have a wide array of available data, but we opted to include only pitcher information in order to simplify our model and keep the player with the most outsized effect on a game

contributing to our prediction. Finally, under "global" data we considered variables such as day of the week, weather, or time of day.

In order for us to build such a model, however, there are two fundamental questions that must be answered. The first is how to consider data for a game in order to avoid leakage. We circumvented any sort of information leakage by only considering aggregate data from before the game started. For this purpose, we created 4 datasets with different time frames for the aggregate data - 15, 30, 60, and 90 day rolling averages. The second fundamental question that must be answered is our definition of success. In order to break even against betting fees, it can be calculated that an accuracy of at least 52.4%[1] is required. This means our models must have an accuracy greater than that in order for us to make money.

Data Understanding

We gathered data from three different sources on the web. Baseball data itself and weather data were scraped from BaseballReference[2], while historical bookie odds data was gathered from OddsPortal[3] and SportsBookReviewsOnline[4].

In order to scrape data from BaseballReference, we gathered a list of links to all individual games and for each of these, we used BeautifulSoup4 and re to select specific elements of the page and associate their values with variables. Finding a page element and parsing it was done using a browser's Inspect Element feature as well as a Jupyter Notebook to receive immediate feedback on whether the parsing was working. The matchup data we required was taken from a page header, while the team and pitcher aggregate stats were taken from a summary table for the game. Finally, the weather data was taken from a text section of the HTML.

---

[1] https://medium.com/the-intelligent-sports-wagerer/why-52-4-is-the-most-important-percentage-in-sports-gambling-16ade8003c04

[2] http://baseballreference.com/

[3] https://www.oddsportal.com/baseball/usa/mlb/results/

[4] https://sportsbookreviewsonline.com/scoresoddsarchives/mlb/mlboddsarchives.htm

This parsing of game links is the most computationally intensive part of the scraping process since it gathers data from several different sections of the HTML code. This is where we used multiprocessing to parallelize the link scraping. We also leveraged a separate process to save the link information so that we wouldn't have any race conditions with file writes.

A possible improvement to this approach would be to verify whether each link has already been downloaded and parsed. This would eliminate double-parsing of a page (stemming from the game being listed on both the home and the away team's pages), which would improve the runtime by approximately half. The caveat to this is that the set manipulation would have to be done carefully since one of the parallel processes might be halfway through parsing a page when a process checks if the page has already been parsed.

In total, we scraped 162 games per year, for 30 teams, for 20 years (from 2000 to 2019), which yielded 97200 rows of data. The scraping took approximately two hours on a 12-core machine with a 200Mbit/s download ethernet connection.

For betting odds, two different datasets were gathered. The first dataset scraped from OddsPortal was ultimately discarded due to a lack of sufficient over/under information. While the general scraping process has already been covered, OddsPortal required some unique reverse engineering to decode the actual betting lines which were obfuscated with a substitution cipher. The second dataset, pulled from SportsBookReviewsOnline, consisted of 10 excel files and required no scraping. By contrast, this dataset only provided 10 years of game data (2010 to 2019) equating to just under 50000 rows of data. This is the biggest factor for selection bias in our data since we only have access to data from 2010 and later and thus cannot use data from before then to train our models. By only using recent data, we forgo all previous "eras" of baseball such as the deadball era and the steroid era.

Data Preparation

Data Munging

This is a binary classification problem, which means our target variable takes on values of 0 or 1. The former corresponds to *under*, while the latter corresponds to *over*. An *under* occurs when the aggregate final score of a game is under the number provided by oddsmakers, while an *over* is the corresponding opposite. Generating the target variable is thus a simple task: determine whether the sum of the runs scored by each team is greater than the oddsmakers' number. However, there are a couple of things that must be considered. First, in approximately 5% of our data, the number provided by the oddsmakers matched the final score of the game. This results in a 'push': bets are nullified and no profit or loss is realized. Considering our target variable is not defined in this case and its lack of impact on profit or loss, we dropped these instances from our dataset. Second, we confirmed that the distribution of the target variable was close to 50/50. This is important because if the split was overly biased to one side or the other, you could make a profit simply by betting on the more common value.

Before any analysis could begin, the two datasets had to be merged and cleaned. Merging required a unique identifier which was created by combining game date, home team, and game number that accounted for doubleheaders. This required standardizing team names and date/time formats, as well as a lookup function to identify doubleheaders. To further complicate data aggregation, the two data sets recorded game information differently. Starting pitcher and betting data was individually recorded for each home and away team while the weather, matchup, and team information was captured once for each game. Reshaping both data sets one final time enabled a successful merge; as a final step, the 1% of rows that were missing values were then dropped.

With the data sets combined, a couple of domain specific issues were left to adjudicate. If a game is called for weather reasons after the 5th inning finishes, it counts as a completed game with a winner and loser. For an over/under bet, however, it is considered void, since it biases results towards *under*s

since less of the game was played. These instances have been removed by comparing total innings pitched and online box scores. There were also some outliers in oddsmakers' numbers that were removed.

Feature Engineering

In order to take a team's relative strength, we calculated their seasonal ELO rating. The ELO rating system quantifies the proficiency of a team based on that team's track record of wins and losses versus other teams adjusted by their own ELO. In our case, the proficiency of the teams is represented as a number around 1000. In the beginning of each season, every team starts with an ELO of 1000 and future ELOs are calculated using formula (1). Each time the eventual outcome W - win (1) or loss (0) - was different from the expected outcome of W (probability), we corrected the ELO for those teams with an appropriate K factor.

$$ELO_t = ELO_{t-1} + K\big(W_{t-1} - \mathbb{E}(W_{t-1})\big) \tag{1}$$

We can calculate the expected win probability of a team using the logistic function and the ELO difference, as is depicted in formula (2).

$$\mathbb{E}(W_{t-1}^{TEAM_1}) = \frac{1}{1 + 10^{(ELO_{t-1}^{TEAM_2} - ELO_{t-1}^{TEAM_1})/400}} \tag{2}$$

Another concern we had with this data set was leakage, as most of the game data we have is only collected during the game. To circumvent this, we took rolling averages of all game related variables so that the data used to predict the over/under of a given game only used the information from previous games. We opted for the use of 15, 30, 60, and 90 game windows, which yielded 4 different datasets. The formula used for this calculation is shown in (3), with *n* being the number of games and *X* being the variable to be transformed.

$$MovAve_t(n) = \frac{1}{n}\sum_{i=0}^{n-1} X_{t-n+i} \tag{3}$$

Finally, since baseball is a very pitcher-driven game, we used the data we had concerning the starting pitcher in each game to create a pitcher proficiency variable by dividing the mean of that pitcher's ERA over his last 5 games against the same team by that pitcher's overall ERA over his last 30 games.

Modeling and Evaluation

Base Model

The initial model used as the baseline was a Random Forest with all of the hyperparameters set to default. This model, when trained on the entire training set and applied to the test set, yielded an accuracy measure of 47.9% and an AUC of 0.488. Upon reviewing the score on the training set, it was clear that the model had highly overfit, since the train score was 98.4% and the AUC was 0.999. By restricting the min_samples_leaf parameter to a value of 1% of the total training data, the baseline scorer yielded a test score of 52% with an AUC of 0.516, as well as a train score of 60.2% and an AUC of 0.646. We took this as our baseline upon which to improve. To establish whether a model was better than the baseline, we compared AUCs since it is a more generalizable metric. However, we also kept the accuracy in mind, since that is where our break even point of 52.4% is defined at. The selection of the best model type was done using a RandomizedSearchCV scoring on ROC AUC, with cross-validation being done on a custom KFold that split each year from 2010-2018 into 5 folds.

Optimized Random Forest

Random Forest is a great tool for binary classification problems as it fits many unique decision trees and aggregates the results, which acts to reduce the variance of single decision trees. For RandomForest, some optimal hyperparameters became apparent instantly. For instance, *entropy* is a better criterion than *gini* while varying the other parameters. The *max_features* hyperparameter was clearly

better when set to equal *n_features*. The remaining two hyperparameters which were explored through cross-validation were *min_samples_leaf* and *min_samples_split*. All the models yielded accuracies below the break-even point. The best model's ROC curve on the test set, along with accuracy annotations, can be found in the appendix.

Support Vector Machine

The biggest difference between SVM and a classic decision tree is that SVM uses kernels that calculate the separation line in higher dimensions, allowing us to account for non-linear decision boundaries. We cross-validated across Gaussian, linear and polynomial kernels, as well as different values for *gamma* depending on the number of features and their variance. Since SVM is very time-consuming, we reduced the number of features to 10 using PCA in order to decrease the training time of our model. We found that our best SVM model predicts all zeros and yields an AUC of 0.516 and accuracy of 50.5% when using a rolling average of 90 days. The ROC curves, along with accuracy notations, can be found in the appendix.

Neural Network

Neural Networks have a rich, albeit short, history with respect to supervised classification problems. While various neural network implementations exist, the scikit-learn *Multi-layer Perceptron classifier* was chosen to minimize dependencies, simplify integration, and promote "apples to apples" comparisons against the other models being evaluated. Similarly to the SVM model, neural networks generally perform better when the input data is normalized. A scikit-learn MinMaxScaler((-1,1))[5] was chosen over the StandardScaler for regularization since only 3 of the 256 dataset features resembled a normal distribution[6]. The 117 indicator features were tested twice, both using MinMax scaling and left unscaled. The hyperparameters explored through cross-validation include hidden layer size, activation

[5] http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html
[6] https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler

method, solvers for weight optimization, L2 penalty amounts, learning rates, and solver iterations. The top performing neural network model used the 90 day rolling average feature set and achieved an AUC of 0.534 on the test data and an accuracy of 52.2%. This model applies an L2 penalty of 0.001 for regularization and easily avoids overfitting as evidenced by an AUC of only 0.733 on the training data. Complete ROC curves along with accuracy annotations can be found in the appendix.

The major concerns regarding this model are scaling, reproducibility, and interpretability. Since the model relies on scaled inputs (based on just the training data), the MinMaxScaler must be stored along with the model to scale any new input data. Additionally, due to the complexities inherent in neural networks such as drop-out on hidden layers, it is very difficult to reproduce and interpret the model.

Gradient Boosting

The gradient boosting ensemble model is an alternative to random forests that aims to minimize bias instead of minimizing variance. Instead of building out full trees that are highly overfit and then averaging across them like random forests, gradient boosting is based on "weak learners" - small trees that are operated on sequentially instead of in parallel. Since we had 256 columns and gradient boosting considers by default all features for its decisions, we decided to prune the features by selecting only the 30 features that were most valuable. This was done by iteratively training a small cross-validated gradient boosting model on the training data and discarding the least important feature until we had 30 features remaining. The only exception was keeping the year feature in order to cross-validate properly. This process was repeated for each of the datasets in case the most important features differed per dataset.

After selecting the features to be used, we ran our cross-validation on the training data across integer distributions of n_estimators, max_depth, and min_samples_leaf, as well as both exponential and deviance loss. The best model for each dataset was selected according to AUC, but none of the models outperformed the baseline RandomForest. The ROC curves along with accuracy annotations can be found in the appendix.

Logistic Regression

As a simple alternative to all the above models, we can consider the classical logistic regression. This is, at its core, just a linear model that predicts the likelihood of a sample being in a class or not. In order to evaluate this model type, we needed to scale the data in a similar fashion to how we scaled it for the neural networks with MinMaxScaler((-1, 1)). Performance both before and after scaling was compared and, while the scaling did improve the AUC of some of the models, it was not a very significant difference, and it decreased the AUC of other models. In the appendix, we have one example from the 15-day rolling average dataset. Nonetheless, scaling was performed for performance purposes. We ran our cross-validation across an exponential distribution of C with parameter $\lambda = 1$. This yielded models with a best accuracy under the 52.4% threshold required to break even. The AUC plots, along with accuracy annotations, can be found in the appendix.

Finally, we tried a form of feature selection by fitting a Lasso-type model on the data before feeding the selected features to a logistic regression. Unfortunately, the Lasso eliminated all the features from our data, which indicates the L1-regularized linear regression is unable to provide any information better than a random guess.

Results

While most of our models did not reach the 52.4% break even point when predicting every game, something interesting happens when the models only looked at games that had a higher probability of being in one class or the other. For example, the RandomForest model would be profitable if we only bet on games with a higher than 55% probability of being in either class. The trade off is that we would only be betting on 9% of the total possible games, which restricts our total profit margin. A full analysis on these thresholds can be found in graphs in the appendix.

Discussion

Our final models provide useful insights that can directly inform over/under betting decisions on MLB games. If you wish to bet on every game, the Neural Network is the only model that predicts outcomes with positive expected value. However, the RandomForest model provides the best accuracy when restricting the predicted class to tighter thresholds, thus identifying better games to bet on. Ultimately, these models could be pipelined into an ensemble to take advantage of each. Although scikit-learn easily automates ensembling, the actual implementation is less straightforward due to differences in feature scaling for each model. Due to the narrow margins involved, these models are best used to augment rather than automate or completely replace human decision making.

There are several types of risk involved with this strategy. Chief among them is the availability and accuracy of external data. This risk can be mitigated by expanding the scraping process to include additional sources and perform some automated validation to quality check our inputs. Separate from the data is the risk of concept drift. To ensure the integrity of our models, it's important to track their performance over time. Changes in where bookies place over/under lines or how the game is played can potentially invalidate decisions our model is making. It is key to incorporate null hypothesis testing to track model performance over time. We recommend using a standard 95% threshold to identify decreases in performance as a trigger to update, retrain, or identify a new model.

At this point in time, sports betting is legal in 10 states. It is crucial that when implementing this model, we make sure it complies with each state's betting regulations and guidelines, as well as with their tax codes. With regard to ethical considerations, it is important to acknowledge that gambling is an addiction and that, by being part of the gambling ecosystem, we would be earning money from addicts. This is something that could speak against the mass deployment of our model.

References

MLB Stats, Scores, History, & Records. (n.d.). Retrieved October 2019, from https://www.baseball-reference.com/.

MLB 2019 Results & Historical Odds. (n.d.). Retrieved October 2019, from https://www.oddsportal.com/baseball/usa/mlb/results/.

scikit-learn. (n.d.). Retrieved December 2019, from https://scikit-learn.org/stable/index.html.

MLB SCORES AND ODDS ARCHIVES. (n.d.). Retrieved October 2019, from https://sportsbookreviewsonline.com/scoresoddsarchives/mlb/mlboddsarchives.htm.

Should I normalize/standardize/rescale? (n.d.). Retrieved December 2019, from http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-16.html.

Culver, J. (2019, September 4). Why 52.4% is the most important percentage in sports gambling. Retrieved December 2019, from https://medium.com/the-intelligent-sports-wagerer/why-52-4-is-the-most-important-percentage-in-sports-gambling-16ade8003c04

Appendix

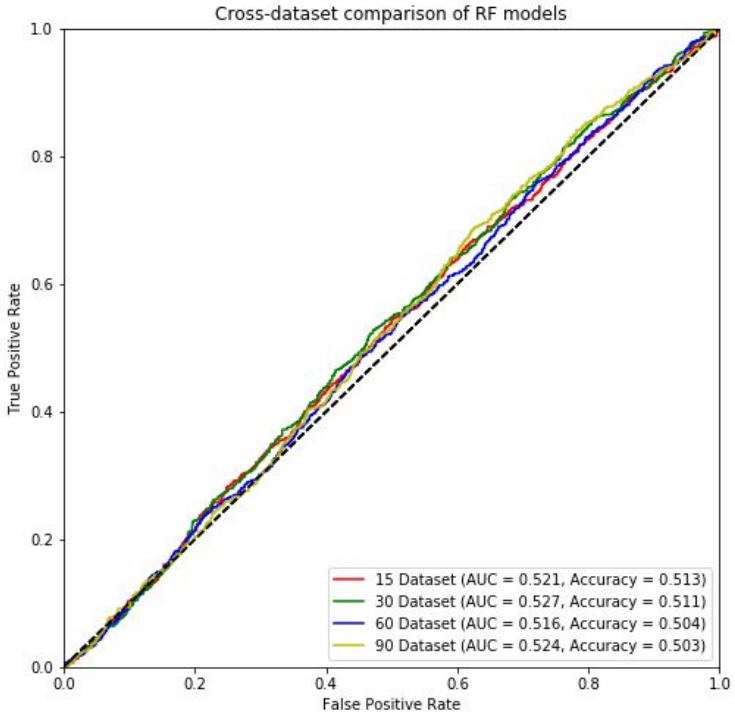
Individual team member contributions:

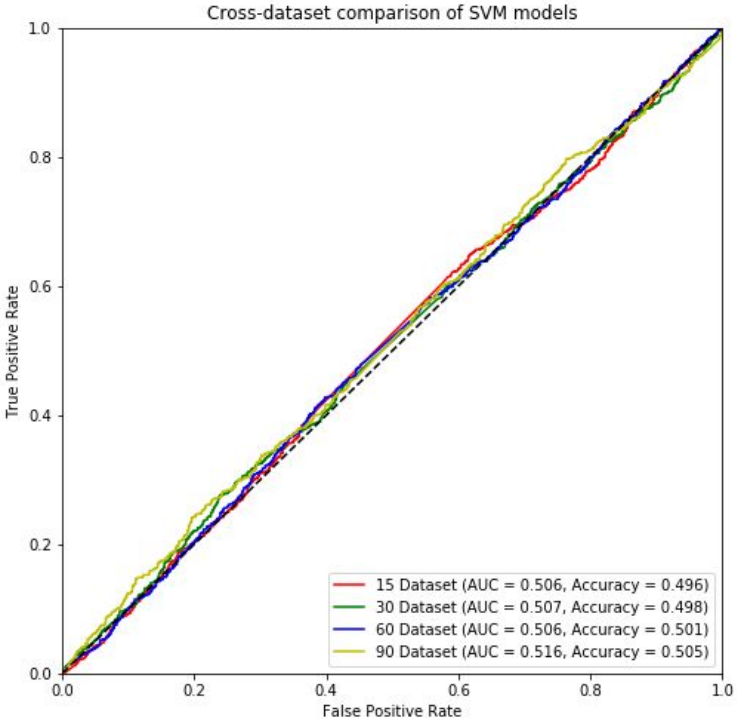
Guido Petri: BaseballReference scraping, Baseline model, GradientBoosting, LogisticRegression, Report sections on scraping, GB, LR and overarching consistency

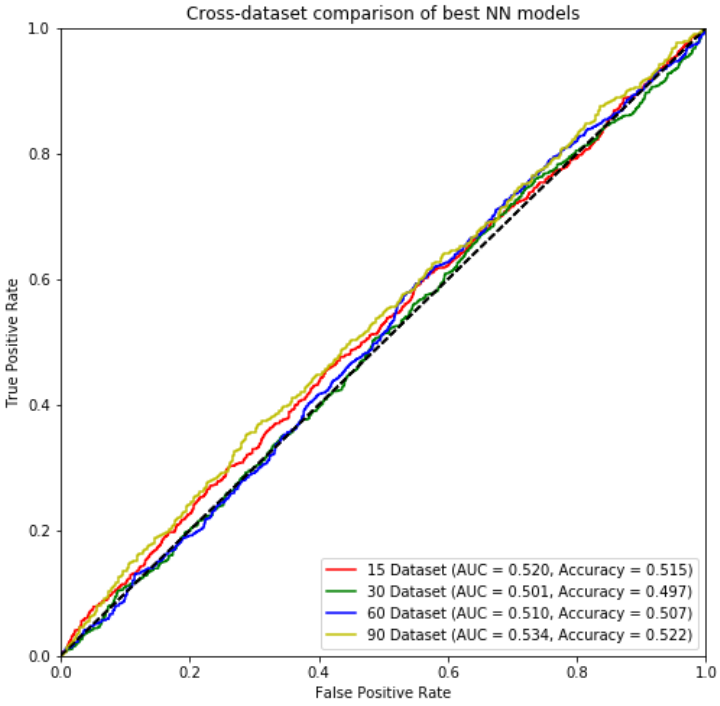
Nathan Griffin: Data cleaning, Baseline model, RandomForest, Report sections on data cleaning and RF, accuracy sensitivity analysis

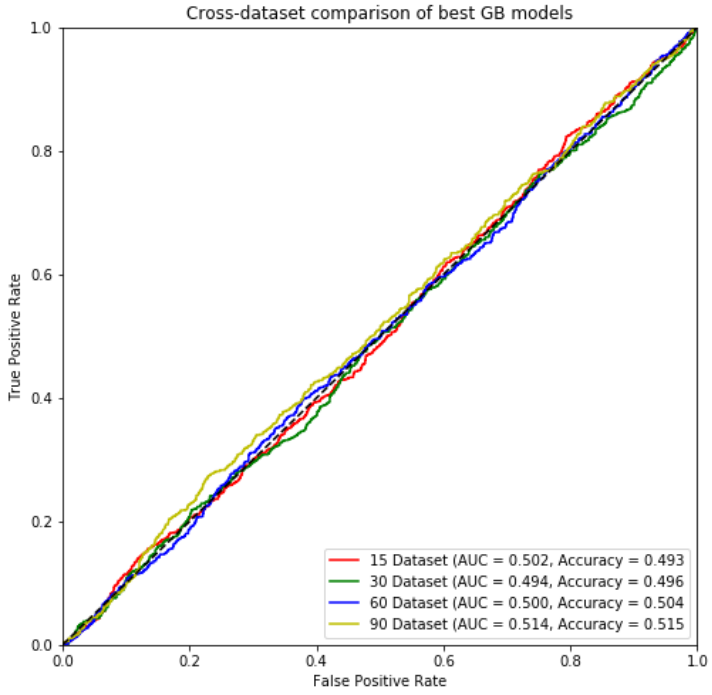
Jonas Peeters: Custom rolling averages, ELO-ratings, pitcher benchmarks, initial logistic regression modelling, SVM, Report sections on feature engineering and SVM, abstract

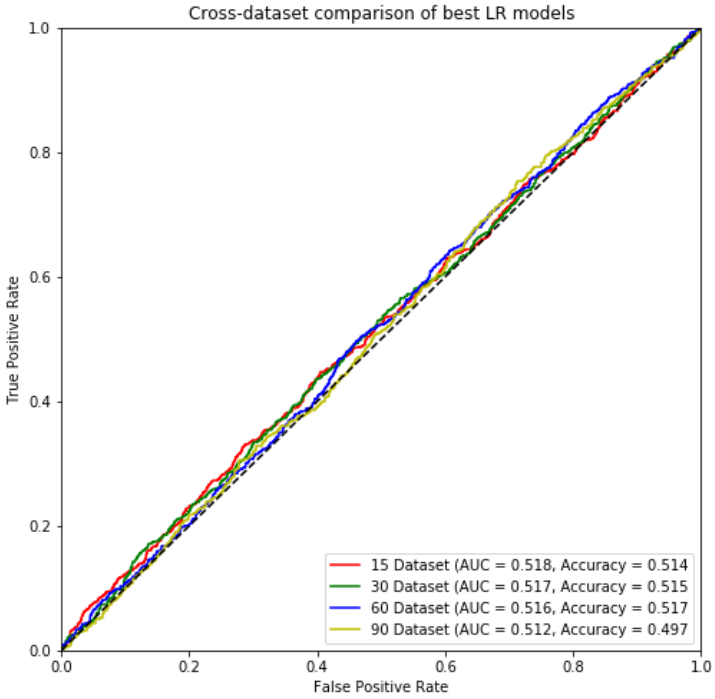
Stephen Roy: OddsPortal and SportsBookReviewsOnline scraping, dataset merging, Neural Networks, Report sections on scraping, NN, discussion, and overarching structure

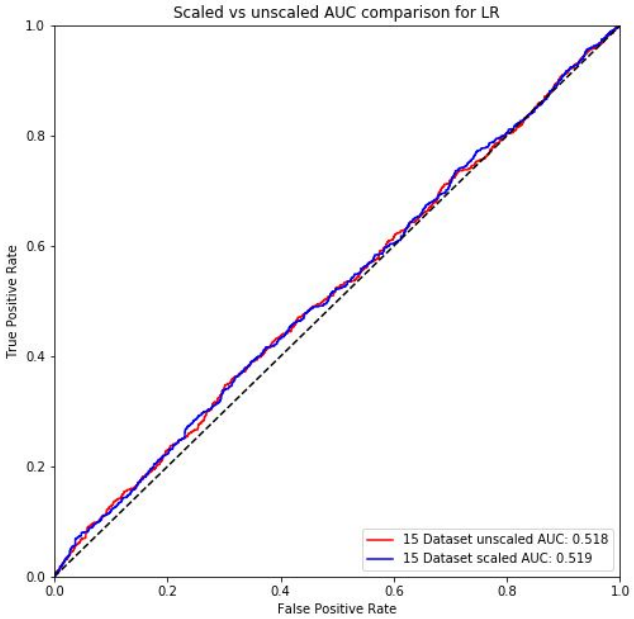

Cross-dataset comparison of RF models
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
15 Dataset (AUC = 0.521, Accuracy = 0.513)
30 Dataset (AUC = 0.527, Accuracy = 0.511)
60 Dataset (AUC = 0.516, Accuracy = 0.504)
90 Dataset (AUC = 0.524, Accuracy = 0.503)


Cross-dataset comparison of SVM models
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
15 Dataset (AUC = 0.506, Accuracy = 0.496)
30 Dataset (AUC = 0.507, Accuracy = 0.498)
60 Dataset (AUC = 0.506, Accuracy = 0.501)
90 Dataset (AUC = 0.516, Accuracy = 0.505)

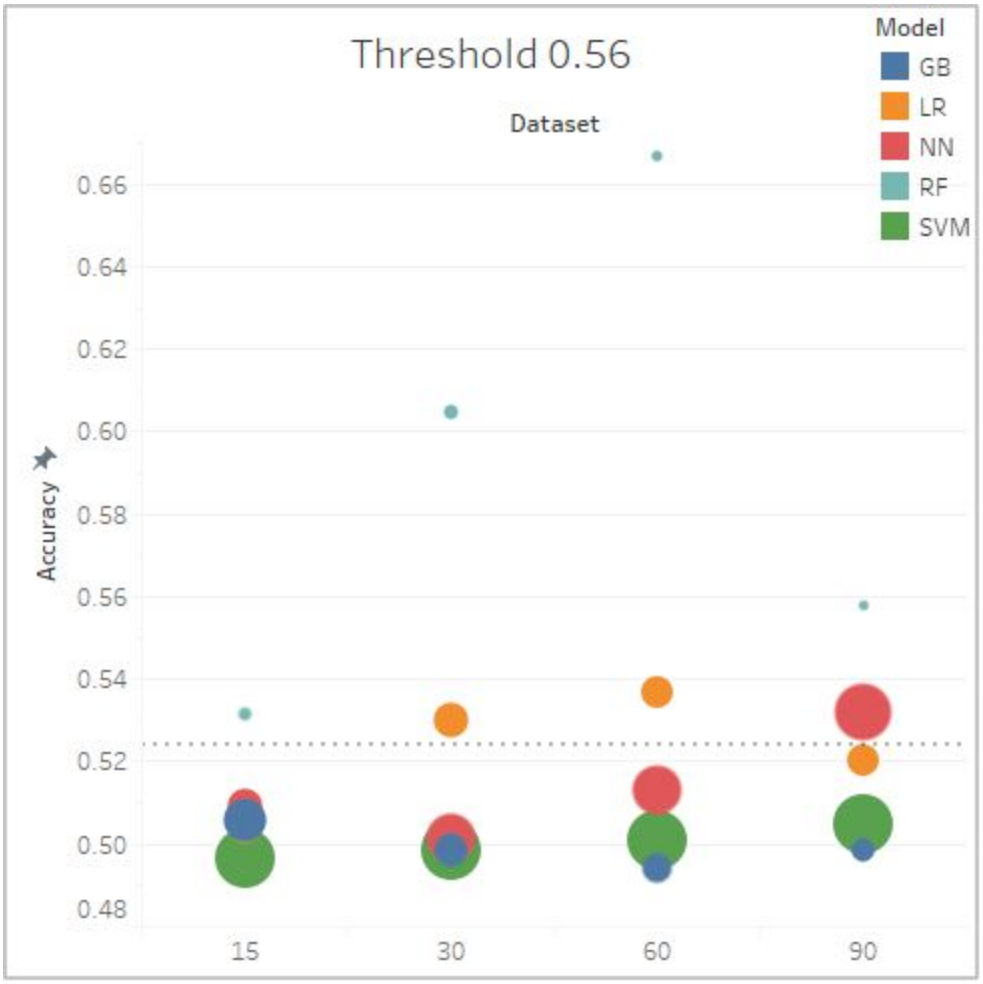

Cross-dataset comparison of best NN models
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
15 Dataset (AUC = 0.520, Accuracy = 0.515)
30 Dataset (AUC = 0.501, Accuracy = 0.497)
60 Dataset (AUC = 0.510, Accuracy = 0.507)
90 Dataset (AUC = 0.534, Accuracy = 0.522)

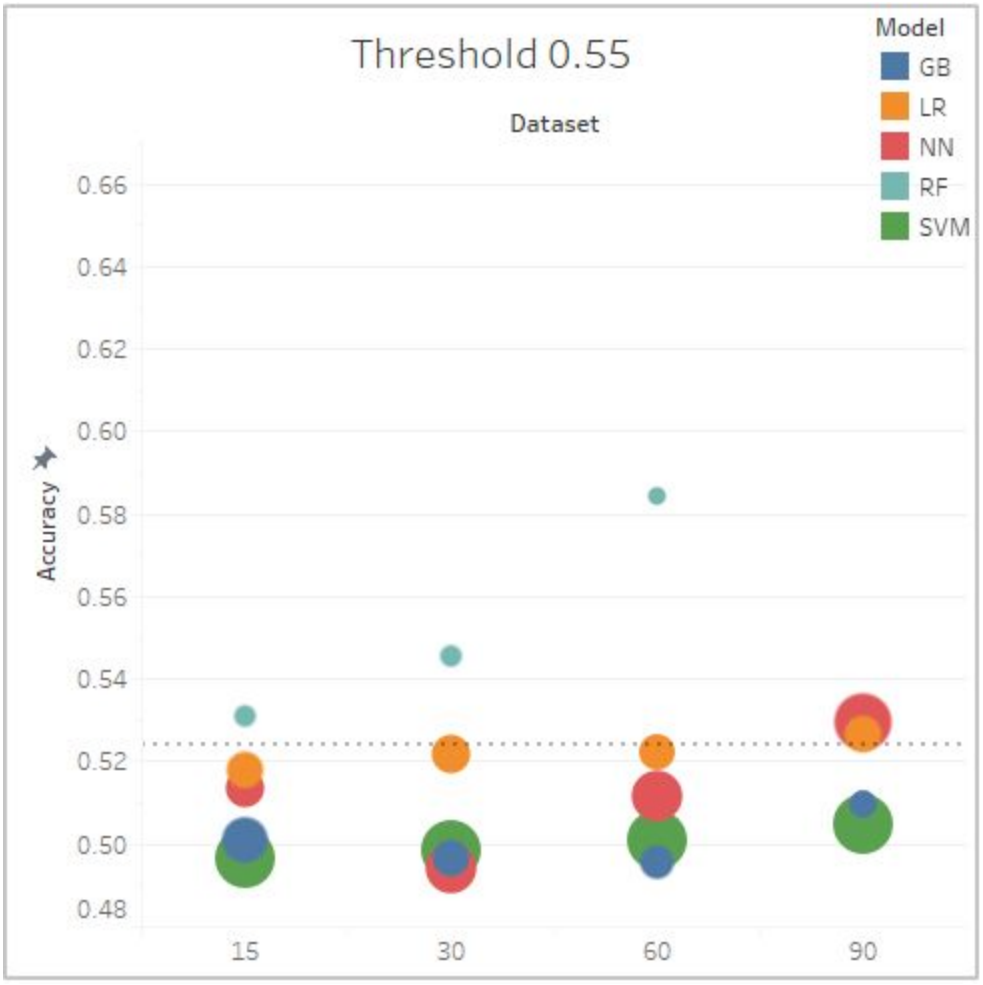

Cross-dataset comparison of best GB models
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
15 Dataset (AUC = 0.502, Accuracy = 0.493
30 Dataset (AUC = 0.494, Accuracy = 0.496
60 Dataset (AUC = 0.500, Accuracy = 0.504
90 Dataset (AUC = 0.514, Accuracy = 0.515

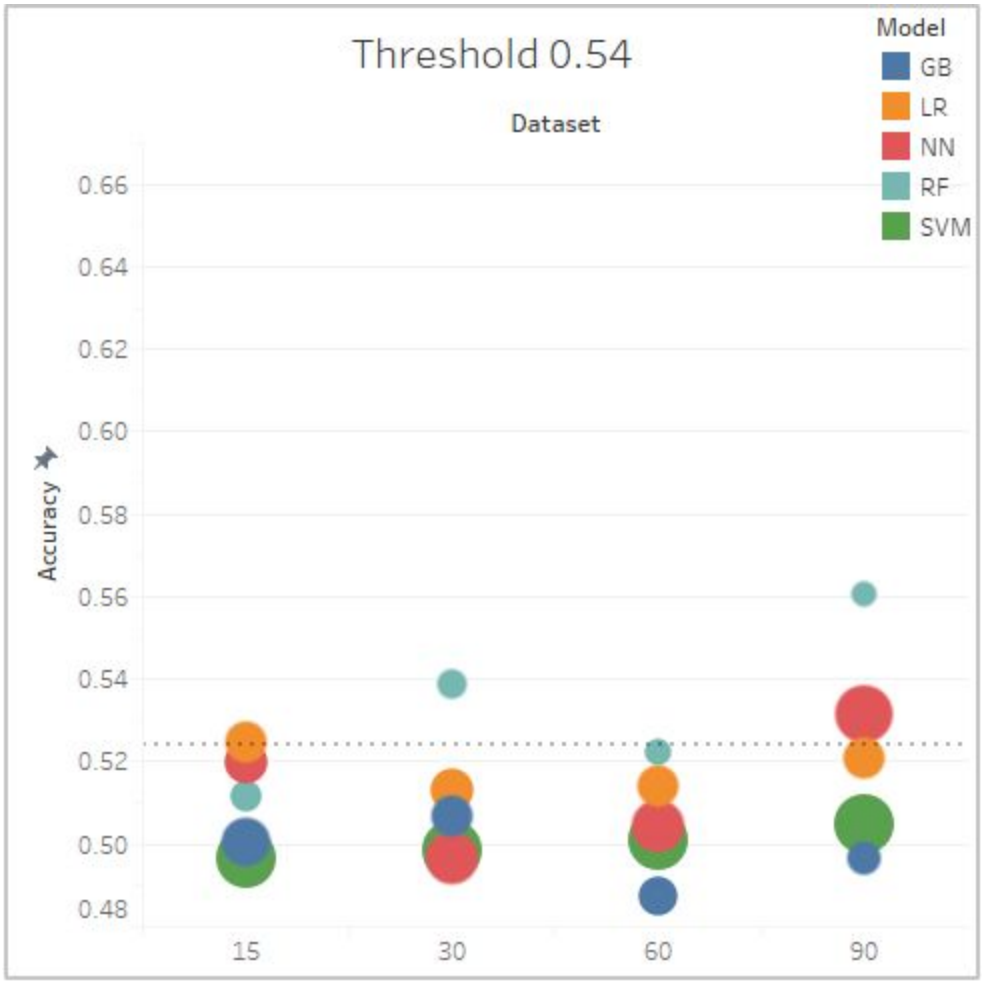

Cross-dataset comparison of best LR models
True Positive Rate
False Positive Rate
15 Dataset (AUC = 0.518, Accuracy = 0.514
30 Dataset (AUC = 0.517, Accuracy = 0.515
60 Dataset (AUC = 0.516, Accuracy = 0.517
90 Dataset (AUC = 0.512, Accuracy = 0.497

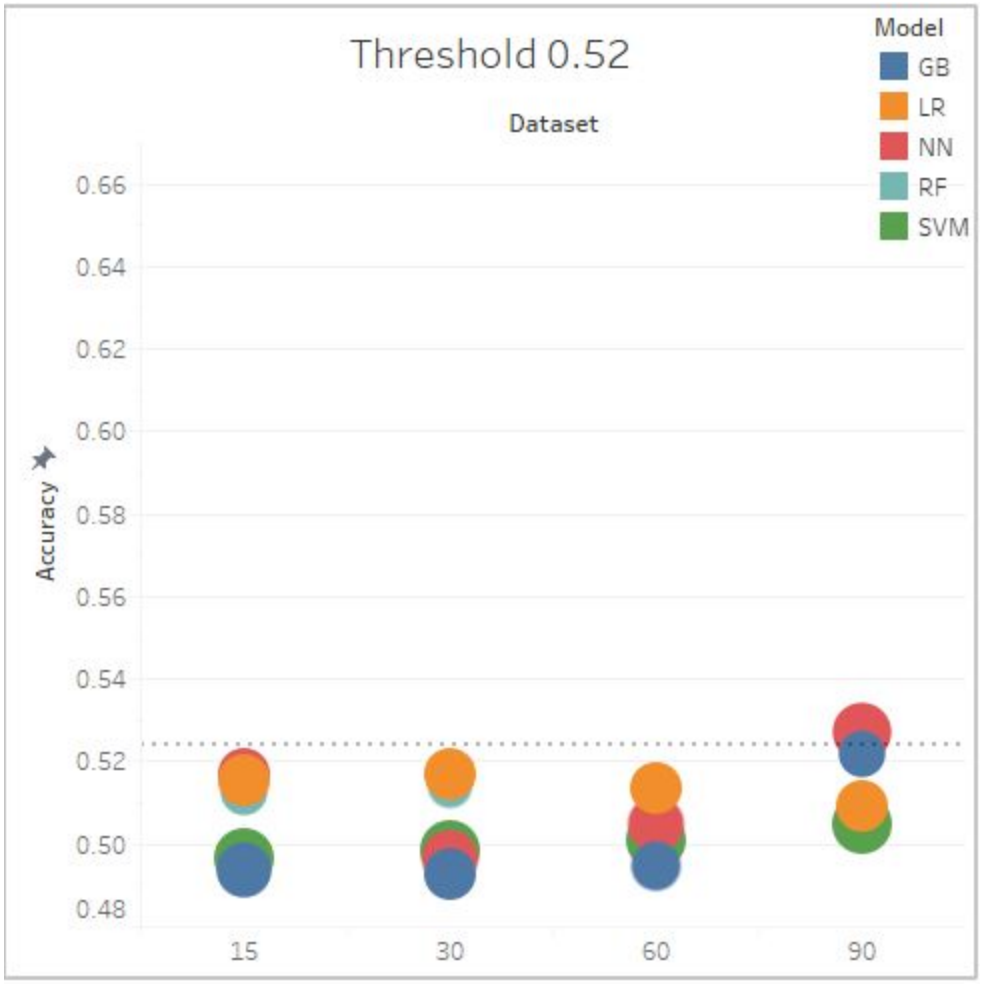

Scaled vs unscaled AUC comparison for LR
True Positive Rate
False Positive Rate
15 Dataset unscaled AUC: 0.518
15 Dataset scaled AUC: 0.519

The below graphs show a sensitivity analysis of prediction accuracy for each model and dataset combination. Here we only look at games where the predicted probability of class inclusion (over or under) is greater than a specific threshold. The dotted line is the break even accuracy. The size of each circle corresponds to the proportion of total games in the test set whose prediction met the threshold requirement. Since SVM had uniform predictions, its marks can be used as a reference for 100% of the test set.




Threshold 0.55
Dataset
Model
GB
LR
NN
RF
SVM
Accuracy
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0.52
0.50
0.48
15
30
60
90


Threshold 0.54
Model
GB
LR
NN
RF
SVM
Dataset
Accuracy
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0.52
0.50
0.48
15
30
60
90


Threshold 0.52
Dataset
Model
GB
LR
NN
RF
SVM
Accuracy
0.66
0.64
0.62
0.60
0.58
0.56
0.54
0.52
0.50
0.48
15
30
60
90