Predicting Strikeouts in Major League Baseball

Spring 2018
DS-GA 1003 / CSCI-GA 2567
Project Report

James Bannon, Mathew Thomas, Paul Fisher

1 Introduction

Each at bat in a professional baseball game consists of an adversarial competition between pitcher and batter. The batter's goal is to reach base (including by hitting home runs) while the pitchers is to get the batter "out." One primary means of achieving this goal is for a batter to strikeout (commonly abbreviated K).

Strikeouts are a particularly efficient way of achieving an out in that they limit the opportunities for runners on base to advance and avoid the uncertainty of what happens once the ball is put in play. They are a pure measure of skill for the pitcher as they do not require the involvement of any other members of the defense (besides the catcher). A pitcher regularly achieving strikeouts is a measure of high skill. By way of the example the average strikeout rate in 2017 was 21.6% or 8.25 per game. The major league leaders in strikeout percentage stuck out between 36.3% (starting pitchers) and 42.3% (min. 60 IP) of the batters they faced. [1]

The goal of this project was to use batter and pitcher data to predict the outcome of a plate appearance. Specifically, we focused on a specific outcome: the occurrence or non-occurrence of a strikeout. We can state our high level problem as follows:

The Strikeout Prediction Problem: For a given pitcher p_i and a batter b_i at the beginning of a specific plate appearance a_i, predict the probability that a_i ends with a strikeout.

In attacking this problem we followed two approaches: developing *an ensemble of probabilistic classifiers.* and *model-informed logistic regression.* Both approaches begin from the point of view of *probabilistic classification* where instead of predicting hard outcomes — in our case 1 and 0 in the case of strikeout or no strikeout, respectively.

The rest of the report is structured as follows. In Section 2 we describe our dataset in words as well as problems we encountered with the data set quality. A full description of the fields in the data sets we used is in Appendix A. Then in Section 3 we describe our two metrics for evaluation model performance: log loss and area under the ROC curve. Section 4 describes the two baseline models we used: a "dummy classifier" as well as a linear fixed effect model and reports the results of those models in terms of our evaluation metrics. With a baseline established we show in Section 5 the models we used to try and outperform the baseline. Sections 6 through 8 describe the Observations and Results of our various approaches, and Sections Section 9 concludes the report with a discussion of challenges we faced and a discussion of possible future directions to pursue.

[1] `https://www.fangraphs.com/leaders.aspx?pos=all&stats=pit&lg=all&qual=60&type=1&season=2017&month=0&season1=2017&ind=0&team=0&rost=0&age=0&filter=&players=0`

Season	**Number of At Bats**	**Number of Pitches**
2012	184,644	696,083
2013	185,251	701,316
2014	184,351	695,665
2015	184,114	693,576
2016	185,003	711,150
2017	185,694	715,537
Total	1,109,056	4,213,327

Table 1: Table of Number At Bats and pitches in each data set for the regular seasons

2 The Data

In constructing our models we used two datasets from the 2012 through 2017 seasons: the AtBat dataset and the PitchFX datasets. Both datasets are split into three sections, Regular Season, Post Season, and Spring Training. For the purposes of this project we focused exclusively on regular season data. This was for several reasons.

The first is that they were the most internally consistent. Players in spring training may not appear in the regular season or may not have a sufficient number of at bats to have stabilized rate stats ([1]). The post season suffers from a similar problem which is that not all teams appear in the playoffs and an insufficient number of at bats may for rates to stabilize. The regular season data also was the *largest* data set and so allowed us to analyze the the importance of features in the most robust setting possible. Below we describe the two data sets while a complete listing of the specific fields that each contains is in Appendix A and the sizes of the datasets are shown in Table 1.

2.1 The At Bat Data

The AtBat dataset consists of relatively coarse-grained information that summarize an entire plate appearance. It included information about the participants — the pitcher and batter — as well as data encoding the context — including the setting and game situation — and outcome of the plate appearance. From this data we were able to extract categorical data about the setting and game context, as well as compute numerical data relating the participants and outcomes of the atbats. Each record in this dataset correspond to one sample in our learning problem, with a single outcome that we tried to predict probabilistically. This formed the core of our analysis.

2.2 The Pitch F/X Data

The PitchFX dataset is a highly detailed dataset containing pitch-by-pitch data including situational information and the result of each pitch. The core of the PitchFX dataset is the detailed physical information measurements of the path of each pitch in 3 dimensions, including its speed, movement, and location as it crosses the plane of the plate. These measurements are made by high speed cameras installed in each major league ballbark, augmented by a human operator. The dataset also contains each pitch's "nasty factor," an MLB-developed derived measure of overall pitch quality, and a machine-classified pitch type.

While both datasets were largely complete, about 1-2% of atbats were present in the AtBat dataset, but not in PitchFX. Missing at bats come from games not played in usual MLB venues with PitchFX cameras as well as times at which the PitchFX system was down. We addressed this problem by filling all PitchFX-derived features with league-average values for those samples.

We turn now how we evaluated models trained on this data set.

3 Performance Evaluation

As mentioned in the introduction we focused on probabilistic classification. Specifically we predicted strikeouts by predicting a **probability** before each at bat, and used a log-loss function to evaluate our model's performance. In its most general (i.e: multi-class) form log loss for N observations in M classes $\{y_{ij}\}, i = 1, \ldots, N, j = 1, \ldots, M$ each given a probability $p_i(j)$ that observation y_{ij} belongs to class j is given by

$$LL = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_i(j)) \tag{1}$$

In the case of binary classification — our case — we have $M = 2$ and the log loss simplifies to

$$LL = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log p_i + (1 - y_i)\log(1 - p_i) \tag{2}$$

where p_i represents the probability that $y_i = 1$, that is, the probability that a strikeout occurs. It's evident from Equation (2) that a perfect model will have a log loss of 0 and that the size of the log loss increases as performance decreases.

Since we are training probabilistic classifiers we also evaluated on the area under the Receiver Operator Characteristic curve (ROC AUC). ROC is a way of measuring the true positive and false negative performance of a probabilistic classifier. The area under the curve is a simple aggregate measure of a classifier's performance. In a ROC plot the classifier's curve — representing the sensitivity and specificity (equivalently, false positive rate and false negative rates) should — should stay as close to the left and top of the graph. An idea curve will represent an inverted 'L'. For a given classifier C the area under the curve should be greater than 0.5 for the model to be more informative than just guessing the mean. An area under the curve of 1 is a perfect classifier. Complete technical descriptions of these can be can be found in [2].

With our evaluation criteria established we turn to our baseline models.

4 Baseline Models

We used two baseline models: a *dummy classifier* and a linear mixed-effect models. For both models we sorted the at bat data in each year by date and within each game by at bat and took the first 80% of the data as training data — which we will refer to as the “training set” as the remaining 20% as the test data — the “test set.”

The dummy classifier worked as follows. For the at bats in the training set it computed the proportion that ended in strikeouts. For eac at bat in the test set it predicted this proportion as the probability of a strikeout.

We used a general linear mixed-effect model as our baseline. As an additional baseline we explored the “majority classifier” that predicts the most frequent outcome — in our case “no strikeout” — in all cases. The results of this majority classifier are in **??**. The majority classifier results are instructive as they show that strikeouts are infrequent and they provide a benchmark

Year	Model	Log Loss	ROC AUC
2012	Dummy	0.50421	0.5
	Fixed Effects	0.4902	0.6157
2013	Dummy	0.5050	0.5
	Fixed Effects	0.4909	0.6181
2014	Dummy	0.5085	0.5
	Fixed Effects	0.4920	0.6249
2015	Dummy	0.5121	0.5
	Fixed Effects	0.4983	0.6140
2016	Dummy	.5201	0.5
	Fixed Effects	0.5051	0.6173
2017	Dummy	0.5251	0.5
	Fixed Effects	0.5082	0.6242

Table 2: Performance of Baseline Models on

if at any point we choose to try to binarize our classifier. However, we will focus on continuous predictions of strikeout probabilities with log loss as our main goal and evaluation criterion.

Linear mixed-effects models ([3]) are extensions of linear regression models for data that are collected and summarized in groups. These models describe the relationship between a response variable and independent variables, with coefficients that can vary with respect to one or more grouping variables. A mixed-effects model consists of two parts, fixed effects and random effects. Fixed-effects terms are usually the conventional "linear regression part" — i.e. real (vector) valued predictors — and the random effects are associated with individual experimental units drawn at random from subpopulations. The random effects have prior distributions whereas fixed effects do not. The overall goal of LMEs is to construct hierarchical models where within-group performance (e.g. deviation from the mean) is used as opposed to simple raw scores. They are an ideal baseline for this case

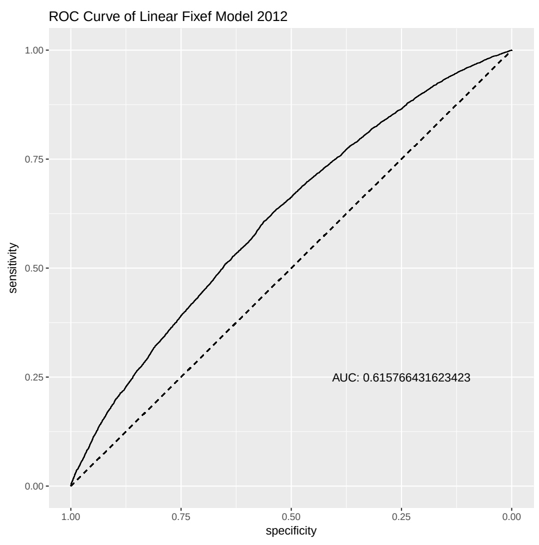
The particular model that we fit as a baseline utilized random effects for pitcher, batter, and stadium and had no fixed effects. In essence the goal was to create a model that roughly captured the key ingredients that may influence an at bat, namely: batter ability, pitcher ability, and situation.

In `R`-like notation the model we fit is as follows:

$$K \sim (1|\text{batter}) + (1|\text{pitcher}) + (1|\text{stadium}) \tag{3}$$

To test this model we sorted the regular season at-bats by date and then for each season took 80% of the data points and trained the model in eq. (3) and then evaluated it on the remaining data from that season as a test set. The results of this process are in **??**.

Thus the random effects model above only improves slightly on picking the maximum likelihood estimators p_i. However, compared to the majority classifier that assigns a very low probability to a strikeout (e.g: $p_i = 10^{-15} \forall i$) which incurs a training and test log loss of 7.402047 and 7.552219 respectively for 2017, these models are very good.



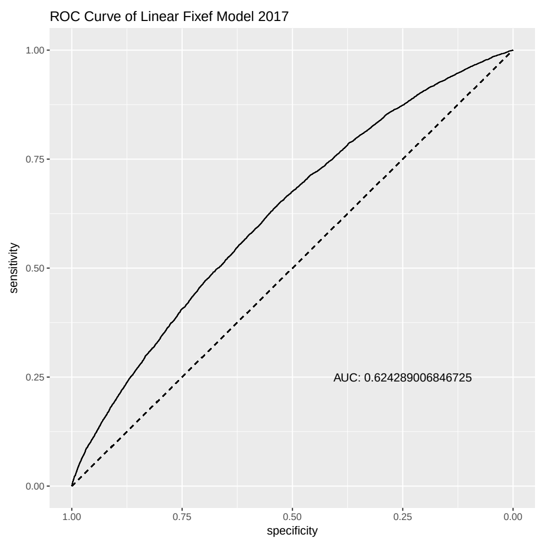


Figure 1: ROC For Linear Fixed Effects Model in 2012 and 2017

5 Models

5.1 Logistic Regression

Logistic regression [4] is a statistical method used to analyze datasets with two possible outcomes. The goal of logistic regression is to find the best fitting model that predicts the correct outcome when given a set of input features. The output of logistic regression is a set of coefficients of a formula that can be used to predict a logit transformation of the probability of presence of the characteristic of interest:

$$logit(p) = w_0 + w_1X_1 + ...w_nX_n$$

where
w => weight vector with components $(w_0, w_1, ...w_n)$
X => Input vector with features $(X_1, ...X_n)$

$$logit(p) = ln(\frac{p}{1-p})$$

5.2 Gradient Boosting

Gradient boosting [5] is an ensemble learning model generated using a collection of weake prediction models, typically decision trees. Gradient Boosing generates the final model in an iterative fashion by optimization of an arbitrary differentiable loss function.

The gradient boosting method determines an approximation $\hat{F}(x)$ by expressing it in the form of a weighted sum of base functions from a hypothesis space H. These base functions are called weak learners.

$$F(x) = \sum_{i=1}^{M} \gamma_i h_i(x)$$

In order to determine F(x), we start with a constant function and incrementally expand it in a greedy fashion by minimizing loss on each step.

$$F_0(x) = \operatorname*{argmin}_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma) F_m(x) = F_{m-1}(x) + \operatorname*{argmin}_{h_m} \sum_{i=1}^{N} L(y_i, F_{m-1}(x) + h_m(x_i))$$

5.3 AdaBoost

AdaBoost [6], short for Adaptive Boosting, is a machine learning algorithm developed by Yoav Freund and Robert Schapire. In AdaBoost, the output of multiple other learning algorithms called weak learners are combined using a weighted sum to produce a final output. AdaBoost is adaptive due to its tuning of its weak learners in a sequential fashion. By evaluating the performance of an individual weak learner on the given data, AdaBoost is able to tune subsequent learners to avoid the misclassifications previously encountered. The performance of individual weak learners need not be particularly good, as long as they all perform better than random guessing. The principle behind AdaBoost is that combining multiple weak learners can produce a strong learner if we tune accordingly. AdaBoost is commonly used with decision trees, with the results of previous decision trees being fed to the tree growing algorithm.

AdaBoost is essentially a training method for a boosted classifier. A boost classifier is a classifier of the form :

$$F_T(x) = \sum_{t=1}^{T} f_t(x)$$

where f_t is a weak learner, ie, a classifier. The weak learners produce a hypothesis $h(x_i)$ for each input point x_i. At each stage of the iteration through all the weak learners, a new weak learner is selected and assigned a coefficient α_t such that the training error up to that point is minimized. The training error is given by

$$E_t = \sum_{i}^{n} E[F_{t-1}(x_i) + \alpha_t h(x_i)]$$

where $h(x_i)$ is the hypothesis, and F_{t-1} is the boosted classifier generated before the current iteration.

At each iteration, a weight w_t is assigned to each sample in the training set equal to the current error $E[F_{t-1}(x_i)]$ on that sample.

5.4 Random Forests

Random forests [7] are an ensemble learning comprised of multiple decision trees. Each of the trained decision trees is used to determine the class of the given input, with the output of the random forest being the mode of the predicted classes (in the case of classification), or the mean predicted value (in the case of regression). Taking this average/mode of the predicted classes avoids the common pitfall of overfitting observed in decision trees.

To train random forests, we use bagging on our input data set, sampling with replacement in order to generate B training sets each with n elements each. We train a decision tree on each of these B training sets.

By averaging the predictions of each of the decision trees (or by finding the mode in the case of classifiers), we get our final prediction, ie, :

$$\hat{f} = \frac{1}{B}\sum_{b=1}^{B} f_b(x)$$

The use of bagging allows us to reduce the variance of the prediction function without increase in the bias. We accomplish this since bagging ensures that the various decision trees aren't highly correlated.

In general, we adjust the number of decision trees in a random forest based on the dataset. We find an optimum number of estimators using cross-validation. We can also determine the ideal number of trees using the out-of-bag error as a metric, where the out-of-bag error is the mean prediction error on an element using only estimators which do not contain the element in their training set.

5.5 Extra Trees

Extra Trees or extremely randomized trees is an ensemble method derived from the random forests method. The goal of the extra trees method is to further randomize the training of decision trees. Extra trees was proposed by P. Geurts et. al [8].

Extra trees does not use bootstrap copies of the learning sample or search for an optimal split point for each node. Instead it chooses split points at random for each node in order to generate sufficiently uncorrelated decision trees. This significantly reduces processing time resulting from choosing optimal split points, while still increasing the accuracy of the model.

Using randomized split points instead of bagging leads to an advantage in terms of bias, and often a reduction in variance as well. This method has yielded state-of-the-art results in several high-dimensional complex problems.

5.6 Multilayer Perceptron

A multilayer perceptron [9] is a type of artificial neural network consisting of a minimum of three node layers. Each of the non-input nodes utilizes a non-linear activation function. Commonly used activation functions include :

$$y(x) = tanh(x)y(x) = (1 + e^{-x})^{-1}$$

Each perceptron is trained by changing connection weights after each input is processed, with respect to the amount of error in the output compared to the expected result of the MLP. This is carried out using a technique called backpropagation.

Learning occurs in the perceptron by changing connection weights after each piece of data is processed, based on the amount of error in the output compared to the expected result. This is an example of supervised learning, and is carried out through backpropagation.

Using gradient descent, the change in each weight is

$$\Delta w_{ji}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial v_j(n)} y_i(n)$$

where $\varepsilon(n)$ is the error in the entire output, y_i is the output of the previous neuron $v_j(n)$ is the local input, and η is the learning rate.

The Feature Vector

Preprocessing

Each item in our dataset was a single plate appearance, a sigle matchup between a batter and pitcher which either did or did not result in a strikeout. The result of the matchup was extracted from the 'descr' field of the AtBat data and stored as 1 for a strikeout and 0 for any other outcome. Our job in feature engineering was to extract as much relevant information as possible about the pitcher, batter, and context of the plate appearance from the AtBat and PitchFX data and put it in a format that could be fed into our models.

Before we could do this, we needed to sort and clean up the data. This required choosing values for NaN fields and mapping stadiums with multiple names to the same value. In order to make our calculations of numerical features work, the data needed to be sorted in chronological order. While the AtBat data contained a date, the atbats within each date were not sorted by time. We instead sorted by stadium to separate the games after sorting by inning, side, and total of outs, baserunners, and score as a proxy for time. The PitchFX data did have a field for time of pitch, so we were able to use that to sort the data.

In order to combine pitch-by-pitch data from PitchFX with our AtBat data, we needed to group the pitches by plate appearence and create a one-to-one correspondance between a record in the AtBat data and a record in the PitchFX data. To do this we created a unique atbat ID using data found in both sets, including date, the identity of the pitcher and batter, the inning, and batting team's score. Then, once the rolling averages described below had been computed on a pitch-by-pitch basis, we dropped all but the first pitch of each plate appearance (the only one containing only data compiled prior to that plate appearance,) and merged the two sets together on that unique ID.

Feature Engineering

As shown above, the AtBats data came with a good number of features that were not real-valued. For some of these features, we used one-hot encoding to convert them to larger sets of binary features. These included the identity of each ballpark, the number of outs, the positions of runners on base, the inning and side of inning. While the inning and number of outs are integer values, including them as numeric feautures could only capture a possible monotonic correlation between the inning/number of outs and the probability of a strikeout. Encoding these as categorical features, in principle, allow us to learn any differences in strikeout rate in different innings or out situation.

We used Player data to construct categorical features based on the handedess of the batter and pitcher, with the knowledge that same-handed matchups tend to favor the pitcher and opposite handed matchups tend to favor the batter. We created a binary feature for each possible handedness combination, including LR, LL, LS, RR, etc. We dealt with missing data by assuming that players for which the batting handedness was missing would bat with the same side that they threw with.

Our assumption was that the most important features would be rate features computed for the pitcher and batter. Specifically, we needed a measure of how often the pitcher struck out batters and how often the batter himself struck out. We were able to calculate this by taking a cumulative sum of the strikeout column on a subset of the AtBat data restricted to the specific plater. This computed the number of strikeouts each player had accrued that season prior to that plate appearance, and then we were able to compute eac player's season to date strikeout rate. It was important to compute these rates using only information about the pitcher and bater compiled *before* the plate appearence in question. To do otherwise would be to train the data we are trying to predict. We were also careful to only split chronologically when doing cross validation and testing.

In order to reduce the effect of tiny sample sizes, we also regressed the strikeout rates to the mean by adding 50 imaginary at bats at league average strikeout rate to each computation. This made sustained records above and below league average to be weighted more heavily than early season noise.

We took recent performance into account by using a sliding window to compute strikeout rates in the last n plate appearances. We used three window sizes: for pitchers, 150, 70 and 25 batters faced, for batters, 150, 60 and 25 plate appearances. The size 150 window represents the last month or two to pick up longer term in-season trends, while the window of size 25 is meant to pick up short term "hotness." The size 70 and 60 windows were based on empirical results on how quickly a pitcher or batter's strikeout rate stabilizes.[2] In order to avoid repeating the same information in these and the overall season-to-date numbers, these window rates are represented as an offset from that player's own (regressed) in-season numbers.

Using the PitchFX dataset we were able to compute several more numeric features for the pitchers and batters. Based on the pitch location and outcome data, we added a record of called strikes, swinging strikes, and swings on pitches outside of the zone. Rates of these should correlate with strikeouts for both pitchers and batters. We calculated cumulative and rolling averages of each of the these using the same technique as for the strikeout rates. We also included the pitchFX "nasty factor."

We also computed features related to the pitcher's fastball. We focused on fastballs because pitchfx fields like speed, break, etc. would have different meanings for different pitch types and we would lose information by averaging over those different pitch types. Since practically every pitcher throws a fastball of some sort, we could restrict our set to fastballs and still have data for almost every pitcher. For the purposes of this data, we counted pitches classified as FF (four-seam fastball), FT (two-seam fastball), or SI (sinker) as fastballs.

We extracted average starting and ending velocity, vertical and horizontal break, and spin rate of a pitcher's fastballs, both over the course of the season and over the last 20 fastballs. Finally, based on empirical findings that a spin rate both above and below average can be helpful to a pitcher, we added a feature which gave the absolute difference between a pitcher's fastball spin rate and league average.[3]

Features

A comprehensive description of the features used in the final models is given in Appendix B.

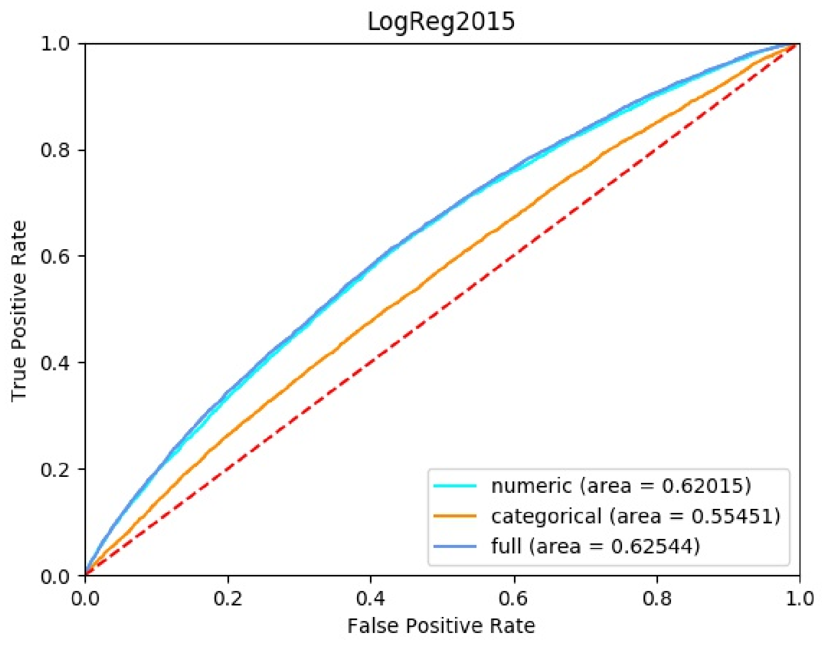
[2] https://www.fangraphs.com/library/principles/sample-size/

[3] https://www.mlb.com/news/what-statcast-spin-rate-means-for-fastballs/c-212735620

Observations and Results

6 Fitting the Models

6.1 Logistic Regression



The Logistic Regression module of the Scikit-Learn library in python is parametrized by multiple options. We applied a grid search in order to determine an optimal parameter set. The grid search was used to choose the best values for :

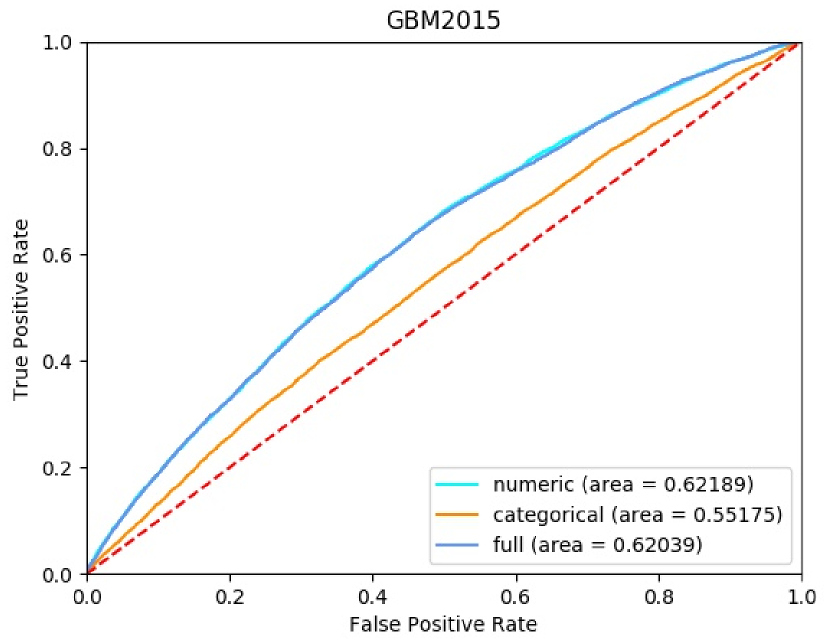
- *penalty* : 'l1' or 'l2' - Used to specify the norm used in the penalization.
- C : Inverse of regularization strength
- $fit\_intercept$: Specifies if a constant/bias/intercept should be added to the decision function.

Year	Features	Log Loss	ROC AUC
2012	Numeric	0.69139	**0.62155**
	Categorical	0.50137	0.55385
	Full	**0.48746**	**0.62700**
2013	Numeric	0.68172	**0.62311**
	Categorical	0.50300	0.54742
	Full	**0.48876**	**0.62837**
2014	Numeric	0.68021	**0.62665**
	Categorical	0.50587	0.55144
	Full	**0.49029**	**0.63313**

Year	Features	Log Loss	ROC AUC
2015	Numeric	0.68943	**0.62015**
	Categorical	0.50914	0.55451
	Full	**0.49575**	**0.62544**
2016	Numeric	0.50423	**0.62090**
	Categorical	0.51789	0.54803
	Full	**0.50317**	**0.62513**
2017	Numeric	**0.50817**	**0.62505**
	Categorical	0.52308	0.54570
	Full	**0.50720**	**0.62799**

Table 3: Logistic Regression results

6.2 Gradient Boosting



The Gradient Boosting module of the Scikit-Learn library in python is parametrized by multiple options. We applied a grid search in order to determine an optimal parameter set. The grid search was used to choose the best values for :

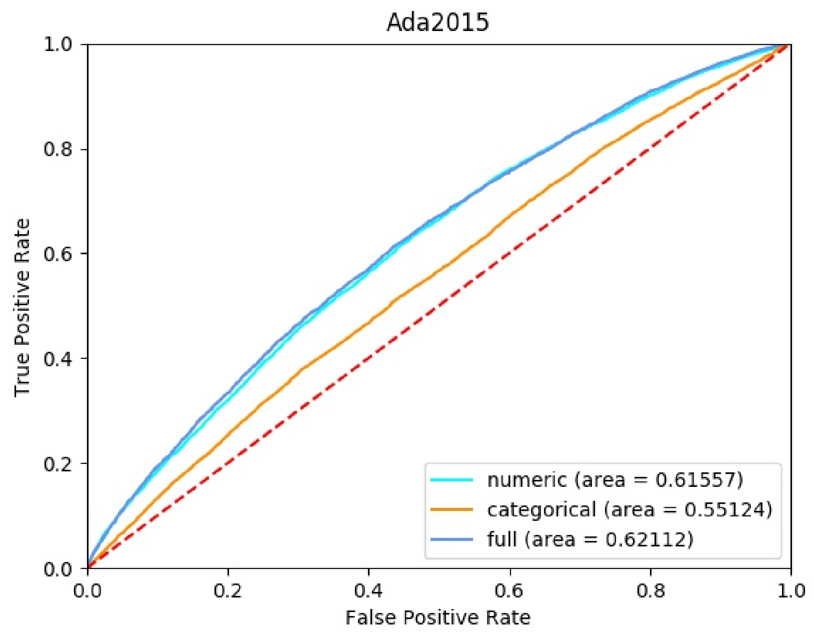
- *loss* : loss function to be optimized
- *n_estimators* : The number of boosting stages to perform
- *max_depth* : maximum depth of the individual regression estimators

Year	Features	Log Loss	ROC AUC
2012	Numeric	**0.48882**	**0.62102**
	Categorical	0.50144	0.55312
	Full	**0.48788**	**0.62541**
2013	Numeric	**0.49058**	**0.62202**
	Categorical	0.50267	0.55132
	Full	**0.48912**	**0.62728**
2014	Numeric	0.49224	**0.62731**
	Categorical	0.50599	0.55228
	Full	**0.49121**	**0.63165**

Year	Features	Log Loss	ROC AUC
2015	Numeric	**0.49679**	**0.62189**
	Categorical	0.50962	0.55175
	Full	**0.49732**	**0.62039**
2016	Numeric	**0.50380**	**0.62291**
	Categorical	0.51790	0.54860
	Full	**0.50335**	**0.62483**
2017	Numeric	0.50946	0.62098
	Categorical	0.52299	0.54632
	Full	0.50899	**0.62424**

Table 4: Gradient Boosting results

6.3 AdaBoost



The AdaBoost module of the Scikit-Learn library in python is parametrized by multiple options. We applied a grid search in order to determine an optimal parameter set. The grid search was used to choose the best values for :

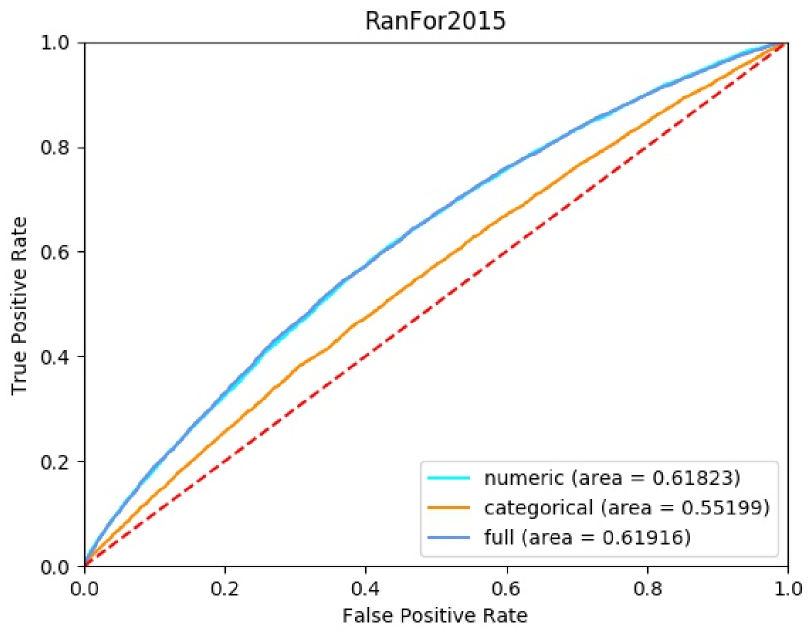
- *base_estimator* : base estimator from which the boosted ensemble is built
- *n_estimators* : maximum number of estimators at which boosting is terminated
- *DCTmax_leaf_nodes* : Grow a decision tree with *max_leaf_nodes* leaves in best-first fashion

Year	Features	Log Loss	ROC AUC
2012	Numeric	0.64895	0.61545
	Categorical	0.65229	0.55340
	Full	0.64768	**0.62034**
2013	Numeric	0.64786	0.61668
	Categorical	0.65249	0.54776
	Full	0.64875	**0.61994**
2014	Numeric	0.64868	0.61774
	Categorical	0.65414	0.54846
	Full	0.64808	0.62478

Year	Features	Log Loss	ROC AUC
2015	Numeric	0.65038	**0.61557**
	Categorical	0.65437	0.55124
	Full	0.65003	**0.62112**
2016	Numeric	0.65083	0.61402
	Categorical	0.65643	0.55056
	Full	0.65165	**0.61874**
2017	Numeric	0.65302	0.62274
	Categorical	0.65799	0.54525
	Full	0.65435	0.62170

Table 5: AdaBoost results

6.4 Random Forests



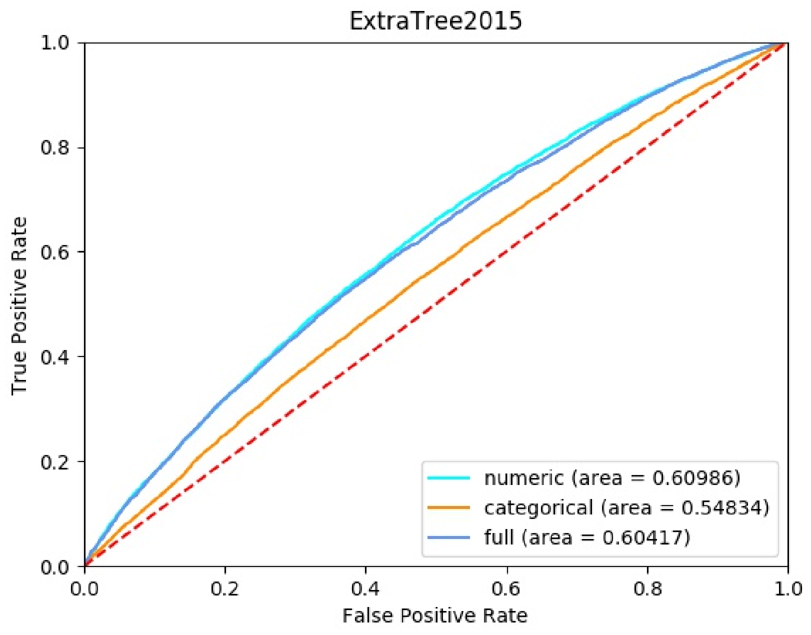
The Random Forest module of the Scikit-Learn library in python is parametrized by multiple options. We applied a grid search in order to determine an optimal parameter set. The grid search was used to choose the best values for :

- *max_depth* : the maximum depth of a tree
- *n_estimators* : the number of trees in the forest

Year	Features	Log Loss	ROC AUC	Year	Features	Log Loss	ROC AUC
2012	Numeric	**0.48955**	**0.61913**	2015	Numeric	**0.49772**	**0.61823**
	Categorical	0.50170	0.55379		Categorical	0.50964	0.55199
	Full	**0.49017**	**0.61973**		Full	0.49854	**0.61916**
2013	Numeric	**0.49088**	**0.61987**	2016	Numeric	0.50513	0.61983
	Categorical	0.50257	0.55387		Categorical	0.51805	0.54730
	Full	0.49180	**0.61824**		Full	0.50620	0.61921
2014	Numeric	0.49323	0.62374	2017	Numeric	0.50865	**0.62482**
	Categorical	0.50601	0.55199		Categorical	0.52305	0.54796
	Full	**0.49400**	0.61916		Full	0.50961	0.62391

Table 6: Random Forest results

6.5 Extra Trees



The Extra Trees module of the Scikit-Learn library in python is parametrized by multiple options. We applied a grid search in order to determine an optimal parameter set. The grid search was used to choose the best values for :

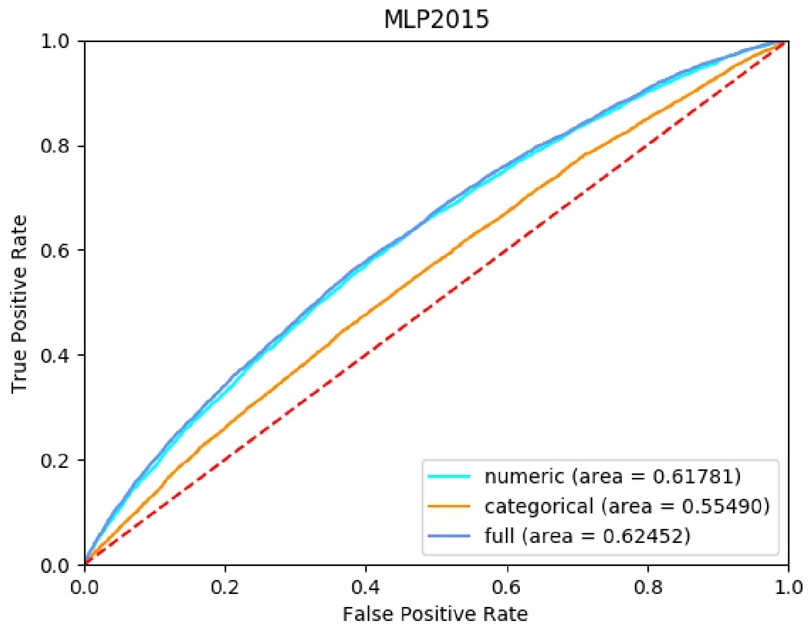
- *max_depth* : the maximum depth of a tree
- *n_estimators* : the number of trees in the forest
- *criterion* : The function to measure the quality of a split

Year	Features	Log Loss	ROC AUC
2012	Numeric	0.49247	0.60953
	Categorical	0.50170	0.55204
	Full	0.49410	0.60324
2013	Numeric	0.49365	0.60847
	Categorical	0.50308	0.54762
	Full	0.49477	0.60386
2014	Numeric	0.49540	0.61557
	Categorical	0.50615	0.55217
	Full	0.49934	0.61216

Year	Features	Log Loss	ROC AUC
2015	Numeric	0.50007	0.60986
	Categorical	0.50991	0.54834
	Full	0.50335	0.60417
2016	Numeric	0.50748	0.61163
	Categorical	0.51799	0.54523
	Full	0.51201	0.60161
2017	Numeric	0.51187	0.61271
	Categorical	0.52333	0.54336
	Full	0.51380	0.60626

Table 7: Extra Trees results

6.6 Multi-Layer Perceptron



The Multilayer Perceptron module of the Scikit-Learn library in python is parametrized by multiple options. We applied a grid search in order to determine an optimal parameter set. The grid search was used to choose the best values for :

- *hidden_layer_sizes* : the ith element represents the number of neurons in the ith hidden layer.
- *activation* : Activation function for the hidden layer
- *alpha* : L2 penalty (regularization term) parameter

Year	Features	Log Loss	ROC AUC
2012	Numeric	**0.48895**	**0.62009**
	Categorical	0.50136	0.55308
	Full	**0.48795**	**0.62619**
2013	Numeric	0.49116	**0.62248**
	Categorical	0.50239	0.55062
	Full	**0.48956**	**0.62443**
2014	Numeric	0.49225	0.62476
	Categorical	0.50549	0.55477
	Full	0.49109	**0.63028**

Year	Features	Log Loss	ROC AUC
2015	Numeric	**0.49750**	**0.61781**
	Categorical	0.50896	0.55490
	Full	**0.49583**	**0.62452**
2016	Numeric	**0.50445**	0.62058
	Categorical	0.51791	0.55131
	Full	**0.50378**	0.62290
2017	Numeric	0.50838	**0.62459**
	Categorical	0.52218	0.55244
	Full	0.50837	0.62301

Table 8: Multi-Layer Perceptron results

7 Bagged & Convex-Fitted Ensembles

Having fit all the models described above on each year and with their associated parameter grids we collected them together into a weighted ensemble.

An ensemble consists of a collection of models $f_i \dots, f_m$ that each take some input data x and output a value in the same space. In our case the f_i's are the models described above and the outputs are their probabilities. In general formulation is:

$$f_e(x) = \sum_{i=1}^{m} w_i f_i(x)$$

In the case of probabilities we impose the restriction that w_i be nonnegative for all i and that $\sum_i w_i = 1$.

The lingering question is how does one decide on what weights to use for an ensemble. One such approach is *bagging* where the outputs are averaged:

$$f_e(x) = \frac{1}{m} \sum_{i=1}^{m} f_i(x) \tag{4}$$

We used this approach and in addition tried a second method.

For the second method we fit the weights w_i for each model by solving a convex program. To be precise, for each year $i \in \{2012, 2013, 2014, 2015, 2016, 2017\}$ we constructed three matrices:

$$\begin{aligned} X_i^{(n)} &= \{ \text{ Set of numeric models evaluated on the data for year } i\} \\ X_i^{(c)} &= \{ \text{ Set of categorical models evaluated on the data for year } i\} \\ X_i^{(f)} &= \{ \text{ Set of full models evaluated on the data for year } i\} \end{aligned}$$

If there are b_i at-bats in each year and m models (in our case $m = 5$) then each matrix has dimension $b_i \times m$. That is, each matrix was a matrix of probabilities with number of columns equal to the number of models in the ensemble and the number of rows equal to the number of at bats. We also developed an *aggregate* matrix

$$X_i^{(a)} = \begin{bmatrix} X_i^{(n)} & X_i^{(c)} & X_i^{(f)} \end{bmatrix}$$

by stacking the outputs horizontally. This approach allowed us to see which weights among all model types and all data types to give the most weight to (in the convex setting) and allowed us to make an all-encompassing average (in the case of bagging).

Fitting the bagging model was trivial; we simply averaged the predicted probabilities and compute the log loss and ROC AUC. The results are in Table 10 and Figure 3.

The task of fitting our ensemble weights was amenable to a writing as a convex program, as shown in (5).

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & LL(X_i^{(t)} w^{(i)}) \\ \text{subject to} \quad & \sum_{j=1}^{m} w_j^{(i)} = 1 \\ & w_j^{(i)} \geq 0 \; j = 1, \dots, m \end{aligned} \tag{5}$$

Year	Ensemble	Log Loss	ROC AUC
2012	Numeric	0.64895	0.61545
	Categorical	0.65229	0.55340
	Full	0.64768	**0.62033**
	Aggregate	0.65229	0.55328
2013	Numeric	0.64901	0.61636
	Categorical	0.65249	0.54775
	Full	0.64875	**0.61992**
	Aggregate	0.65246	0.54968
2014	Numeric	Solver Failed	Solver Failed
	Categorical	Solver Failed	Solver Failed
	Full	Solver Failed	Solver Failed
	Aggregate	Solver Failed	Solver Failed
2015	Numeric	Solver Failed	Solver Failed
	Categorical	Solver Failed	Solver Failed
	Full	Solver Failed	Solver Failed
	Aggregate	Solver Failed	Solver Failed
2016	Numeric	Solver Failed	Solver Failed
	Categorical	Solver Failed	Solver Failed
	Full	Solver Failed	Solver Failed
	Aggregate	Solver Failed	Solver Failed
2017	Numeric	0.6546	0.61099
	Categorical	0.65794	0.54612
	Full	0.65382	0.61427
	Aggregate	0.6575	0.57018

Table 9: Convex Program Ensemble Results

Year	Ensemble	Log Loss	ROC AUC
2012	Numeric	0.5357	**0.62139**
	Categorical	0.54218	0.55451
	Full	0.53575	**0.62355**
	Aggregate	0.53734	**0.62738**
2013	Numeric	0.5356	**0.62098**
	Categorical	0.54297	0.55285
	Full	0.53625	**0.62171**
	Aggregate	0.53781	**0.62632**
2014	Numeric	0.53868	0.61380
	Categorical	0.54662	0.55739
	Full	0.53887	**0.62227**
	Aggregate	0.54080	**0.62377**
2015	Numeric	0.54253	0.60617)
	Categorical	0.54844	0.55576
	Full	0.542743	0.61249
	Aggregate	0.54393	0.61453
2016	Numeric	0.54786	0.60553
	Categorical	0.55509	0.55168
	Full	0.54848	0.61230
	Aggregate	0.549867	0.61417
2017	Numeric	0.55184	0.61332
	Categorical	0.55951	0.55094
	Full	0.55201	0.61813
	Aggregate	0.55383	0.61969

Table 10: Bagged Ensemble Results

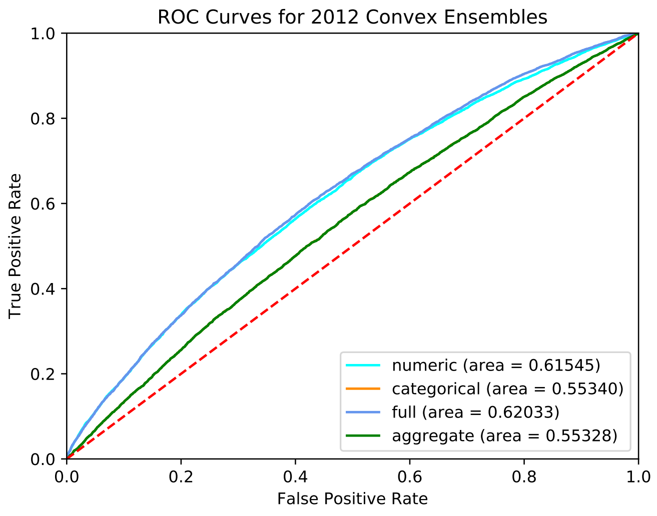
Where for reference:

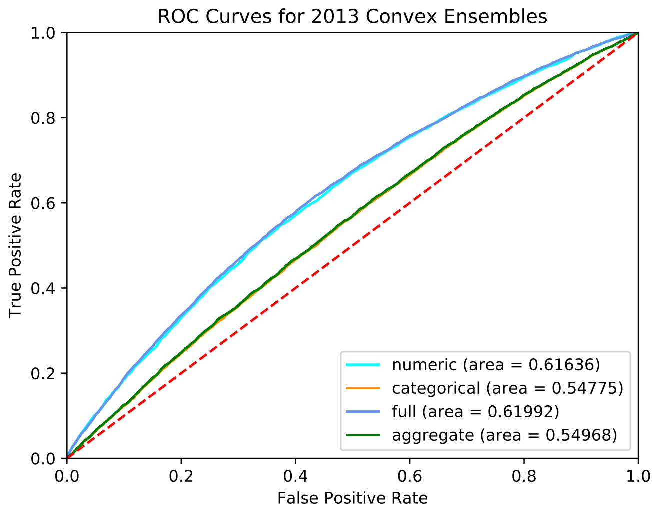
$$LL(X_i^{(t)}w^{(i)}) = -\frac{1}{b_i}\sum_{j=1}^{b_i} y_j \log\left\{(X_i^{(t)}w^{(i)})_j\right\} + (1-y_j)\log\left\{1-(X_i^{(t)}w^{(i)})_j\right\}$$

Using the library `CvxPy` we solved this program for each year and for each ensemble type (including the aggregates). We present the results of this approach in Table 9 and in Figure 2

In general the convex weighting seemed to hurt performance as opposed to pure averaging and in some cases, despite trying all available solvers, the program failed. The failure can be attributed to data set size and the poor performance can be attributed to the fact that there may be very close global minimizers. It is interesting to note that the convex program, in the years that it worked, did not converge to the bagging results as the bagging performed better. It is possible this is due to the algorithm terminating before convergence.

In some cases, however, bagging did seem to improve performance in ROC AUC, but not log loss.





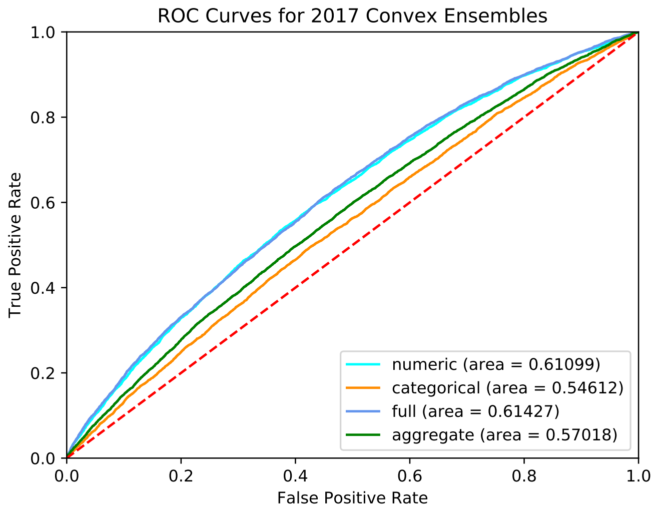
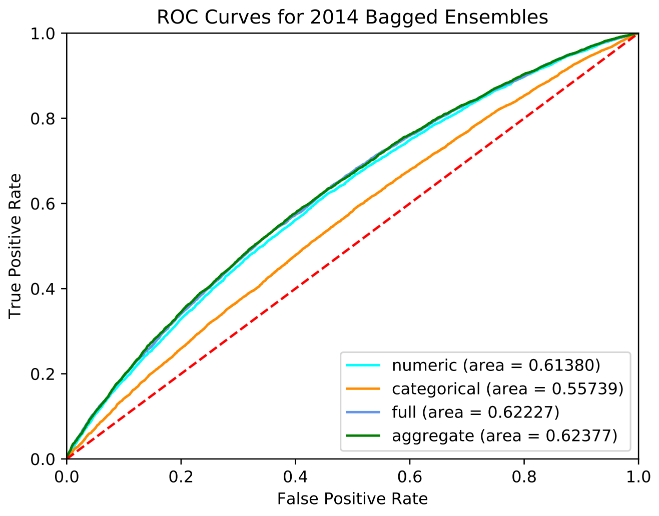
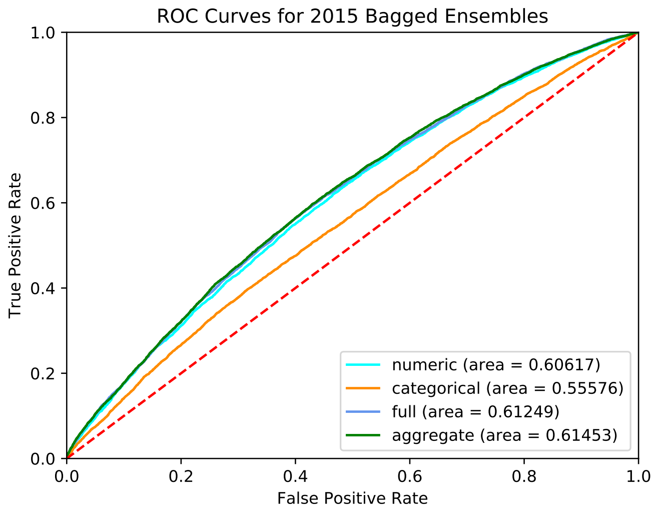


Figure 2: ROC Curves for Convex-Fit Ensembles Where Solver Didn’t Fail





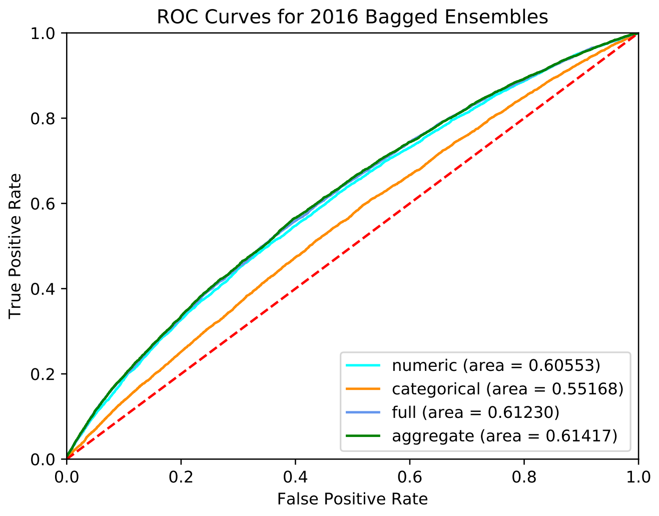


Figure 3: ROC Curves for Bagged Ensemble in Years where Convex Solver Failed.

8 Model-Informed Logistic Regression

Another approach which we attempted was the use of the predictions generated by the other models as an input for a logistic regression model. The idea proved difficult to implement due to processing requirements, with our own systems being unable to handle the computation, and NYU HPC systems returning a system time out. We were able to collect results from two seasons worth of Model-Informed Logistic Regression predictions. The results and diagrams are as shown below :

Year	Features	Log Loss	ROC AUC
2012	Numeric	0.69139	0.62155
	Categorical	0.50137	0.55386
	Full	0.48746	0.62700
2013	Numeric	0.68172	0.62311
	Categorical	0.50300	0.54744
	Full	0.48876	0.62837

Table 11: Model Informed Logistic Regression Results

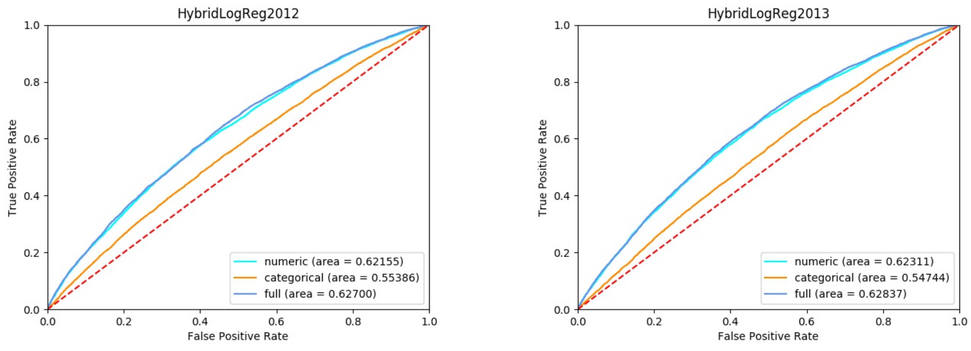


Figure 4: Model Informed Logistic Regression Results

As we can see, the results were nearly identical to standard logistic regression. Given the exorbitant computational requirements of the model and the difficulty in obtaining results, we decided that the processing time required for the model to run was not justified by the performance achieved.

9 Conclusion

To close we review some challenges that we encountered in our project, as well as some problem insights. We then close with a summary.

9.1 Challenges & Problem Insights

We found that it was difficult to consistently show significant improvement in performance over the baseline fixed-effects model. While several of our models did beat the baseline, the gains were relatively small.

We think there are a few reasons for this. First of all, the baseline already contained the most important information in predicting the probability of a strikeout, the track record of the pitcher and batter. We knew at the beginning of this project that the baseline model itself was strong and that our goal was to use creative feature engineering and model building to extract any additional predictive power we could.

Secondly, our prediction task is probabilistic and there is inherent noise in this process. It may be that were are up against an upper bound of predictive accuracy because of the variance of the of this task. In order to test this hypothesis, we created a array of 10,000 random probabilities from .1 to .3, mimicking the common range of probabilities of strikeouts, and then created binary data using those probabilities. The original array of probabilities should be a ”perfect” probabilistic classifier of those data. And yet, these predictions still have a log loss of 0.4922. It is possible that there is not much room *even in principle* for us to have improved on the baseline model.

We faced several challenges in building and testing our models. We dealt with missing data and the challenge of merging two independent datasets. We also ran into the limits of time when it came to training and validating large numbers of models on large datasets. We were forced to search over smaller parameter grids than we would have liked and we not able to include all the models we had in our ensemble methods.

9.2 Next Steps

While we tried several approaches to beat the baseline models, we were by no means exhaustive.

Most of our modeling followed a program of fitting a parameter vector (in the case of perceptron, logistic regression) or an ensemble (random forest, extra trees, adaboost, and gradient boosting) on a set of training data (with hyperparameters chosen via cross validation on a dataset) and then using the fitted model to predict the results of a test set.

As a first approach there are other schools and methodologies we could try.

One such option is Bayesian estimation. Bayesian statistics involves a prior distribution as well as a likelihood which, when maximized, gives a full distribution over set of outcomes. In our case this would involve setting a prior probability measure (possibly Beta or Bernoulli) and then fitting a posterior. We could even include as our prior the probability of a strikeout on average over a given preceding window. This would allow us to reflect actual knowledge and perhaps prevent the models from having to “relearn” the baselines. A variation on this idea is to pursue empirical Bayesian estimation which uses the dataset to fit both the prior and a posterior has had some success in baseball analytics([10],[11]) and may have been useful here.

Another option is large-scale deep learning. The use of high-powered GPU-enabled computing for large neural networks has solved many previously difficult problems, such as image recognition. It is possible that something similar would happen in this case. Neural networks are also useful for learning features. The outputs of hidden layers can be used as “feature extractors” for other models, such as logistic regression. We could take a first step towards this using the outputs of the hidden layers of our multilayer perceptron models to add features to see if that helped.

There are also other interesting avenues that we could pursue that are not dedicated solely to outperforming the baseline. For example we could use regular season data to predict post-season performance for teams that make the playoffs. Another option would be to keep a real-time strikeout probability calculator that updates after each pitch during an at-bat.

9.3 Summary

We tried to solve the problem of probabilistically predicting strikeouts in Major League Baseball. Using MLB AtBat and PitchFX data from the 2012-2017, we engineered features regarding the pitcher, batter, and context of each plate appearance. We then tried several individual modeling approaches and then attempted to use these models to create ensemble methods. Our goal was to improve upon the performance of our baseline model, which was a fixed effects model containing features for batter, pitcher, and ballpark. In the end, several of our models narrowly beat this baseline, while others did worse. Interestingly the categorical features we tried — which were just ballpark and various one-hot encodings of game states — were less informative than the numeric features, suggesting that future research should focus on quantifying player attributes rather than the scenario in the game.

References

[1] https://www.fangraphs.com/library/principles/sample-size/.

[2] http://blog.revolutionanalytics.com/2016/08/roc-curves-in-two-lines-of-code.html.

[3] https://www.mathworks.com/help/stats/linear-mixed-effects-models.html?s_tid=gn_loc_drop&requestedDomain=true.

[4] https://en.wikipedia.org/wiki/Logistic_regression.

[5] https://en.wikipedia.org/wiki/Gradient_boosting.

[6] https://en.wikipedia.org/wiki/AdaBoost.

[7] https://en.wikipedia.org/wiki/Random_forest.

[8] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3–42, 2006.

[9] https://en.wikipedia.org/wiki/Multilayer_perceptron.

[10] "Understanding empirical bayes estimation (using baseball statistics)." http://varianceexplained.org/r/empirical_bayes_baseball/.

[11] W. Jiang and C.-H. Zhang, *Empirical Bayes in-season prediction of baseball batting averages*, vol. Volume 6 of *Collections*, pp. 263–273. Beachwood, Ohio, USA: Institute of Mathematical Statistics, 2010.

A Data Fields

AtBat Data

- date (string)
- stadium (string)
- pitcher (string)
- batter (string)
- outs (int)
- home_score (int)
- away_score (int)
- descr (string)
- inning (int)
- side (string)
- bases (string)

PitchFX Data

- date (string)
- stadium (string)
- inning (int)
- side (string)
- pitcher (string)
- pitch_count (int)
- batter (string)
- balls (int)
- strikes (int)
- ay (float)
- px (float)
- ax (float)
- sz_bot (float)
- vz0 (float)
- vy0 (float)
- pfx_x (float)
- type_confidence (float)
- z0 (float)
- tfs (float)
- pz (float)
- start_speed (float)
- az (float)
- zone (int)
- break_angle (float)
- spin_dir (float)
- end_speed (float)
- vx0 (float)
- sz_top (float)
- nasty (float)
- descr (string)
- pfx_z (float)
- break_y (float)
- pitch_type (string)
- tfs_zulu (string)
- x (float)
- spin_rate (float)
- y0 (float)
- break_length (float)
- y (float)
- x0 (float)
- on_1b (string)
- on_2b (string)
- on_3b (string)
- umpcall (string)
- outcome (string)
- offense_score (int)
- defense_score (int)

B Features

Numeric Features

- pitcherkrate: The proportion of his batters faced this season to date that the pitcher has struck out, regressed by averaging with 50 battters faced at league-average strikeout rate. (real)
- wind0pkrate: The difference between the proportion his previous 150 battters faced that the pitcher has struck out, and pitcherkrate (i.e. how much more or less the pitcher has been striking batters out recently.) (real)
- wind1pkrate: The difference between the proportion his previous 70 battters faced that the pitcher has struck out, and pitcherkrate (i.e. how much more or less the pitcher has been striking batters out recently.)(real)

- wind2pkrate: The difference between the proportion his previous 25 battters faced that the pitcher has struck out, and pitcherkrate (i.e. how much more or less the pitcher has been striking batters out recently.) (real)
- batterkrate: The proportion of his plate appearances this season to date in which the batter has struck out, regressed by averaging with 50 plate appearances at league-average strikeout rate. (real)
- wind0bkrate: The difference between the proportion his previous 150 plate appearances in which the batter has struck out, and batterkrate (i.e. how much more or less the batter has been striking out recently.) (real)
- wind1bkrate: The proportion his previous 60 plate appearances in which the batter has struck out (i.e. how much more or less the batter has been striking out recently.) (real)
- wind2bkrate: The proportion his previous 25 plate appearances in which the batter has struck out (i.e. how much more or less the batter has been striking out recently.) (real)
- score: Score of the pitching team minus the score of the batting team. (integer number of runs)
- cumss: The fraction of pitches thrown by the pitcher prior to this at bat which resulted in *swinging strikes*. Regressed by averaging in 25 pitches with league-average swinging strike rate. (Includes foul tips caught by the catcher.) (real)
- windss: The difference between the fraction of the last 50 pitches thrown by the pitcher prior to this at bat which resulted in swinging strikes and cumss. (How much more or less than usual has the pitcher been getting swinging strikes.) (real)
- cumss_batter: The fraction of pitches seen by the batter prior to this at bat at which he swung and missed. Regressed by averaging in 25 pitches with league-average swinging strike rate. (Includes foul tips caught by the catcher.) (real)
- windss_batter: The difference between the fraction of the last 50 pitches seen by the batter prior to this at bat which resulted in swinging strikes and cumss_batter. (How much more or less than usual has the batter has been swinging and missing.) (real)
- cumcs: The fraction of pitches thrown by the pitcher prior to this at bat which resulted in *called strikes*. Regressed by averaging in 25 pitches with league-average called strike rate. (real)
- windcs: The difference between the fraction of the last 50 pitches thrown by the pitcher prior to this at bat which resulted in called strikes and cumcs. (How much more or less than usual has the pitcher been getting called strikes. This can perhaps capture the influence of the current catcher.) (real)
- cumcs_batter: The fraction of pitches seen by the batter prior to this at bat that he took for strikes Regressed by averaging in 25 pitches with league-average called strike rate. (real)
- windcs_batter: The difference between the fraction of the last 50 pitches seen by the batter prior to this at bat which resulted in called strikes and cumcs_batter. (How much more or less than usual has the batter has been taking pitches for strikes.) (real)

- cumO_swing: The fraction of pitches thrown by the pitcher so far this season that were outside the legal strike zone that resulted in strikes or contact. Regressed by averaging in 25 pitches with league-average rate of strikes outside the zone. (Includes swings ouside the zone and blown calls by umpires.) (real)
- windO_swing: The difference between the fraction of last 50 pitches thrown by the pitcher that were outside the legal strike zone that resulted in strikes or contact and cumO_swing. (How may more or fewer strikes outside the zone the pitcher has been getting recently.) (real)
- cumO_swing_batter: The fraction of pitches seen by the batter so far this season that were outside the legal strike zone that resulted in strikes or contact. Regressed by averaging in 25 pitches with league-average rate of strikes outside the zone. (Includes swings ouside the zone and blown calls by umpires. Proxy for how often the batter swings at balls) (real)
- windO_swing_batter: The difference between the fraction of last 50 pitches seen by the batter that were outside the legal strike zone that resulted in strikes or contact and cumO_swing_batter. (real)
- cumnasty: The average PitchFX "nasty factor" of the pitcher's pitches this season prior to this at bat. Regressed by averaging in 25 pitches with league-average nasty factor. (real, roughly 100 point scale)
- windnasty: The difference between the average PitchFX "nasty factor" of the pitcher's previous 50 pitches andcumnasty. (real, roughly 100 point scale)
- cumfbsspeed: The average starting speed (velocity) of the pitcher's fastballs this season prior to this at bat. All fastball features include four-seam and two-seam fastballs and sinkers. This and the following quantitites are not regressed, since they should stabilize fairly quickly. (real, in mph)
- windfbsspeed: The difference between average starting speed (velocity) of the pitcher's last 20 fastballs, and their normal average velocity. A smaller window of 20 pitches is used for fastball data since we want to capture a smaller window, e.g. the last few innings, to see if a pitcher is tiring. (How much harder or softer than usual is the pitcher throwing?) (real, in mph)
- cumfbespeed: The average end speed (velocity) of the pitcher's fastballs this season prior to this at bat. (real, in mph)
- windfbsspeed: The difference between average end speed (velocity) of the pitcher's last 20 fastballs, and their normal average velocity. (How much harder or softer than usual is the pitcher throwing?) (real, in mph)
- cumfbspin: The average spin rate of the pitcher's fastballs this season prior to this at bat. (real, in rpm)
- windfbspin: The difference between the average spin rate of the pitcher's last 20 fastballs and their overall spin rate (real, in rpm)
- spinvsavg: The absolute difference between the pichers fastball spin rate and the league average spin rate (real, in rpm)
- cumfbpfx_x: The average horizontal movement of the pitcher's fastball. (real, in inches)

- windfbpfx_x: The difference in average horizontal movement of the pitcher's fastball over the last 20 fastballs. (real, in inches)
- cumfbpfx_z: The average vertical movement of the pitcher's fastball. (real, in inches)
- windfbpfx_z: The difference in average vertical movement of the pitcher's fastball over the last 20 fastballs. (real, in inches)

Categorical Features

- AT&T Park - Yankee Stadium: Binary features for each of the 30 parks in full time use each season. Excludes special event locations.
- 1b, 1b2b, 1b2b3b, 1b3b, 2b, 2b3b, 3b, E: Binary features for each of the possible baserunner-states.
- bottom, top: binary features of the top or bottom of the inning
- inn_1 - inn_12: Binary feature for the current innning, all extra innings are counted as the 12th inning. This was done to make the number of features consistent and avoid noise from small samples.
- outs_0 - outs_2: binary features for the number of outs.
- LL, RR, LR, RL, etc.: Binary features for each possible handedness matchup.