Toronto Blue Jays 2021-01 Questionnaire Responses (Baseball Research Analyst)

By Dan Goldberg, 2021-01-19
Project Github Repository

Question 1: Which shortstop converted the most outs above average?

The leader in Outs Above Average (OAA) in this dataset was playerid 162066 with 11.09 outs above average. I used a fully Bayesian model (Multilevel Logistic Regression) so the leaderboard below has OAA_mean (the point estimate of OAA) and OAA_std (the uncertainty around that estimate) instead of just OAA. I've also added OAA_per_Opp to give a rate stat next to the counting OAA stat.

rank	playerid	opportunities	OAA_mean	OAA_std	OAA_per_Opp
1	162066	227.9	11.09	4.83	0.049
2	162648	196.6	9.15	4.04	0.047
3	197513	90.3	4.46	1.91	0.049
4	154448	225.9	4.34	3.78	0.019

rank	playerid	opportunities	OAA_mean	OAA_std	OAA_per_Opp
5	9742	97.6	4.22	1.8	0.043
6	2950	151.2	4.11	2.73	0.027
7	168314	178.1	4.07	3.1	0.023
8	5495	226.8	3.98	3.78	0.018
9	9148	155.7	2.88	2.65	0.018
10	9074	181	2.81	3.19	0.016
11	162294	50	2.67	1.13	0.053
12	132551	200.5	2.65	3.5	0.013
13	161551	231.2	2.36	3.51	0.01
14	171164	13	2.02	0.24	0.155
15	5419	163.9	2.02	2.74	0.012
16	7580	121.8	1.64	1.97	0.013
17	167746	26.6	1.48	0.56	0.056
18	11742	214.5	1.22	3.15	0.006
19	184486	9	1.17	0.25	0.129
20	121615	31.8	1.13	0.6	0.036

/* note that this metric excludes outs above average added due to increasing the likelihood of a doubleplay or from tagging basestealers. I decided not to try to handle those special, and much tougher, cases; the available data is insufficient to quanitfy those aspects of making outs. Instead, this model only looks at whether the SS got the out making the primary play, being the first to touch the ball.

1 - Methodology

To estimate the OAA metric I decided to follow the lead of Tom Tango and the Statcast team and train a probabilistic model to predict the likelihood of an average shortstop making an out on a given play. As they explain, using this probability (i.e. 70%) we can then take the observed outcome as a binary variable (0 or 1) and interpret the shortstop of converting that original probability into either 0% (failure) or 100% (success). For example if the shortstop makes the play they would be adding the difference for that particular observation (100% - 70% = 30%) and get that amount of credit (i.e. 0.3 outs above average).

So the plan was to train a probabilistic model to learn some function from a multivariate input (some features of a given play) to a probability of an average shortstop making an out. To do this I wanted to experiment with different kinds of model families, and different hyperparameter configurations within model families. The space of potential models would be those which could

output some probability with respect to a binary outcome, and so each potential model could be optimized against a log-loss objective.

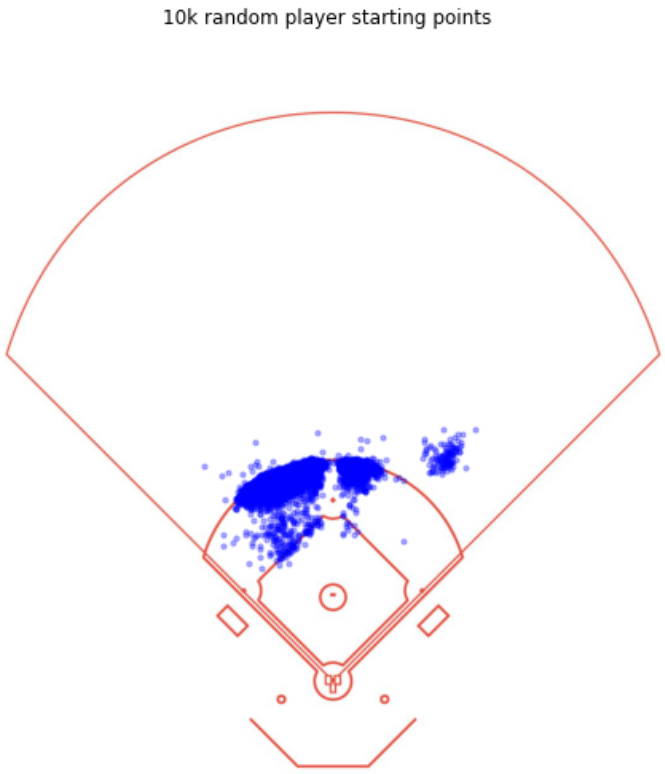
It being my strongest programming language and having a rich statistical ecosystem second only to R, I programmed in python, mostly leveraging scikit-learn.

1.1 - The Dataset

In exploring and preprocessing the dataset I had two goals in mind.

The first goal was to find an appropriate subset of all groundballs that the shortstop should be evaluated on. For example, when the shortstop is on the 3B side of the infield, a groundball to 1B really has nothing to do with the SS (in most cases) and so the SS should not get any credit or blame for an out being or not being made. The SS should only be evaluated on plays the could have potentially had some impact on. I decided as a starting point to filter out any plays that were made by another infielder (i.e. the 3B cuts in front of the SS and then makes the out at first) or any plays that were not turned into outs. This would still include some plays that were completely out of the SS's reach, though I address this in the design of input features for the model.

The second goal was to turn the raw dataset into input features that would describe the difficulty of the groundball (from the SS perspective) in such a way that was agnostic to where the SS was initially positioned. Doing so would require capturing the relative positions of the batted ball and the shortstop no matter where the SS stood. As it turns out, the starting position for a SS varies quite a bit, as seen in this plot of the SS starting position on 10k random groundballs.

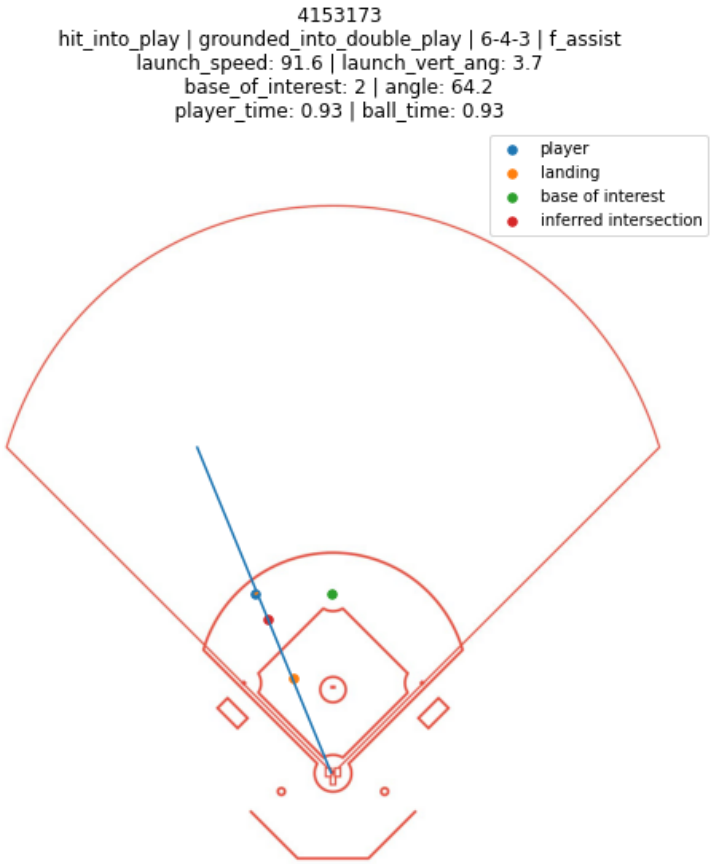


I decided that there is a lot of information about the difficulty of the play in the batted ball trajectory. From both Fangraphs and Statcast, OAA decomposes a defensive play into the

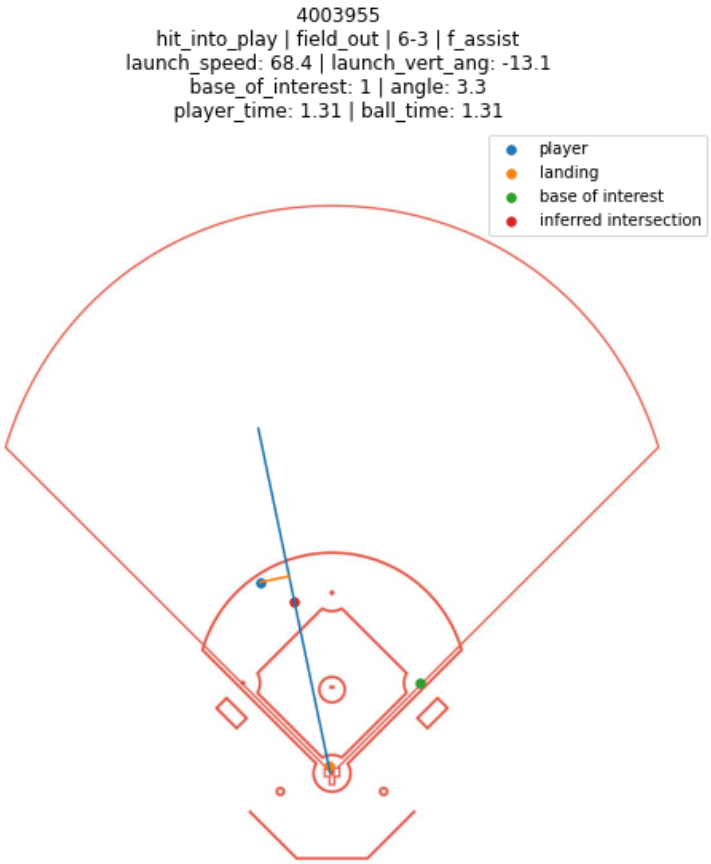
following components of a play: positioning, range, fielding, and throwing, with positioning being a component that the team gets credit for rather than the player. By utilizing the batted ball trajectory I thought I could create some features that captured aspects of range, fielding, and throwing, though certainly imperfectly. The following were some intermediate features I designed using this data:

- The orthogonal projection from the player position vector to the span of the launch trajectory. This gives the absolutely smallest distance the player would have to move to get in the ball's path (albeit possibly after the ball has passed that point).
- The inferred interception point, if it exists - the point that minimizes the player's time to the ball's path. This assumes the player moves at 21 ft/s, which is less than the ~25ft/s many players can sprint at, according to Statcast (though still a gross simplification). I used calculations outlined in the Appendix to solve this minimization equation.
- The assumed "base of interest" for the player, depending on the runners on base and whether they are in motion or not. This is a prerequisite for the next feature.
- The angle to the base of interest at either the inferred interception point, or the orthogonal projection point if that doesn't exist. This is intended to capture an aspect of the player's momentum towards or away from the base the will throw to.
- The time it takes to get to the intersection point (or orthogonal projection point), where the hypothesis is that less time is better, and gives the defender more time to complete the play.
- The difference between the ball's time to the interception point and the SS's time to the interception point. If the ball takes much less time to reach that interception point then it is very unlikely the SS can actually field the ball - this should be quite a reliable indication of whether the play is well outside the fielder's range or not.

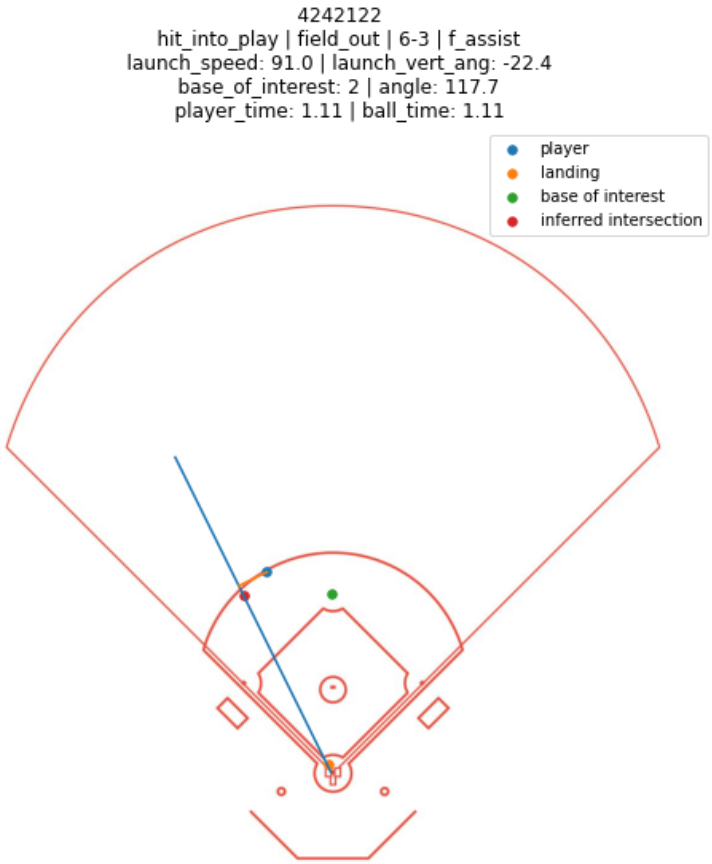
I created a visualization tool while exploring the dataset and developing these features which makes these descriptions a bit clearer. The following are some examples of groundball observations from the dataset, which are considered evaluation datapoints for the SS:



In this example you can see the player's starting position as the blue dot, and the span of the ball's launch trajectory as the blue line. The orange dot is where the ball landed, and the red dot is where the player could intercept the ball to minimize time to the ball - in this case since the player is already on the ball's trajectory it would just mean charging the ball. The green dot is indicating that 2B is the base of interest in this case, meaning there's a runner on 1B and a force play at 2B (plus the runner is not in motion).

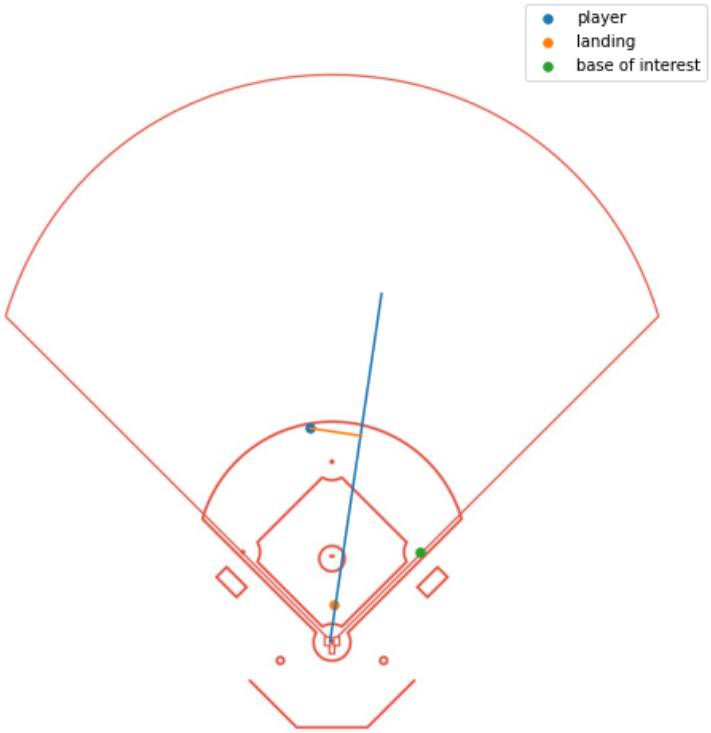


In the second example you can see a similar situation, except now the ball's trajectory is to the player's left. There is still a red dot indicating there is an inferred interception point. However in this example it is easier to see the orange line, which is the shortest path to the ball's trajectory span, orthogonal to the trajectory. The "angle to the base of interest" feature captures the angle between the green, blue, and red dots, with the blue (player position) as the vertex. In this case the angle is very acute (close to 0) and the player's momentum will be carrying him towards the base as he intercepts the ball at the red dot.

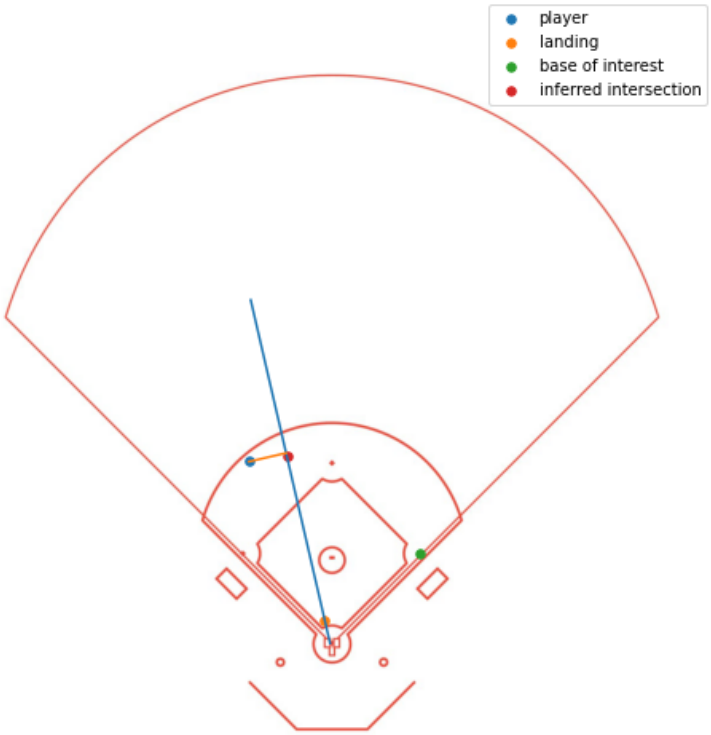


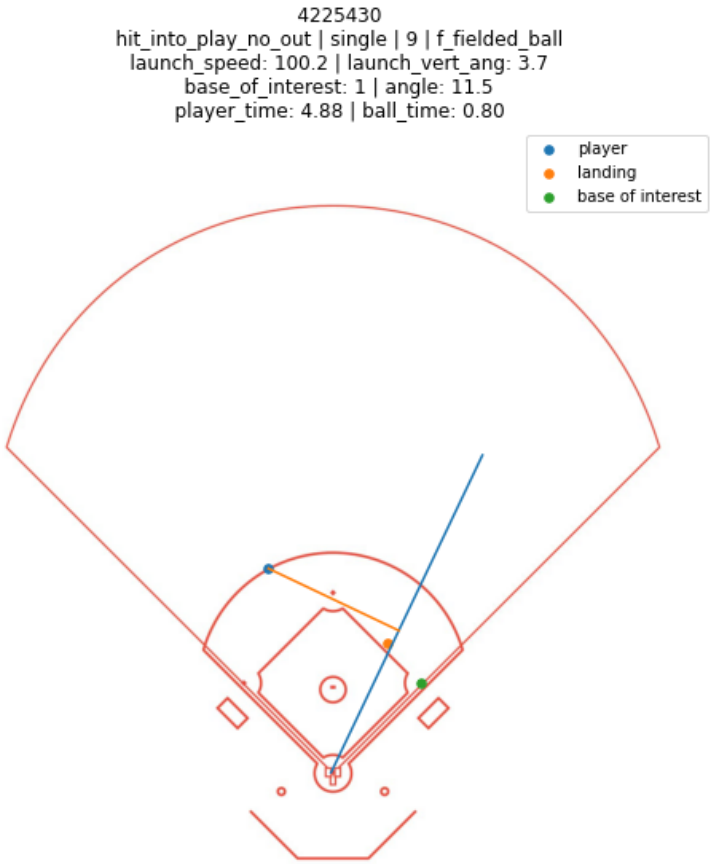
This example illustrates a tough play in the hole, where the angle is > 90 degrees to 1B.

4054482
hit_into_play_no_out | single | 9 | f_fielded_ball
launch_speed: 106.3 | launch_vert_ang: -3.7
base_of_interest: 1 | angle: 40.2
player_time: 1.72 | ball_time: 1.00



4052854
hit_into_play_no_out | field_error | 6 | f_throwing_error
launch_speed: 75.6 | launch_vert_ang: -7.2
base_of_interest: 1 | angle: 35.8
player_time: 1.29 | ball_time: 1.29





Here are more examples, some of which have no red dot, meaning given the player speed assumption the player cannot intercept the ball at all. There are times when a player does intercept the ball despite this calculation not giving an inferred interception point, though this means the SS has likely made a very tough, high-range play.

All features were standard-scaled before being fed into each model, since some models (like Support Vector Machines) assumed scaled features. I started out also applying a log transform before scaling two of the features, but never progressed to experimenting with different feature scalings. With more time I would've run experiments on scaling and also on withholding certain features; one of my worries about the design matrix I used was high correlation between inputs, which might cause more difficult training for some of the models. Given the time constrains, though, I moved on with what I thought was a solid design matrix.

1.2 - Candidate Models

The most important prerequisite of this modelling exercise was to train a model to output probabilities. A simple classification or even confidence score would not suffice. For this reason there were three classes of models I initially considered:

1. Generalized linear or non-linear models with inverse logit output (Standard Logistic Regression, Logistic GAM, Logistic Multilevel Regression, Neural Network w/Sigmoid Output)

   These models can be trained on a log-loss (a.k.a. binary cross-entropy) objective which would likely calibrate their probability predictions pretty nicely (though less-so for neural

networks). My guess was that these models would do a better job with calibrated probabilities, but I was also concerned that for the linear models they wouldn't be able to learn how to handle any complex non-linear interactions between features that might exist. Still, the closer I can get my features to reflect the true causal graph, the better a linear model might perform. GAMs are very flexible and was my guess for the best option out of this family, though they can be quite manual in the design of the basis functions (I think) and I am not very experienced with GAMs. Multilevel models in particular are attractive because they implement partial pooling to decrease variance and improve the estimates on clusters (levels) with low sample size, and inherently reduce overfitting.

2. Ensemble of non-probabilistic classification models for bootstrapped probability score (i.e. Random Forest Decision Trees, Gradient Boosted Trees)

   Ensemble models have two things going for them - they are fast, and they can make highly accurate class predictions. The downside in this case is that they are not probabilistic, and so they have a good chance of being poorly calibrated in their probability predictions. The only reason they can output what can be used as probability predictions is because they are ensemble models, and so by averaging over multiple component models one can get a sample distribution on an estimate. Ensembles also help reduce overfitting in that many high-variance + low-bias models averaged together create one mid-variance low-bias model.

3. Other generative models (i.e. Naive Bayes, Mixture of Gaussians, Gaussian Process)

   I thought learning multivariate generative model like a Gaussian Process would been a good candidate since it's so flexible, and would possibly learn a well-calibrated probability, though these models are usually more time-expensive to train. Gaussian processes capture relationships between input vectors in addition to just a function from input to output, so as long as the density across the input space is relatively smooth the GP model should do pretty well, and output decent probability estimates. The rub with Gaussian Processes is in hand designing the kernel (the prior), which is a manual process and adds a lot of complexity when actually trying to make the most out of this model.

1.3 - Training Method

My approach is to pick a suite of models (those listed above) and for each, to determine a hyperparemeter space to search over, if such hyperparameters require tuning for the model family. My hyperparmeter tuning method of choice is Bayesian Optimization (modelling the loss as a function of hyperparams, usually with a Gaussian Process) since it can be so efficient in it's search through that space and intelligently picks the next configuration to try. For each model family I configured an experiment (model family, hyperparameter space, preprocessing pipeline) to automate a search for the best model as much as possible. I spent quite a lot of time programming utilities to help me execute this Bayesian Optimization plan, which in the end made configuring and running experiments extremely easy. The outer loop over hyperarameters just requires an objective to minimize, and I decided the inner loop would return the mean of the loss across 5-fold cross validation on the model at the given config.

The major exception to this plan was the Multilevel Logistic Regression model, which was a fully Bayesian model and required slightly different tooling, with no hyperparameters. For this model used the python API for Stan to define bespoke model and sample parameters using the NUTS MCMC optimizer. A typical Bayesian workflow would have me carefully select priors by first forward-simulating to output space, though I didn't do this proper workflow due to time constraints; I just did enough work to get the model to converge (which was harder than expected) and used generic priors.

One thing I didn't do but that would've been good to do for this kind of task was to train the non-probabilistic models using probability calibration cross validation. I left this out of scope out of concern for time, privileging the Bayesian Optimization piece over that to leverage automating experiments as much as possible.

1.4 - Model Evaluation & Selection

Each model configuration got a 5-fold cross validation score on the log-loss objective, which is the conditional probability of the output data given the model probabilities. It was my hypothesis that training to this objective aligns the model optimization with probability calibration as much as possible. Once all initial model configurations were trained I could read the results into a dataframe from the model registry (where I saved their details) to find the best model (by log-loss) in each experiment. I then compared each experiment's best model on a number of metrics including Brier score, accuracy, F1, as well as visually inspecting their probability calibration curves and their histogram of probability predictions. Finding the best couple models from this inspection, I then directly compared these two models, picking a winner with an eye on having the best probability estimates possible.

See the python notebook that follows the Appendix in this PDF for the code and graphics from model selection and OAA calculation.

---

2 - Code

My programming efforts focused on creating functionality to make useful visualizations of the data, a generic pipeline for training models, and a way to save models for evaluation and model selection. I also wanted to showcase how I would try to build experiment code for production. Please see the github repository for this project here.

2.1 - utils.preprocessing i.e. shortstop_global_preprocessing()

This function is where all the feature design and other preprocessing happens. This module has the functionality to load and then convert the raw data into a usable form for any modelling experiments. I decided to abstract away all this processing into a function since my starting point was to have all model experiments share the same features and preprocessing/scaling. This module heavily relies on the utils.geometry module, described below.

2.2 - utils.geometry i.e. likely_interception_point()

The linear algebra and calculus that I needed to calculate things like the orthogonal projection vector from the player's starting point to the ball's trajectory span, or the likely interception point for the player and ball given some basic assumptions. It was fun figuring out how to put

these calculations together - see the Appendix for the derivation of the likely interception point calculation.

2.3 - utils.ml_training i.e. ModelExperiment

Sets up a (model family, hyperparameter space, preprocessing pipeline) experiment configuration. Implements the training scheme for non-Bayesian models, including the inner loop k-fold cross validation, outer loop bayesian optimization, and model saving in the inner loop. Also allows for single k-fold CV scoring without Bayesian Optimization, or simply training a single model, and then saving the output to the model registry.

2.4 - utils.ml_utils i.e. ModelPersistance

Saves models from an experiment so it's easy to load the exact model from a pickle, along with meta-data of the model like parameter settings and the value of objective function at those settings. This makes it easy to check on the results of an experiment and load the best model to re-train and deploy. Includes a method to load the model from pickle, given the model ID.

2.5 - utils.viz_utils i.e. Diamond

Built as part of exploring the dataset, this class provides a convenient way to plot coordinates on a to-scale diagram of a generic baseball diamond, along with line segments for ball trajectory and player trajectory. I used an svg from this site and confirmed it is to-scale.

Question 2: In addition to what's included in the provided dataset, what variables or types of information do you think would be helpful in answering this question more effectively?

In gathering more information I'd be interested in doing three things:

1. Be able to model the probability of making a doubleplay.
2. Be able to estimate the probability of caught stealing at 2B on a stolen base attempt.
3. Improving the current model of SS making the primary play.

In the current dataset there isn't enough information to be able to model doubleplay probabilities. A prerequisite for that is knowing the inning outs before the start of the play - we need to know if doubleplay is even possible, or if there's two outs and hence no doubleplay will happen regardless. With that info we could try to model the doubleplay probabilities given the rest of the current information.

Other than that, there is additional information that would be useful for improving both a model of making a primary play and a model of successful doubleplays:

- Actual position and time of interception - would help break up components of range, field, and throw.
- Actual position of other fielders - would be an important step for modelling probability of doubleplay if the SS is the one turning the doubleplay.
- The handedness of the batter and the speed of the batter - to estimate the batters time to first base.
- The speed of the baserunners - to estimate the time to their destination base.

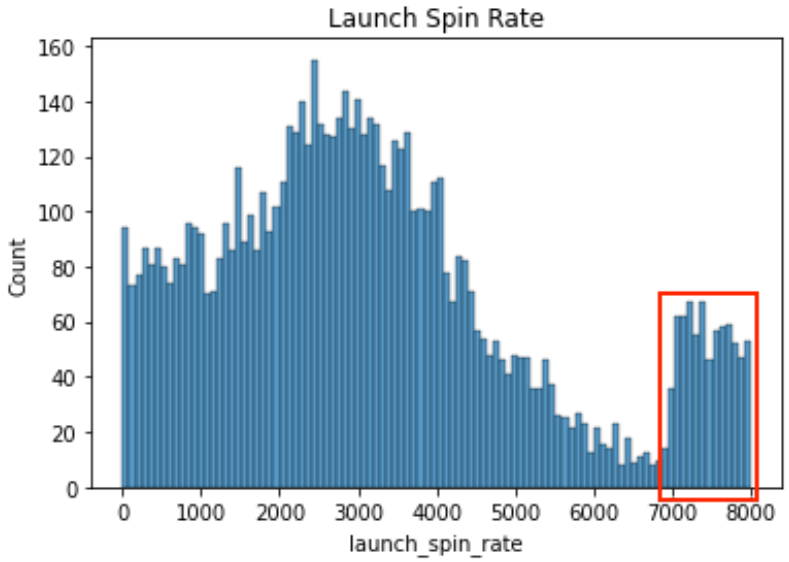
- The handedness of the pitcher - probably not super valuable.
- Accurate spin readings, plus 3D spin, not just 2D spin - balls will fly/bounce differently depending on the spin.
- The ballpark - they might have different hop profiles for groundballs due to different materials (i.e. turf), groundskeeping, design of infield.

For modelling OAA from tagging a baserunner we would need to create a model predicting the probability of an out when the SS is receiving a throw at a base and there is a baserunner running to that base. For this we would need to track the specific positions (x,y,z) of the ball as it moves towards the base as well as the positions of the baserunner as he moves towards the base (x,y). Then we could quantify the impact of Javier Baez turning into El Mago.

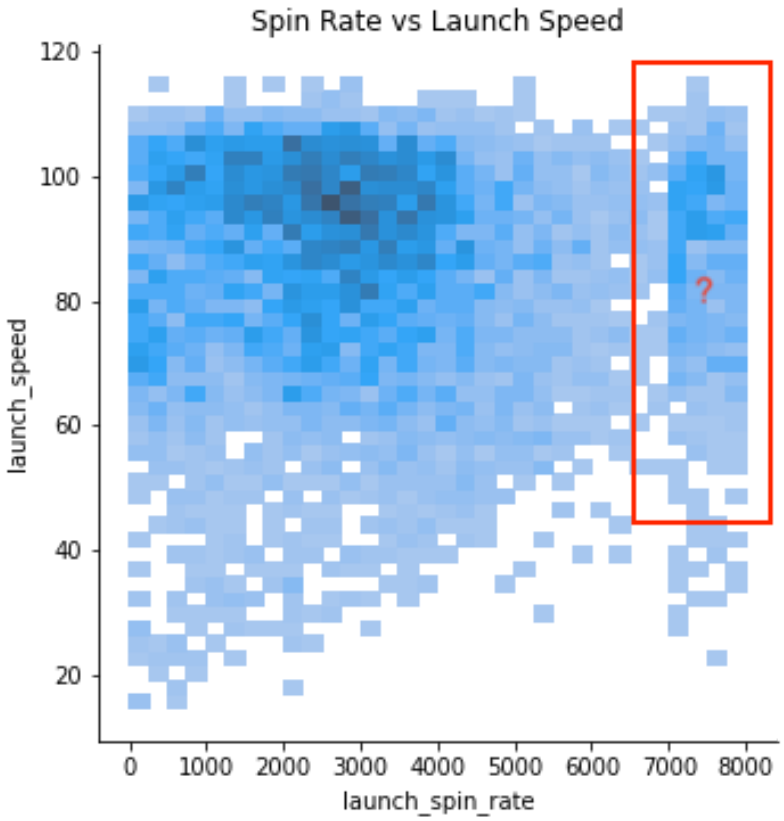
Question 3: Other than the final leaderboard, what is one interesting or surprising finding you made?

While exploring the shortstop defense dataset I became interested in the launch spin and launch axis measurements. These columns contain much missing data, and I thought that they might be useful for modelling the probability of the shortstop making an out - my hypothesis was that a groundball with top spin or side spin might behave much more erratically than a ball starting with backspin, and that more spin might make some plays like charging plays more difficult. I didn't end up using the spin values, but at the time I thought that I would try to impute the missing values in some way to try to utilize the useful information that the non-missing values might have for modelling the out probability. To do this I eventually created a model of spin using a few features I thought would be causally related to spin - launch speed and launch angle (horizontal and vertical).

First when I was exploring the dataset I noticed is that there is a weird concentration of launch_spin_rate between approximately 7000 and 8000 rpm. This concentration of density is actual discontinuous with the general, seemingly gaussian density around 3000 rpm. Here is a histogram of launch_spin_rate:

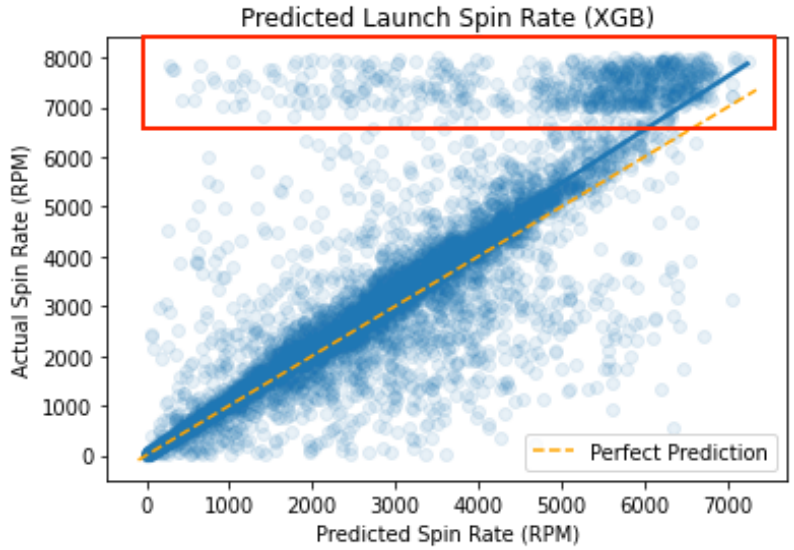


This > 7000 rmp density is so separated from the central density, that this is evidence of some separate causal process leading to these high spin measurements. Looking at spin rate compared to launch speed, we still see a clear second density.



Faulty measurement equipment is one potential theory, though I also thought that it could be caused by something on the field: if a pitcher were using a foreign substance like pine tar it might increase the likelihood of very high spin batted balls. Of course, that possibility could exist no matter what the pitcher does since a batter might contact the ball on a spot of the bat that has pine tar on it (hello George Brett!). These high-spin observations still could be caused by some combination of launch angle and speed, so I decided that I would try to create a model of launch_spin_rate as a function of other launch parameters (angle, speed, hang time, and distance). If the model was unable to predict this second mode at all, then it would be more evidence of a different mechanism at play causing these high spin measurements.

I trained an XGBoost Regressor to predict spin rate from the launch characteristics mentioned above, and this was the result:



And clearly, though the model did pretty well it still leaves a distinct mode between 7000 and 8000 rpm which isn't predicted with anywhere near the same accuracy as the rest of the range. This approach made me think that if a team were interested in figuring out which pitchers use high-grip foreign substance they could create a statistical model of spin rate and find the pitchers with the highest incidence of outliers, as in this case.

Having said this, my money would still be on equipment measurement error in this case, and I would investigate that hypothesis by looking for in-game correlations between spin rates, controlling for other factors. I didn't go down that route due to time constraints, though I decided that I would ignore the spin values in modelling shortstop OAA.

As an aside, I also thought about using Alan Nathan's Trajectory Calculator to model/impute the spin measurements, which might also be a good way to try to identify outliers.

Appendix

1 - Inferred Interception Point

Many important starting-point-agnostic features I built relied on having a particular point where the player might intercept the ball. To approximate where the player might intercept the ball (if possible) I used the following variables:

$s_{player} \rightarrow$ speed of player (ft/s)
$s_{ball} \rightarrow$ speed of ball (ft/s)
$\mathbf{p} \rightarrow$ starting position of player
$\mathbf{b} \rightarrow$
position of ball at interception (also the position of the player at interception)
$time_{player} \rightarrow$
time for player to move from starting position to interception position
$time_{player} \rightarrow$ time for ball to move from launch position to interception position

We are trying to solve the equation

$$time_{player} = time_{ball}$$

for unknown $\bold{b} $ if a real root exists. If no real root exists then the player cannot intercept the ball. For this derivation we will make the simplification that the player speed, s_{player}, and ball speed, s_{ball}, are constant. We know this is incorrect because the player will accelerate or decelerate over time, and the ball will decelerate due to air or ground friction. We also assume the ball doesn't move in a 3rd dimension (up and down) and that the speed is parallel to the ground.

$$time_{player} = \frac{\parallel \mathbf{b} - \mathbf{p} \parallel}{s_{player}}$$

$$time_{ball} = \frac{\parallel \mathbf{b} \parallel}{s_{ball}}$$

We will decompose the $\mathbf{b}$ vector into its component x, y dimensions since we can leverage knowledge about the relationship between these components assuming a straight-line ball trajectory.

For the ball vector, $\mathbf{b}$, we know that the span of this vector falls on $y = mx$ since it starts at the origin.

$$\parallel \mathbf{b} \parallel = \sqrt{b_x^2 + m^2 b_x^2} = \sqrt{(m^2+1)b_x^2}$$

Where

$$m = \frac{b_y}{b_x}$$

And, decomposing $\mathbf{p}$ in the same way:

$$\parallel \mathbf{b} - \mathbf{p} \parallel = \sqrt{(b_x - p_x)^2 + (mb_x - p_y)^2}$$

$$= \sqrt{b_x^2 - 2b_x p_x + p_x^2 + m^2 b_x^2 - 2mb_2 p_y + p_y^2}$$

$$= \sqrt{b_x^2 + m^2 b_x^2 - 2b_x p_x - 2mb_x p_y + p_x^2 + p_y^2}$$

$$= \sqrt{(m^2+1)b_x^2 - 2(p_x + mp_y)b_x + p_x^2 + p_y^2}$$

Now, substituting these into the full equation:

$$\frac{\sqrt{(m^2+1)b_x^2}}{s_{ball}} = \frac{\sqrt{(m^2+1)b_x^2 - 2(p_x + mp_y)b_x + p_x^2 + p_y^2}}{s_{player}}$$

Squaring both sides and rearranging, we get:

$$(\frac{s_{player}}{s_{ball}})^2 (m^2+1)x^2 = (m^2+1)b_x^2 - 2(p_x + mp_y)b_x + p_x^2 + p_y^2$$

Giving the quadratic equation of the form $0 = ax^2 + bx + c$:

$$0 = (1 - (\frac{s_{player}}{s_{ball}})^2)(m^2 + 1)x^2 - 2(p_x + mp_y)b_x + p_x^2 + p_y^2$$

where

$$a = (1 - (\frac{s_{player}}{s_{ball}})^2)(m^2 + 1)$$

$$b = -2(p_x + mp_y)$$

$$c = p_x^2 + p_y^2$$

which can be solved using the quadratic formula (np.roots in numpy).

2 - Model Comparison Jupyter Notebook

Please see below.

This notebook evaluates the results of the different model experiments and generates the OAA leaderboard

The goal of this notebook is to compare different models trained on the tbj2021 questionnaire groundball dataset, ultimately select one, and use it's probability predictions to evaluate Outs Above Average for the set of shortstops in the dataset. Since accurate probabilities are the key to the OAA metric, not only will I look at the log-loss for these models, but also the calibration of their probability output. A good probability calibration curve (y=x), minimal pathelogical behaviour from probability estimates (not missing values near 0 or near 1), and a comparatively good Brier score will determine which model I pick.

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt

from sklearn import metrics
    # metrics.brier_score_loss
    # metrics.log_loss
    # metrics.accuracy


# custom code found in this package
from ss_defense_experiment_main_preamble import experiment_prep
from utils.ml_training import ModelPersistance
from utils.viz_utils import Diamond, plot_single_sample
from utils.evaluation import train_model, summarize_model
```

Note that model experiments have already been run, so the decisions for feature design, set-up of models and choice of hyperparameters are not covered in this notebook. To see this part of the project, look at the `src/ss_defense_experiments_*.py` files for entrypoints (config) for model experiments. The `src/utils/` directory has the code for running a model experiment and saving it to the model registry.

Table of Contents:

1 - Grab results and summarize by model

The results of model training were saved in the `data/models/model_registry.jsonl` file, along with the config for each of the models trained. This code block ingests this data to compare models.

In [2]:
```
results = ModelPersistance.retrieve_registry_records(sorted_by='objective_value'
```

In [3]:
```
grouped = results.groupby('experiment_name')
grouped = grouped.agg({'id':'count', 'objective_value':[np.mean, np.std, np.min]
grouped
```

Out[3]:

	id	log-loss		
	count	mean	std	amin
experiment_name				
Random Forest	20	0.255637	0.036753	0.244740
SVM-RBF v1	20	0.271277	0.026757	0.258142
Gaussian Process	1	0.263653	NaN	0.263653
Gradient Boosted Trees	1	0.270949	NaN	0.270949
SVM-Poly v1	15	0.349089	0.075535	0.285957
Multilevel Logistic Regression v1	1	0.302501	NaN	0.302501
Logistic Regression	20	0.308613	0.009266	0.305087
GAM v1	1	0.316038	NaN	0.316038
Logistic Regression No Preprocessing (Baseline)	20	0.566157	0.000433	0.565985

The log-loss 'amin' column is the lowest log-loss for any model in that experiment. You can see that some experiments tried 15 or 20 different models (different hyperparameter settings), while others only trained one model. Some reasons only one model was trained for some experiments:

- Multilevel Logistic Regression v1 has no hyperparameters, and it is inappropriate to iteratively change priors to train a new model on the same data.
- Gaussian Process models automatically tune their hyperparameters in the sklearn libray.
- Gradient Boosted Trees used XGBoost which didn't play well with the Bayesian Optimization library I was using.
- GAMv1 was a particular combination of specific basis functions, and while there was definitely room to modify those basis functions, I ran out of time.

Notable from this table are the following observations:

- The 'dummy' baseline model performed much worse than the other models. The only difference between `Logistic Regression No Preprocessing (Baseline)` and `Logistic Regression` was preprocessing to the input matrix, so it's clear that this feature design had a big impact on the success of the models.
- Random Forest had a model that had the best log-loss of all the model families in the group, with rbf-SVM 2nd, and Gaussian Process 3rd. That being said it doesn't look like there was that much difference between the different model families, so the choice of which model to select will most likely come down to just the probability calibration evaluation.

- There is evidence that the model families leading in log-loss overfit, at least compared to the `Logistic Regression` model family. The std of log-loss between models in a given family show that there is higher variance and thus more overfitting for those model families.

2 - Load the best model for each family, and plot their probability calibration info

In [4]:
```
all_models = {}
```

In [5]:
```
%%time

for experiment_name in grouped.index:
    model_id = results[results['experiment_name']==experiment_name].sort_values(
    all_models[experiment_name] = {
        'id':model_id,
        'model_details': results[results['id'] == model_id],
        'model': ModelPersistance.load_model_by_id(model_id)
    }
all_models.keys()
```

```
CPU times: user 1.34 s, sys: 2.25 s, total: 3.59 s
Wall time: 6.69 s
```

Out[5]:
```
dict_keys(['Random Forest', 'SVM-RBF v1', 'Gaussian Process', 'Gradient Boosted
Trees', 'SVM-Poly v1', 'Multilevel Logistic Regression v1', 'Logistic Regressio
n', 'GAM v1', 'Logistic Regression No Preprocessing (Baseline)'])
```

Load data

When running the experiments I didn't save actual trained models (because I was evaluating mean log-loss on 5-fold CV) and instead just saved the settings for the models (with one exception). So I have to load and preprocess this data to train the models on the entire dataset. All the non-baseline models used the same preprocessing to make this step easier. In reality I would want to experiment with the featuresets but for the sake of this exersice I picked one preprocessing I like and went with it for all models.

In [11]:
```
%%time

df, X, y, feature_columns, target_name, index, param_payload = experiment_prep()
```

```
/Users/dangoldberg/miniconda3/envs/tbj2021/lib/python3.7/site-packages/pandas/co
re/indexing.py:1675: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_single_column(ilocs[0], value, pi)
/Users/dangoldberg/miniconda3/envs/tbj2021/lib/python3.7/site-packages/pandas/co
re/indexing.py:1596: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
```

```
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  self.obj[key] = value
CPU times: user 26.6 s, sys: 263 ms, total: 26.9 s
Wall time: 26.9 s
```

Logistic Regression

In [12]:

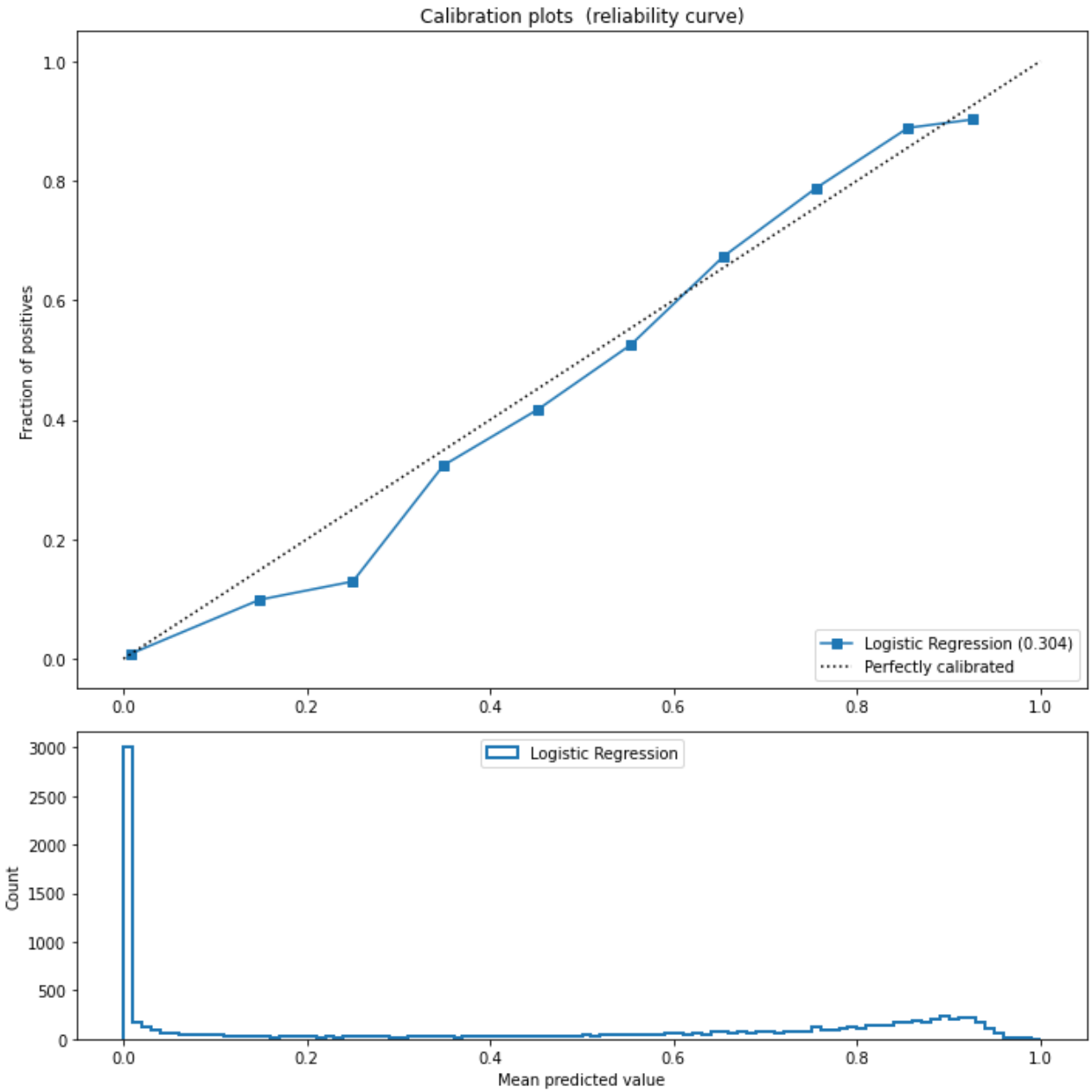
```
%%time

current_model_name = 'Logistic Regression'

# train model on all data
current_model, preds, preds_proba = train_model(all_models[current_model_name]['
```

```
CPU times: user 137 ms, sys: 7.22 ms, total: 144 ms
Wall time: 107 ms
```

In [14]:

```
all_models = summarize_model(all_models, current_model_name, y, preds, preds_pro
```

```
Logistic Regression:  0.092 Brier Score

        Precision: 0.809
        Recall: 0.927
        F1: 0.864
        Log-Loss: 0.304
        Accuracy: 0.875
```



What we see here is a really nice start. The calibration curve follows the diagonal really closely, and the probability outputs are spread nicely across [0,1] in the histogram. There is a little bit of pathelogical behaviour towards the 95%+ predicted value, where it seems there are very very few plays that qualify as that high probability. Instead it looks like the non-zero mode is around 90% or a bit higher. The huge spike at 0 is a result of the dataset containing plays that are well out of the SS's range, though not fielded by anyone in the infield so still included. The model can handle this well, though, thanks to the informative features in the input. All in all this is a great curve, with sudden jumps discontinuities in the frequency of any predicted probabilities in the domain.

Random Forest

In [24]:

```
%%time

current_model_name = 'Random Forest'
```

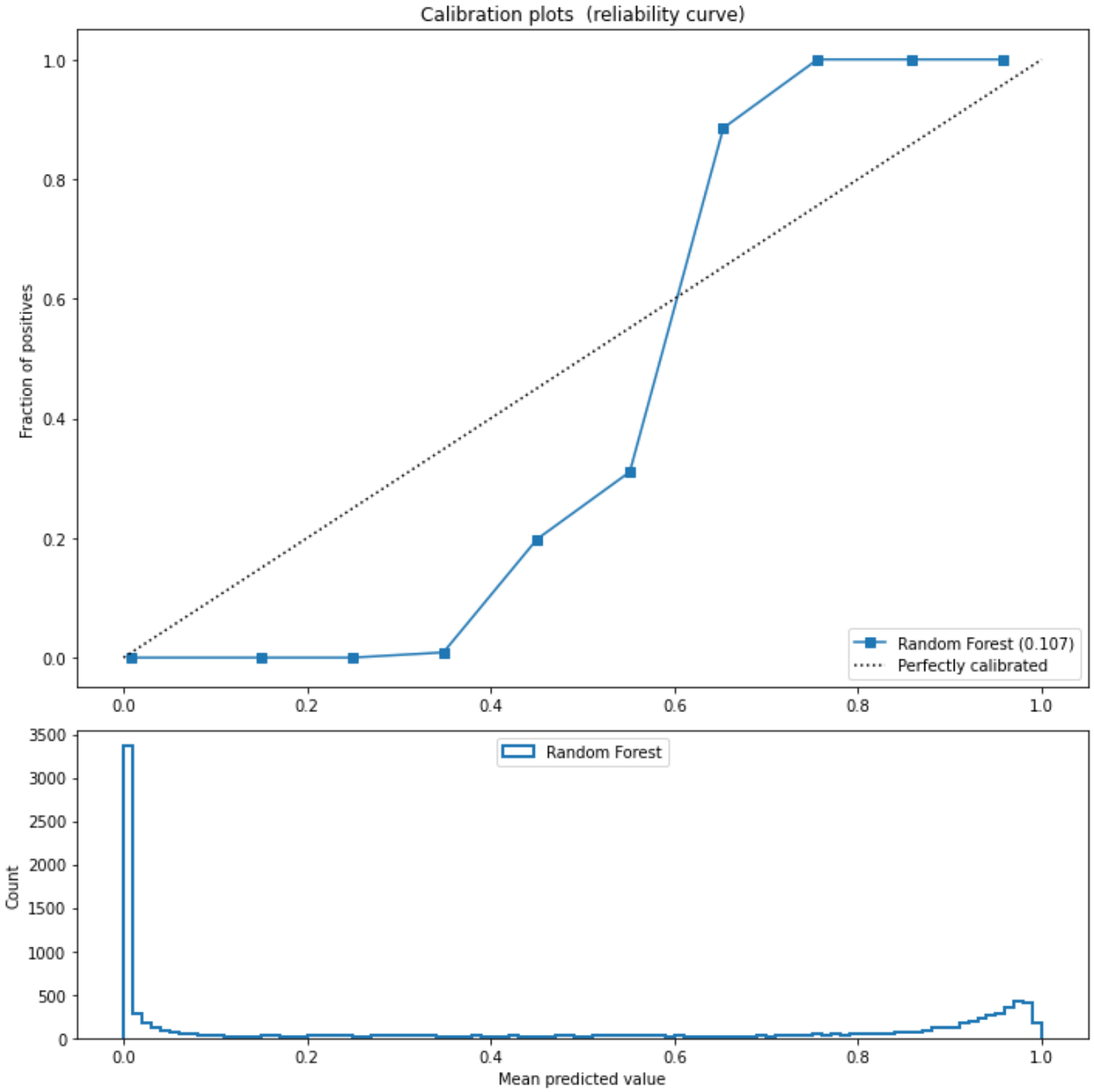
```
# train model on all data
current_model, preds, preds_proba = train_model(all_models[current_model_name]['
```

```
CPU times: user 33.9 s, sys: 320 ms, total: 34.3 s
Wall time: 35.6 s
```

In [25]:

```
all_models = summarize_model(all_models, current_model_name, y, preds, preds_pro
```

```
Random Forest:  0.026 Brier Score

        Precision: 0.949
        Recall: 0.989
        F1: 0.969
        Log-Loss: 0.107
        Accuracy: 0.973
```



This curve is way different than the previous one, and much worse from a calibration standpoint. The quality of outcome predictions in-sample is very high, as is traditional for random forest models, but that doens't mean much without a robust out-of-sample evaluation (which we're

not doing in this notebook). What we really want is a well calibrated probability prediction, and we definitely don't have that here. Predictions with probabilities of up to around 35% have around 0% chance of actually being successful outs, and the inverse is true of predcitions around 75% and up. This would wreck the OAA metric as a player's metric would be entirely at the mercy of what kind of opportunities they got, rather whether they made the most of their opportunities. We want to pass on this model. (note: there are strategies for improving calibration, including training it with the CalibratedClassifierCV class, which I didn't do here). The Brier score I think is misleading here because the accuracy is so high.

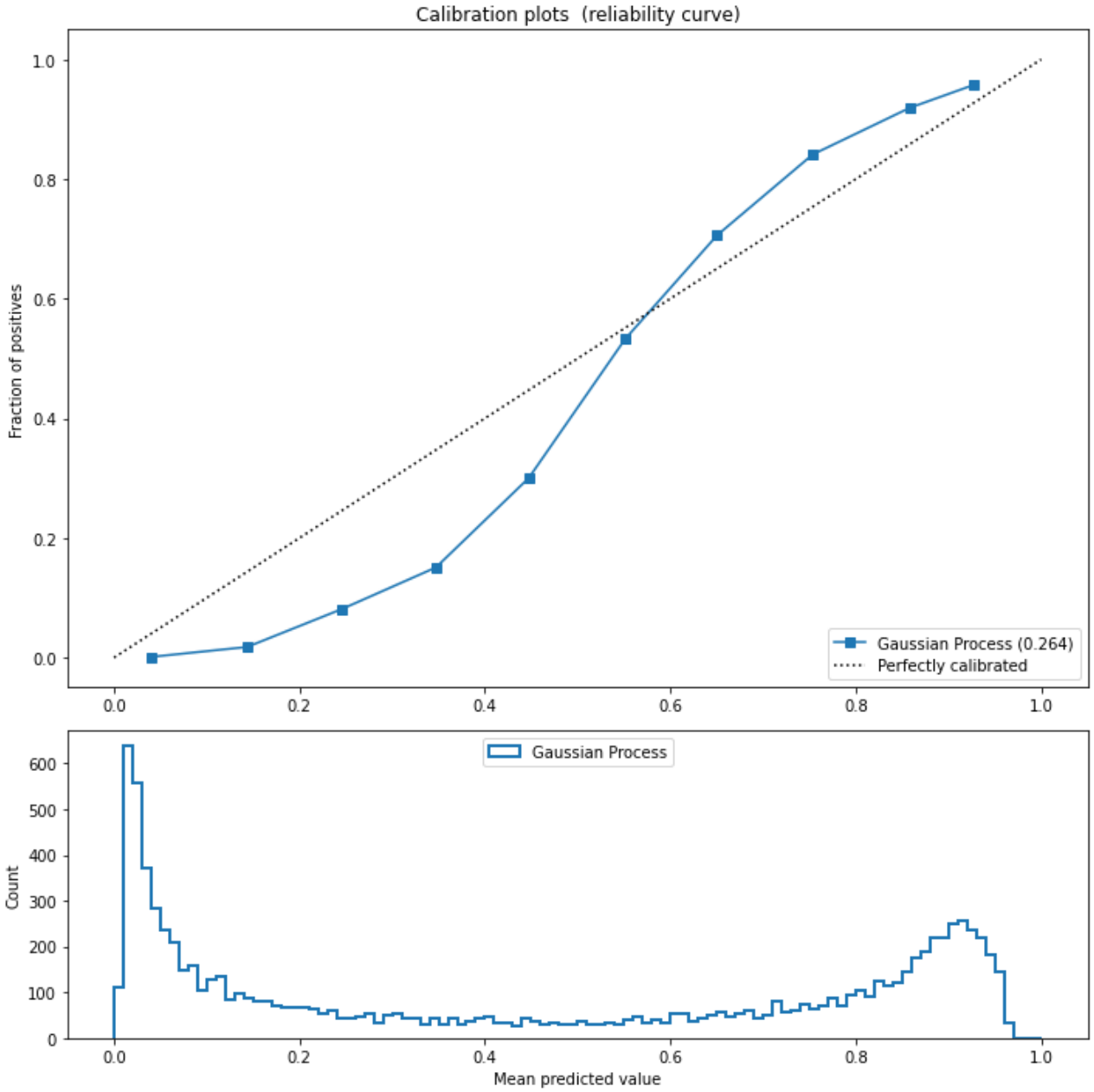
Gaussian Process

In [26]:
```
%%time

current_model_name = 'Gaussian Process'

# train model on all data
current_model, preds, preds_proba = train_model(all_models[current_model_name]['
```

```
CPU times: user 6min 48s, sys: 52.4 s, total: 7min 41s
Wall time: 6min
```

In [27]:
```
all_models = summarize_model(all_models, current_model_name, y, preds, preds_pro
```

```
Gaussian Process:  0.074 Brier Score

        Precision: 0.863
        Recall: 0.944
        F1: 0.902
        Log-Loss: 0.264
        Accuracy: 0.912
```



This has a really interesting histogram of probability predictions, since there is such little density at the 0% prediction unlike the other curves. This model outputs predictions with much more density between 20% - 80% than at the extremes. If I didn't know any better I'd think this was a good thing, but knowing the dataset, I know there should be a huge spike around 0% since there are so many balls that are just way out of the SS's range still in the dataset. Perhaps that being the case was a downfall for the Gaussian Process model, and that it would've done better if the dataset was more balanced than it is. The accuracy is impressive despite the smoothness at 0%, and this actually might be a really good candidate for another iteration after working on the dataset filtering.

SVM-rbf

In [28]:

```
%%time

current_model_name = 'SVM-RBF v1'
```

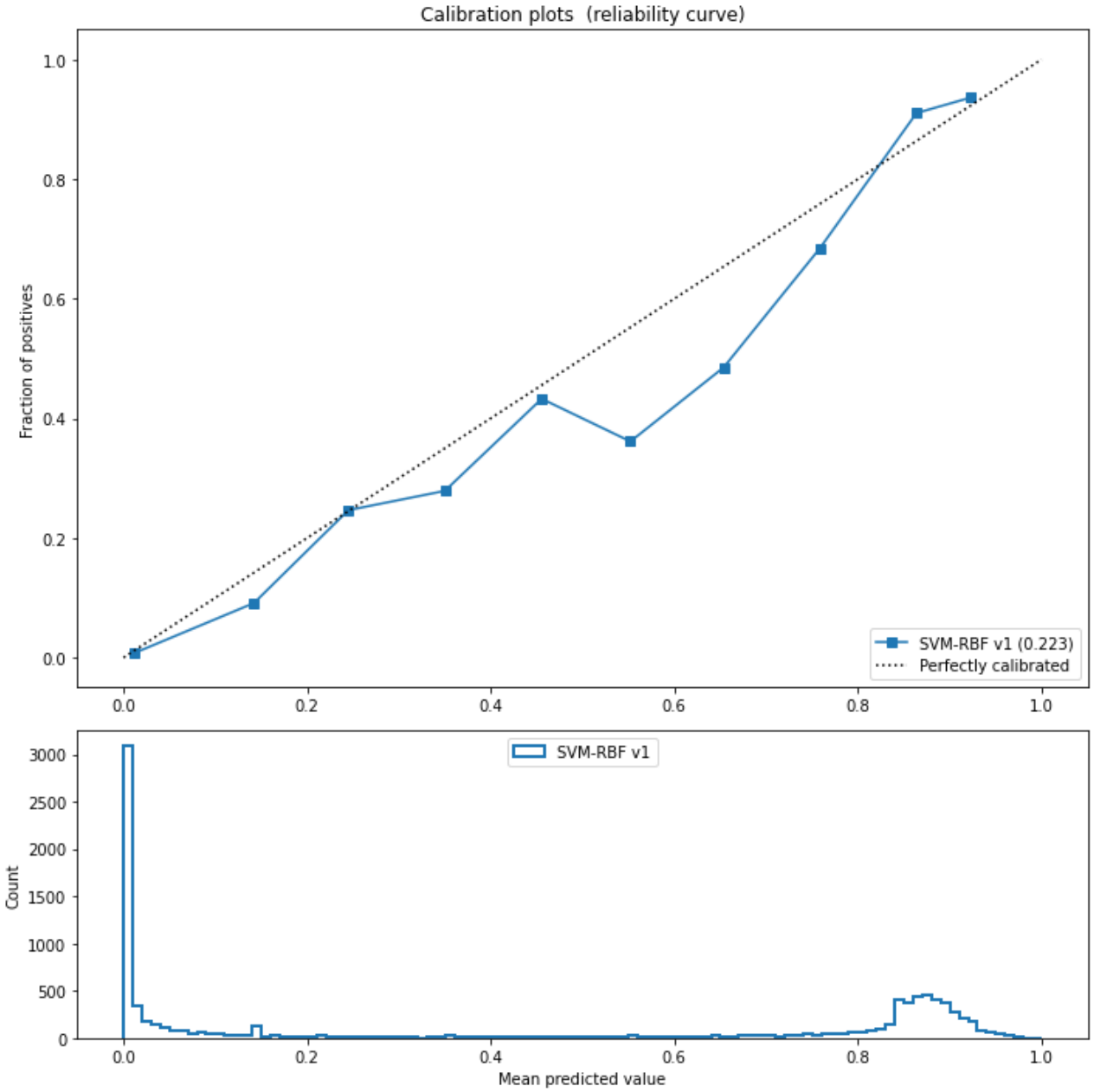
```
# train model on all data
current_model, preds, preds_proba = train_model(all_models[current_model_name]['
```

```
CPU times: user 22.7 s, sys: 422 ms, total: 23.1 s
Wall time: 24.9 s
```

In [29]:

```
all_models = summarize_model(all_models, current_model_name, y, preds, preds_pro
```

```
SVM-RBF v1:  0.065 Brier Score

        Precision: 0.855
        Recall: 0.954
        F1: 0.902
        Log-Loss: 0.223
        Accuracy: 0.911
```



The rbf-kernel SVM is a great all-purpose model since it's max-margin approach (with kernel) makes for a very flexbile algorithm that balances over-fitting and under-fitting well, and it can capture complex non-linear interactions between features. The metrics do show that this was a

good model, as the Brier and Log-Likelihood are very good, despite a small kink in the calibration curve (at areas of very low density). Part of why the calibration is good is because the scikit-learn implementation of this algorithm has built-in probability calibration during training. This leads me to believe that if I had used this probability calibration on other models (i.e. Random Forest) I would've gotten better performance from those models too.

This model would be the leader so far if not for two subtle pathologies with the predictions: at around 17% for the SVM (y-axis) there is a spike in frequency, which would indicate a pathology of a disproportionate amount of density being placed on the same exact prediction. There's another similar spike in density on the converse side of the prediction space, at around 83%, also noticable on the histogram. This is something I'd rather avoid.

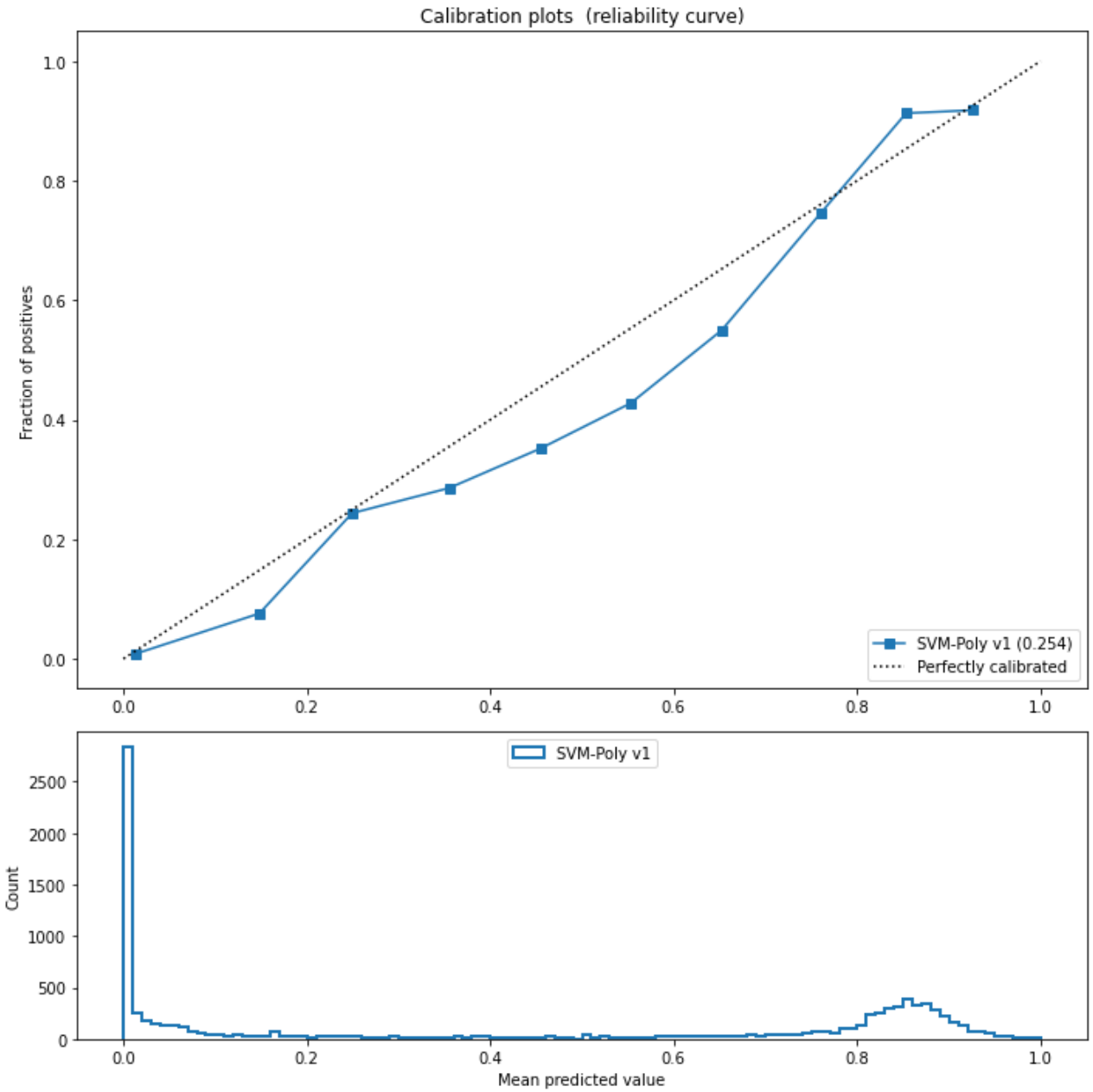
SVM-Poly

In [30]:
```
%%time

current_model_name = 'SVM-Poly v1'

# train model on all data
current_model, preds, preds_proba = train_model(all_models[current_model_name]['
```

```
CPU times: user 20.4 s, sys: 213 ms, total: 20.6 s
Wall time: 21 s
```

In [31]:
```
all_models = summarize_model(all_models, current_model_name, y, preds, preds_pro
```

```
SVM-Poly v1:  0.074 Brier Score

        Precision: 0.842
        Recall: 0.948
        F1: 0.892
        Log-Loss: 0.254
        Accuracy: 0.901
```



This looks a lot like the rbf-SVM, with maybe a little bit worse calibration (metrics and curve are a little bit worse), though without the bad pathology noticed above. This is the leader so far.

Gradient Boosted Trees

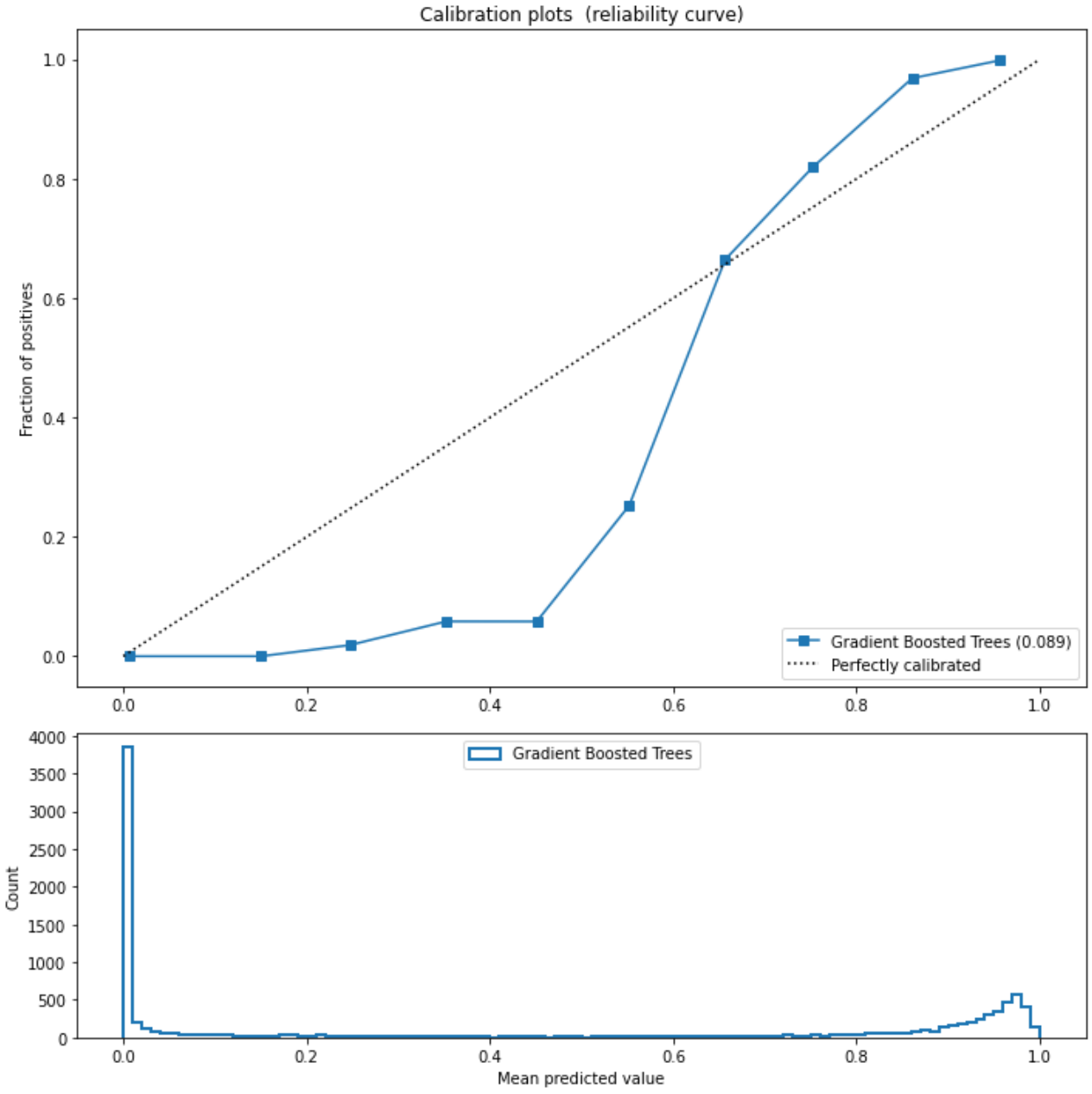
In [32]:

```
%%time

current_model_name = 'Gradient Boosted Trees'

# train model on all data
current_model, preds, preds_proba = train_model(all_models[current_model_name]['
```

```
[22:36:23] WARNING: /Users/travis/build/dmlc/xgboost/src/learner.cc:1061: Starti
ng in XGBoost 1.3.0, the default evaluation metric used with the objective 'bina
ry:logistic' was changed from 'error' to 'logloss'. Explicitly set eval_metric i
f you'd like to restore the old behavior.
CPU times: user 5.05 s, sys: 94.1 ms, total: 5.15 s
Wall time: 1.61 s
```

In [33]:
```
all_models = summarize_model(all_models, current_model_name, y, preds, preds_pro
```

```
Gradient Boosted Trees:  0.021 Brier Score

        Precision: 0.952
        Recall: 0.996
        F1: 0.974
        Log-Loss: 0.089
        Accuracy: 0.977
```



The XGBoost results look a lot like the Random Forest results, it's cousin algorithm. Basically the exact same output here, though the calibration curve is a lot better at high predicted probability. This would potentially be a really great model if I used probability calibration on it.

GAM v1

In [45]:
```
%%time

current_model_name = 'GAM v1'
```

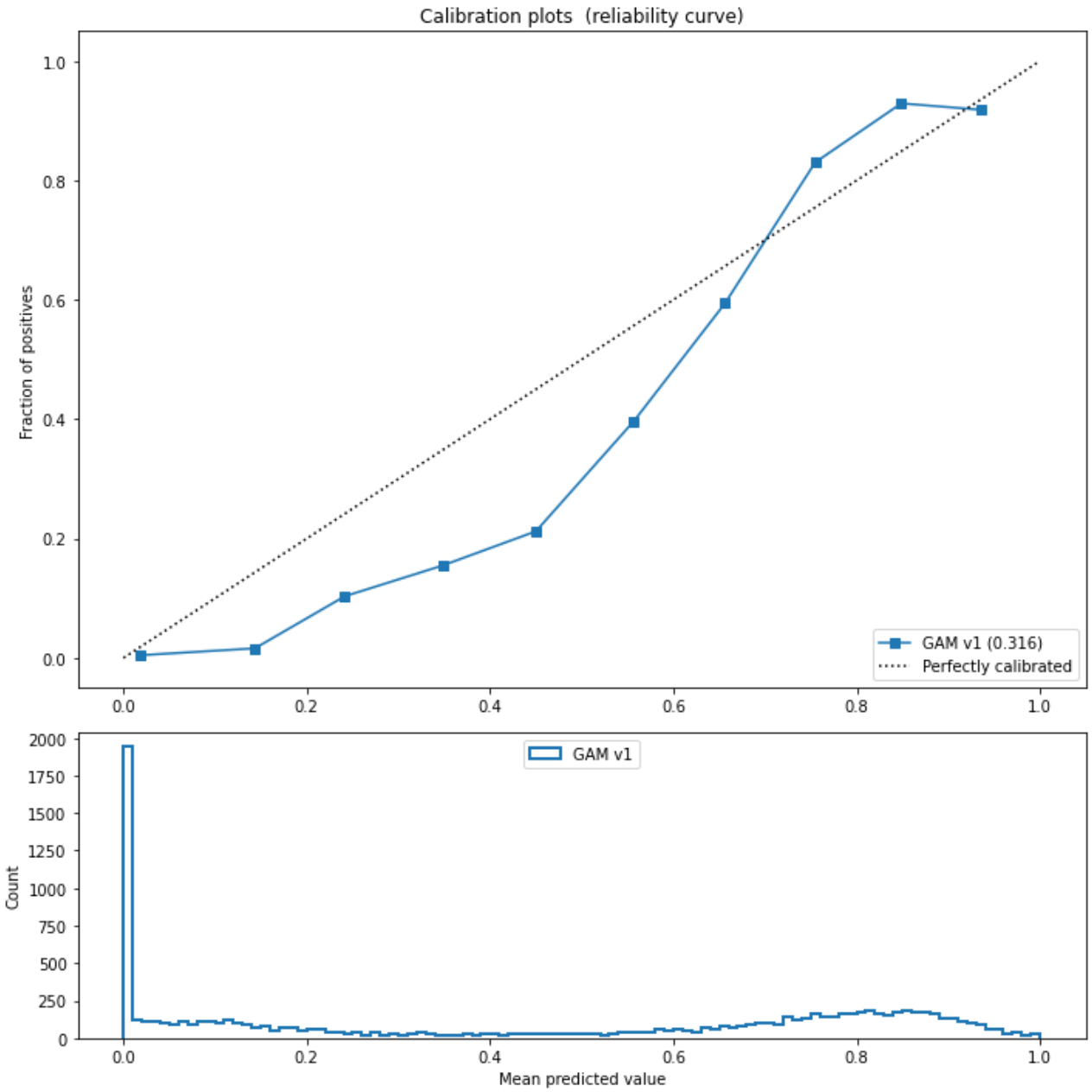
```
# train model on all data
current_model, preds, preds_proba = train_model(all_models[current_model_name]['
# clean up the predictions that give probabilities outside of [0, 1], and i'm no
preds_proba[preds_proba > 1.0] = 1.0
preds_proba[preds_proba < 0.0] = 0.0
```

```
/Users/dangoldberg/miniconda3/envs/tbj2021/lib/python3.7/site-packages/pygam/uti
ls.py:78: UserWarning: Could not import Scikit-Sparse or Suite-Sparse.
This will slow down optimization for models with monotonicity/convexity penaltie
s and many splines.
See installation instructions for installing Scikit-Sparse and Suite-Sparse via
Conda.
  warnings.warn(msg)
CPU times: user 1.2 s, sys: 178 ms, total: 1.38 s
Wall time: 1.07 s
```

In [46]:

```
all_models = summarize_model(all_models, current_model_name, y, preds, preds_pro
```

```
GAM v1:  0.088 Brier Score

        Precision: 0.807
        Recall: 0.961
        F1: 0.878
        Log-Loss: 0.316
        Accuracy: 0.885
```



The GAM doesn't look great here, with a comparitively high Brier and Log-Loss, with a bad calibration curve. There's almost certainly a lot of room to improve with tweaks to the basis functions, which I didn't experiment with.

Multilevel Logistic Regression v1

This prediction has to be done manually because the model is a bespoke pyStan model, not a scikit-learn model, and I didn't build a great wrapper for it (yet).

In [63]:

```
%%time

current_model_name = 'Multilevel Logistic Regression v1'

from scipy.special import expit
from sklearn.preprocessing import OrdinalEncoder

model = all_models[current_model_name]['model']
```

```
### preprocessing step

# reindex player id
oe = OrdinalEncoder(dtype=int)
df.loc[:, 'playerid_cat'] = oe.fit_transform(df[['playerid']])
levels = df['playerid_cat'].values + 1 # reindex with 1-index
num_levels = np.unique(levels).shape[0]

column_transformer = param_payload['feature_preprocessing']
X_t = column_transformer.fit_transform(X[feature_columns])

# train model on all data
# evaluate model
params = model.params  # return a dictionary of arrays
bias = params['bias'].mean(axis=0)
slope1 = params['slope1'].mean(axis=0)
slope2 = params['slope2'].mean(axis=0)
slope3 = params['slope3'].mean(axis=0)
slope4 = params['slope4'].mean(axis=0)
slope5 = params['slope5'].mean(axis=0)
slope6 = params['slope6'].mean(axis=0)
slope7 = params['slope7'].mean(axis=0)
slope8 = params['slope8'].mean(axis=0)
slope9 = params['slope9'].mean(axis=0)
slope10 = params['slope10'].mean(axis=0)
level_param = params['shortstop_effect'].mean(axis=0)

# get predictions
preds_proba = expit(
    bias \
    + slope1*X_t[:,0] \
    + slope2*X_t[:,1] \
    + slope3*X_t[:,2] \
    + slope4*X_t[:,3] \
    + slope5*X_t[:,4] \
    + slope6*X_t[:,5] \
    + slope7*X_t[:,6] \
    + slope8*X_t[:,7] \
    + slope9*X_t[:,8] \
    + slope10*X_t[:,9] \
    + [level_param[l-1] for l in levels]
)

preds = (preds_proba > 0.5).astype(int)
```

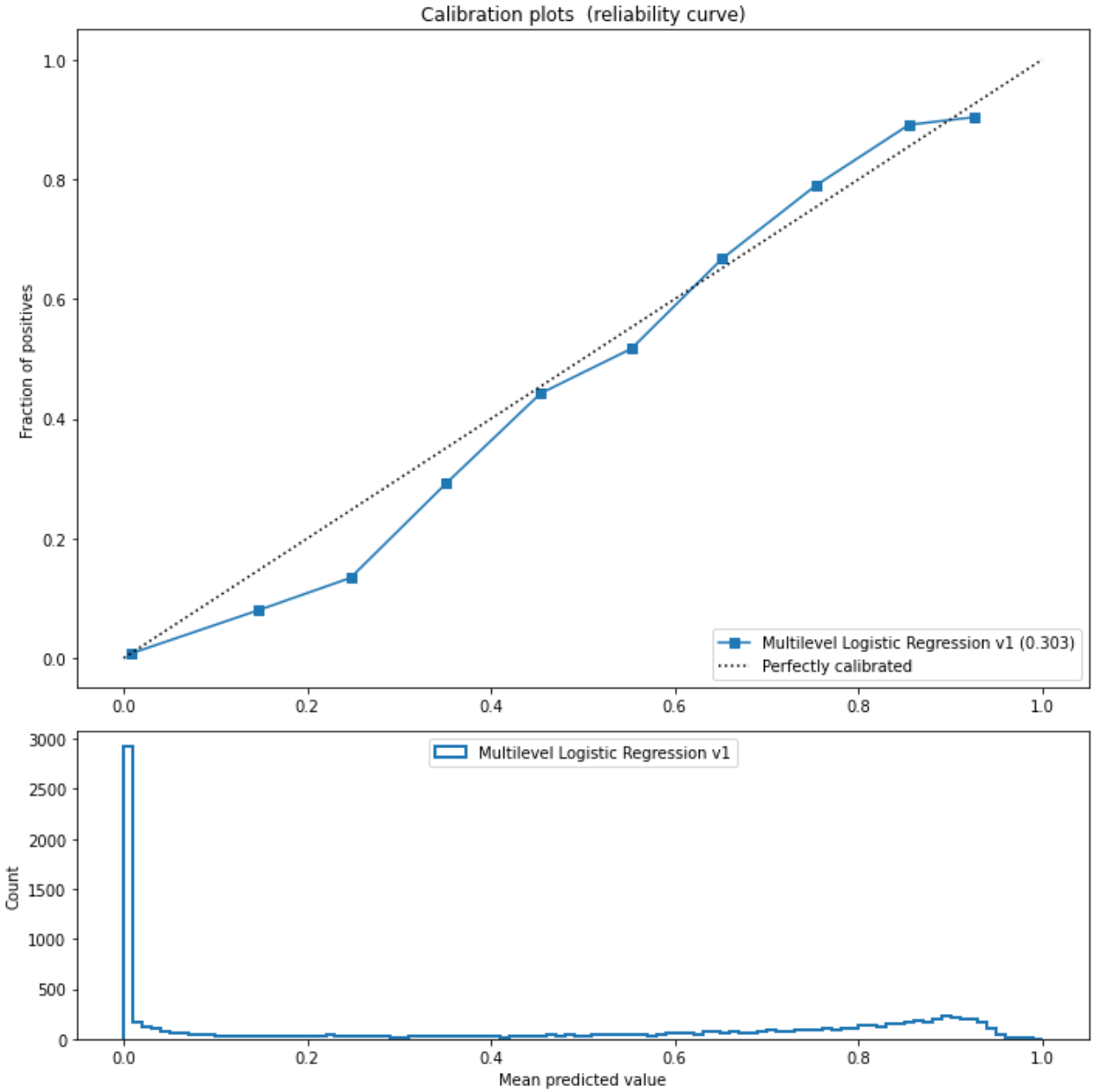
```
CPU times: user 53.4 ms, sys: 63 ms, total: 116 ms
Wall time: 186 ms
```

In [64]:

```
# convert preds_proba into sklearn style
sk_preds_proba = np.concatenate([ # Stan model only outputs prob of class 1
    1-preds_proba.reshape(-1,1),
    preds_proba.reshape(-1,1)
], axis=1)

all_models = summarize_model(all_models, current_model_name, y, preds, sk_preds_
```

```
Multilevel Logistic Regression v1:  0.092 Brier Score

        Precision: 0.810
        Recall: 0.926
```

```
F1: 0.864
Log-Loss: 0.303
Accuracy: 0.875
```



This is almost an exact carbon copy of the scikit-learn LogisticRegression model, which is both reassuring (that I made the Stan model correctly), and evidence that the partial pooling of using a variable intercept for each SS didn't do much. Though, the metrics are ever so slightly better than the vanilla Logistic Regression, so this one edges out that model.

Decision

I'll go with the SVM-Poly and Multilevel Logistic Regression models as the two finalists, due to the great accuracy, and very good calibraion of the SVM, and the decent accuracy with the excellent calibration curve of the Multilevel Logistic Regression.

3 - Compare best two models in more detail

We can look at some summary stats, as well as some spot checks of samples they disagreed on to look for any bad pathology we want to avoid.

In [101…

```
%%time

# the multilevel logistic regression variables are already loaded
glmm_model, glmm_preds, glmm_preds_proba = model, preds, preds_proba

# reload the svm variables
svm_model, svm_preds, svm_preds_proba = train_model(all_models['SVM-Poly v1']['m
```

```
CPU times: user 23.5 s, sys: 428 ms, total: 23.9 s
Wall time: 25.9 s
```

Samples w/ Biggest Dissagreements

In [102…

```
prob_differences = glmm_preds_proba - svm_preds_proba[:, 1]
```

In [103…

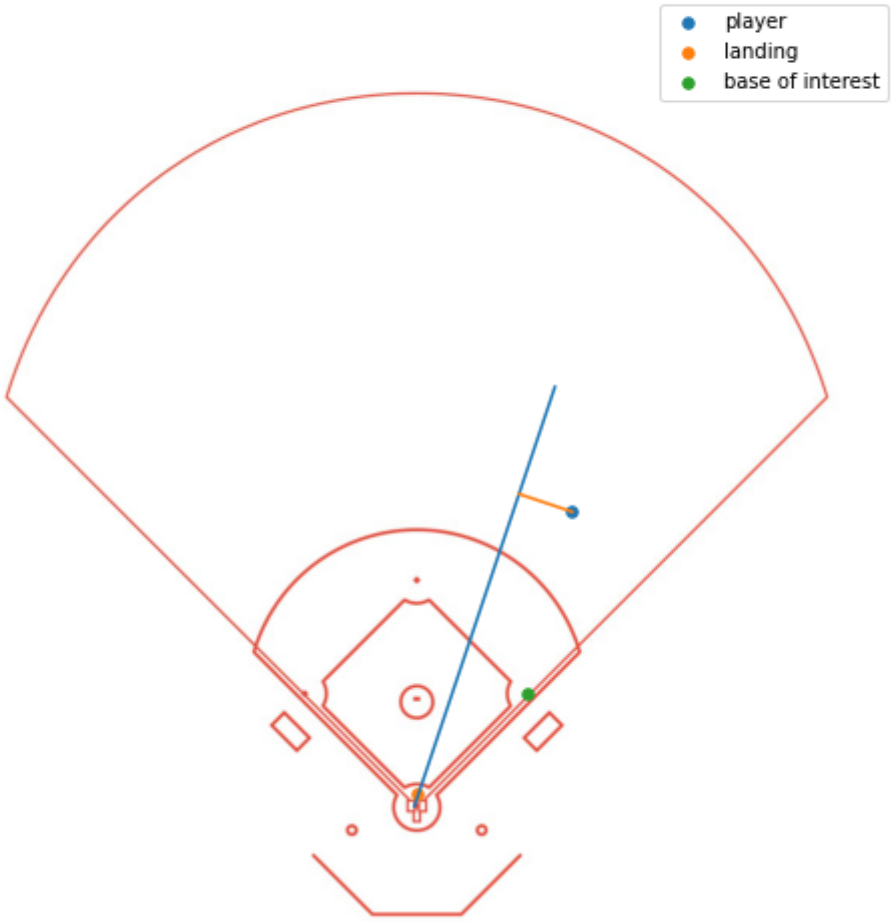
```
df.loc[:, 'prob_differences'] = prob_differences
```

In [104…

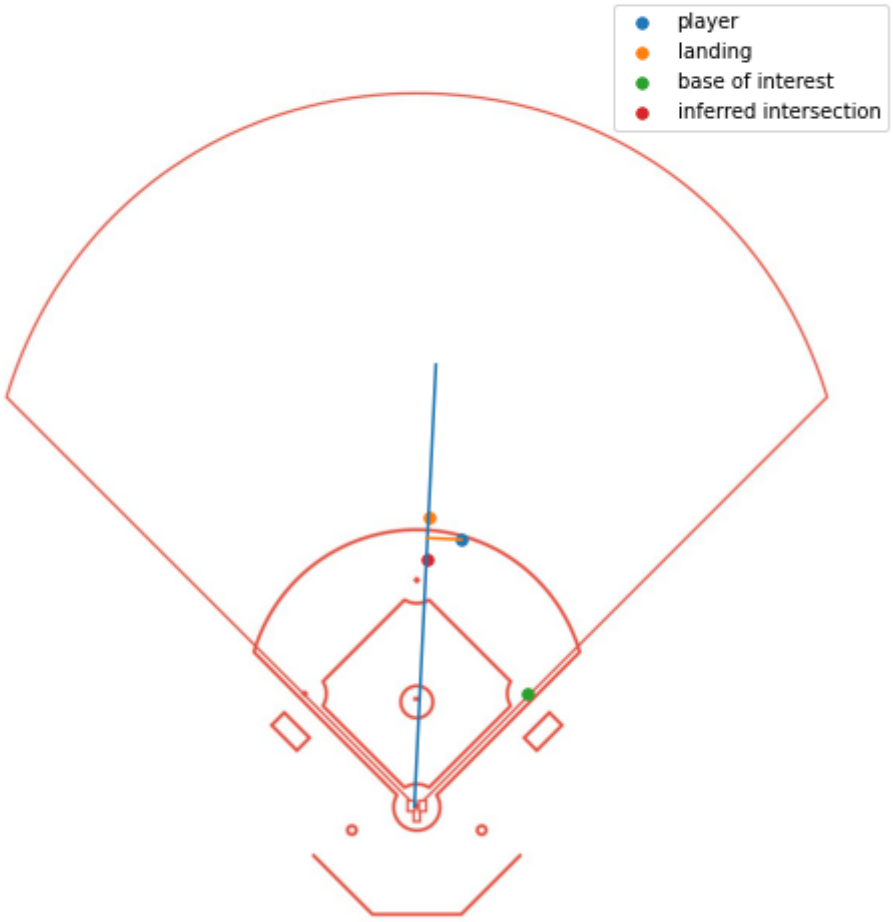
```
print('5 Biggest Where GLMM > SVM')
for i in range(5):
    plot_single_sample(df.reset_index().sort_values('prob_differences').iloc[[i]
```

```
5 Biggest Where GLMM > SVM
/Users/dangoldberg/miniconda3/envs/tbj2021/lib/python3.7/site-packages/numpy/cor
e/_asarray.py:136: VisibleDeprecationWarning: Creating an ndarray from ragged ne
sted sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with dif
ferent lengths or shapes) is deprecated. If you meant to do this, you must speci
fy 'dtype=object' when creating the ndarray
  return array(a, dtype, copy=False, order=order, subok=True)
/Users/dangoldberg/Desktop/code/interviews/tbj/tbj_202101/src/utils/viz_utils.p
y:84: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequence
s (which is a list-or-tuple of lists-or-tuples-or ndarrays with different length
s or shapes) is deprecated. If you meant to do this, you must specify 'dtype=obj
ect' when creating the ndarray
  coords = np.array(self.coords)
/Users/dangoldberg/Desktop/code/interviews/tbj/tbj_202101/src/utils/viz_utils.p
y:93: UserWarning: Matplotlib is currently using module://ipykernel.pylab.backen
d_inline, which is a non-GUI backend, so cannot show the figure.
  fig.show()
/Users/dangoldberg/Desktop/code/interviews/tbj/tbj_202101/src/utils/geometry.py:
81: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify 'dtype=objec
t' when creating the ndarray
  return np.array([min_time_x, min_time_y])
```

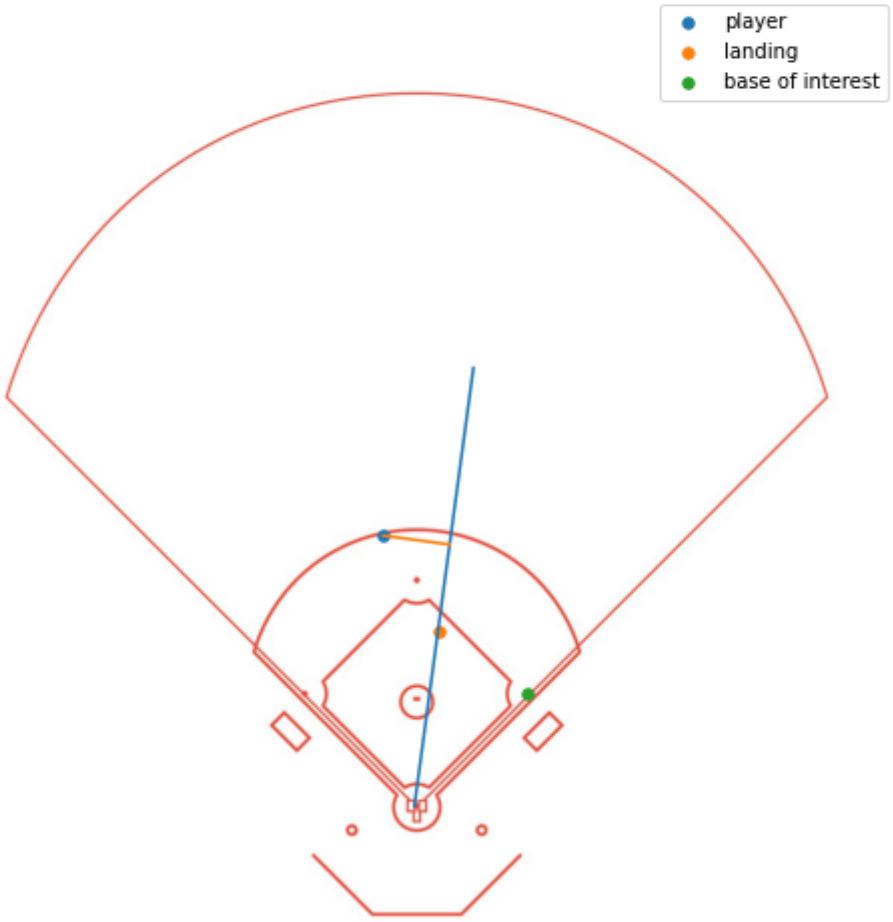
4085578
hit_into_play_no_out | single | 4 | f_fielded_ball
launch_speed: 92.3 | launch_vert_ang: -17.2
base_of_interest: 1 | angle: 94.5
player_time: 1.51 | ball_time: 1.45



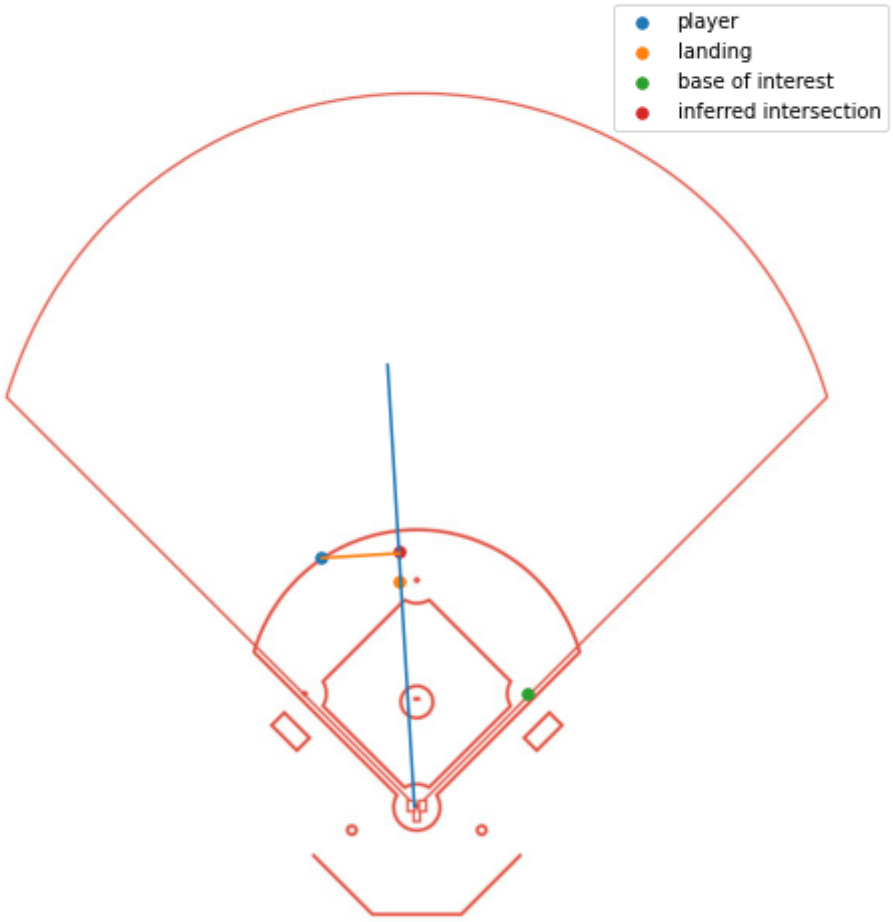
4008352
hit_into_play | field_out | 6-3 | f_assist
launch_speed: 93.1 | launch_vert_ang: 1.2
base_of_interest: 1 | angle: 83.5
player_time: 1.08 | ball_time: 1.08

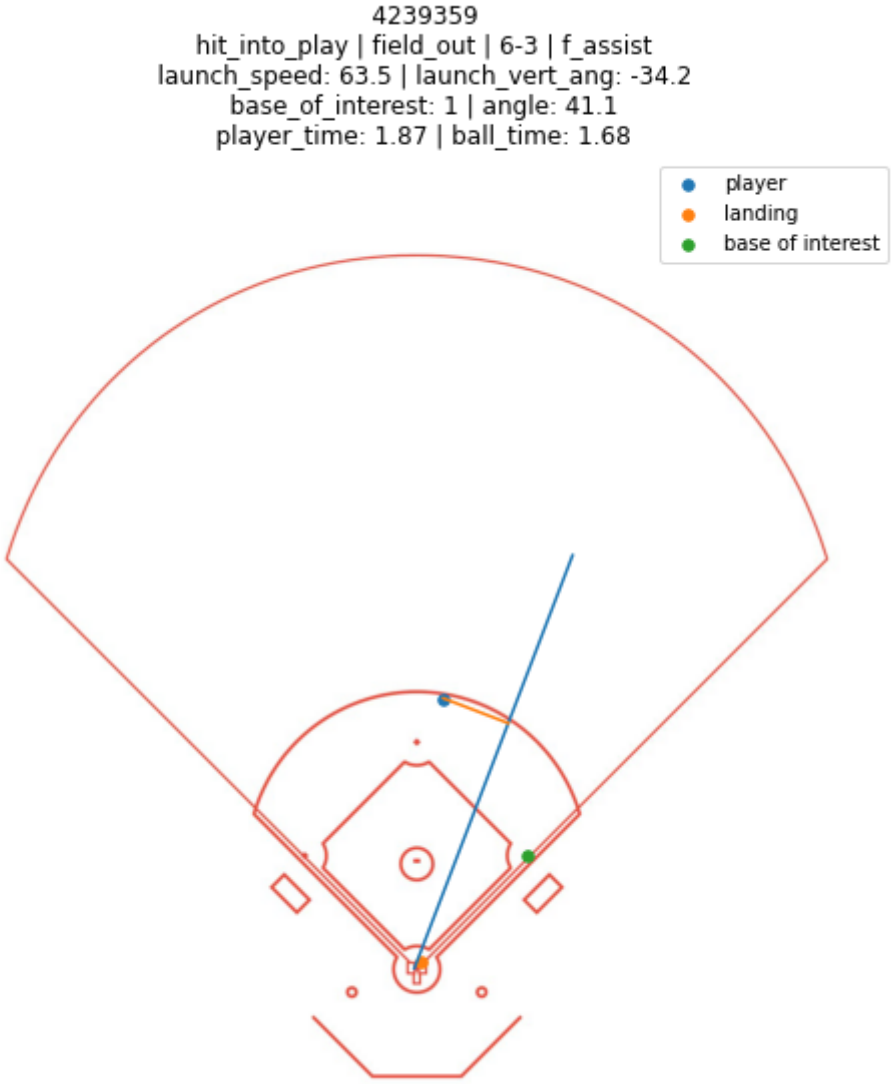


4078975
hit_into_play_no_out | single | 6 | f_fielded_ball
launch_speed: 64.0 | launch_vert_ang: 8.7
base_of_interest: 1 | angle: 40.1
player_time: 1.80 | ball_time: 1.68



4197503
hit_into_play | field_out | 6-3 | f_assist
launch_speed: 49.8 | launch_vert_ang: 24.6
base_of_interest: 1 | angle: 39.2
player_time: 2.09 | ball_time: 2.09



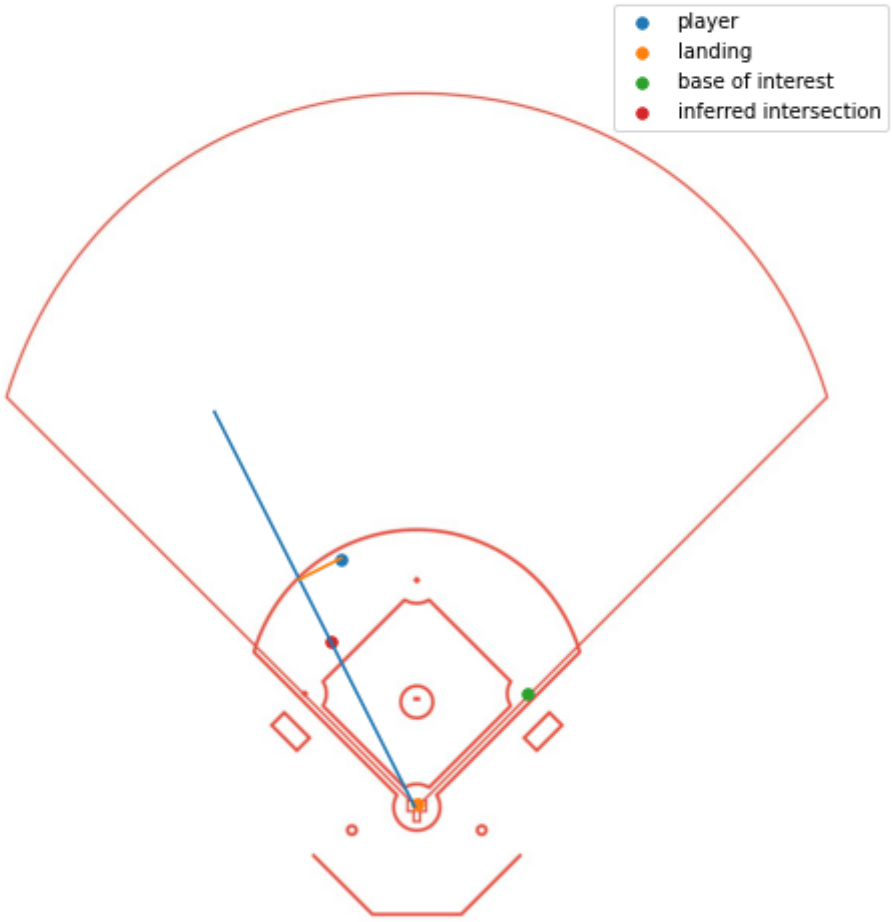


In [85]:

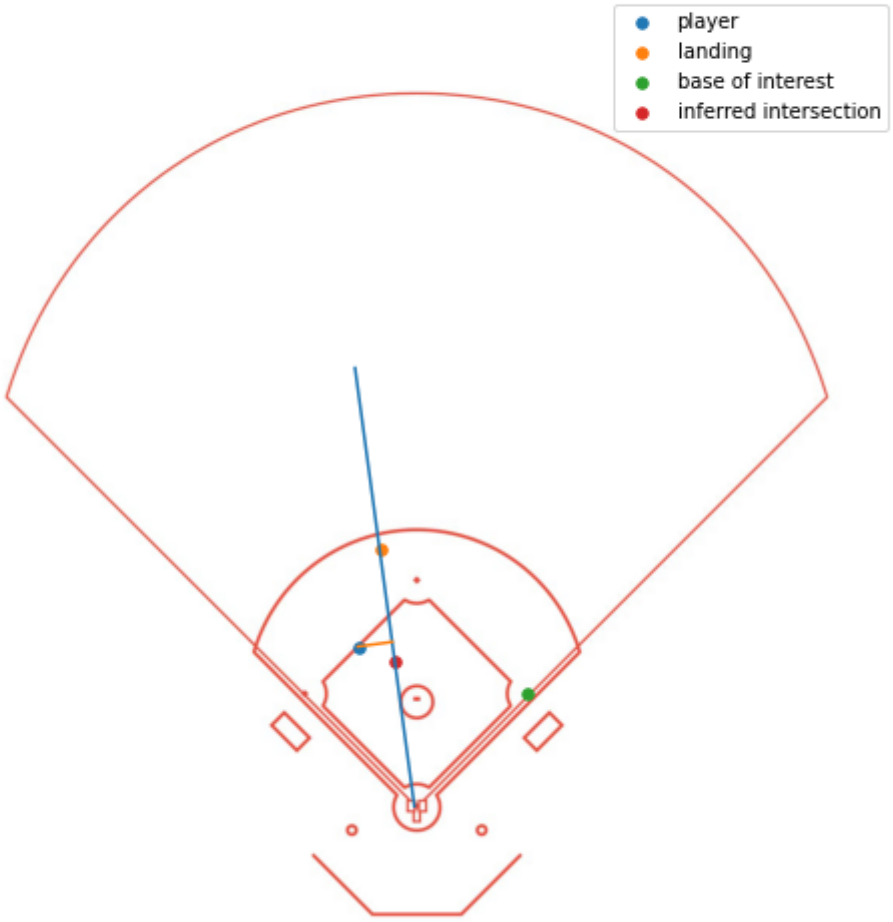
```
print('5 Biggest Where SVM > GLMM')
for i in range(5):
    plot_single_sample(df.reset_index().sort_values('prob_differences', ascendin
```

```
5 Biggest Where SVM > GLMM
```

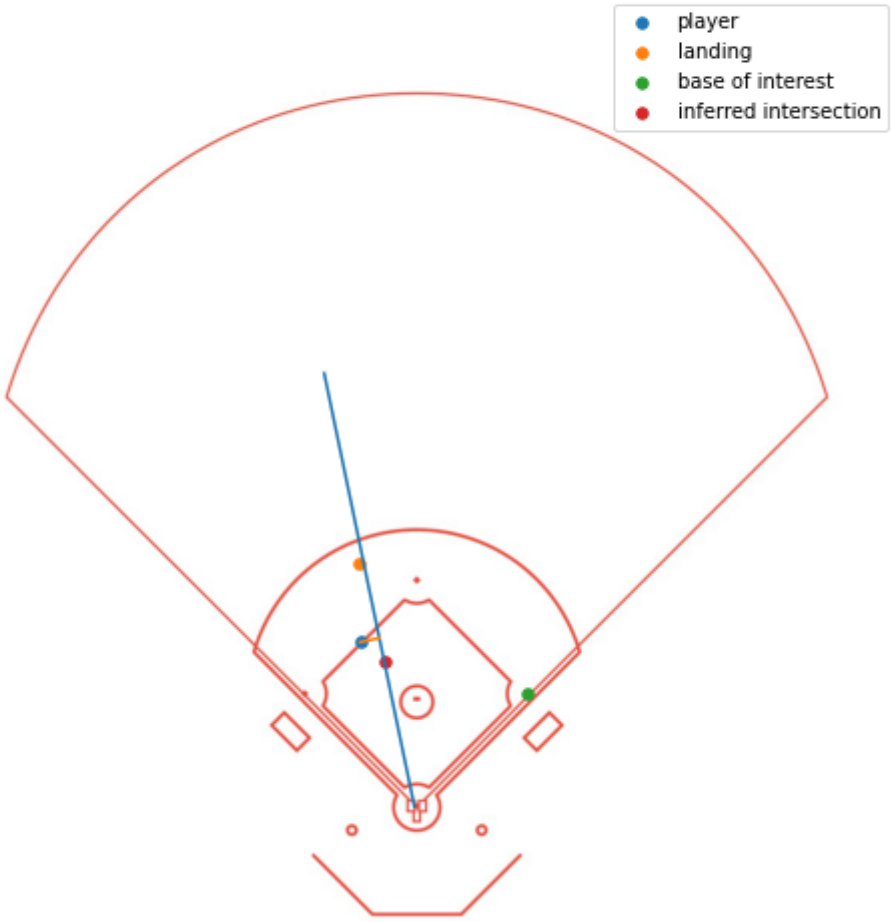
4085305
hit_into_play_score | single | 5 | f_fielded_ball
launch_speed: 33.9 | launch_vert_ang: 36.8
base_of_interest: 1 | angle: 61.2
player_time: 2.23 | ball_time: 2.23



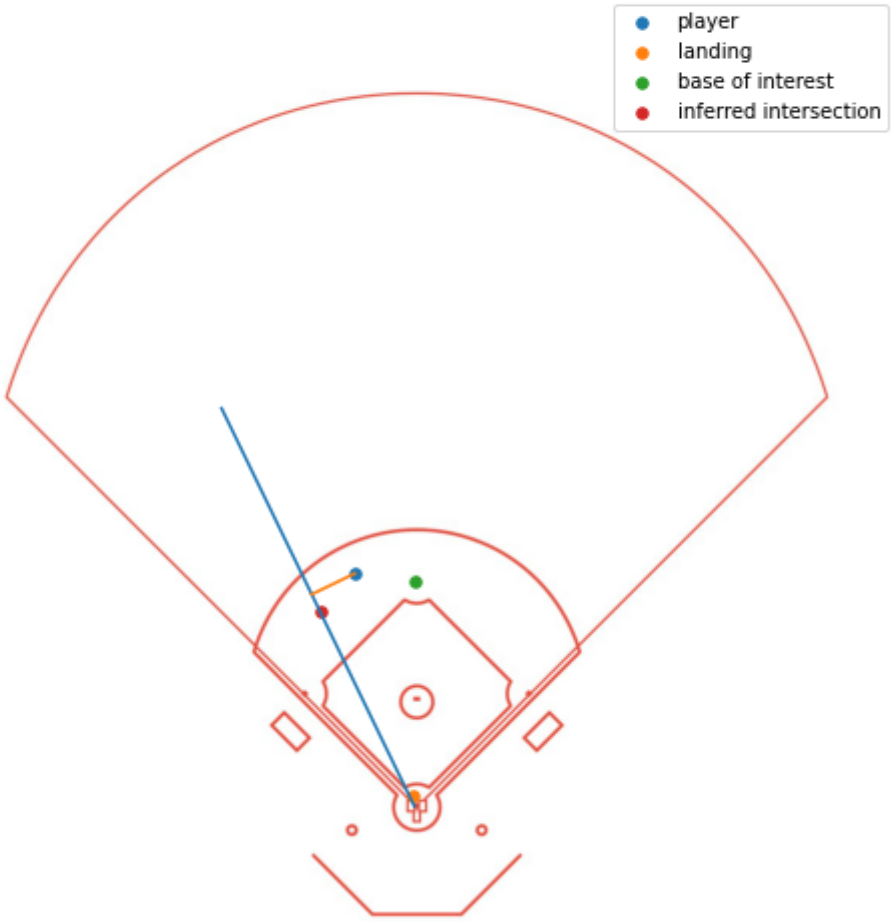
4197712
hit_into_play_score | single | 8 | f_fielded_ball
launch_speed: 56.2 | launch_vert_ang: 23.8
base_of_interest: 1 | angle: 5.5
player_time: 1.06 | ball_time: 1.06

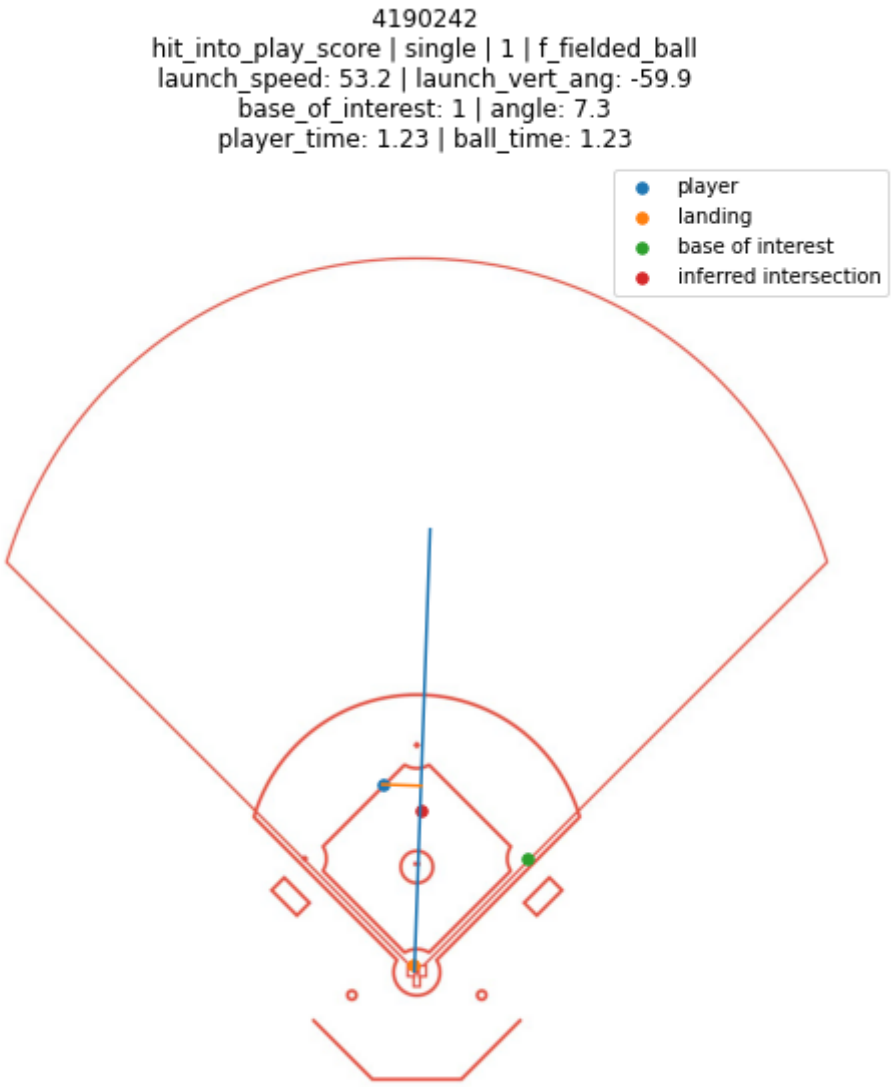


4190874
hit_into_play_score | single | 7 | f_fielded_ball
launch_speed: 73.0 | launch_vert_ang: 13.1
base_of_interest: 1 | angle: 21.4
player_time: 0.82 | ball_time: 0.82



4086942
hit_into_play_no_out | single | 6 | f_fielded_ball
launch_speed: 63.8 | launch_vert_ang: 8.0
base_of_interest: 2 | angle: 125.6
player_time: 1.38 | ball_time: 1.38





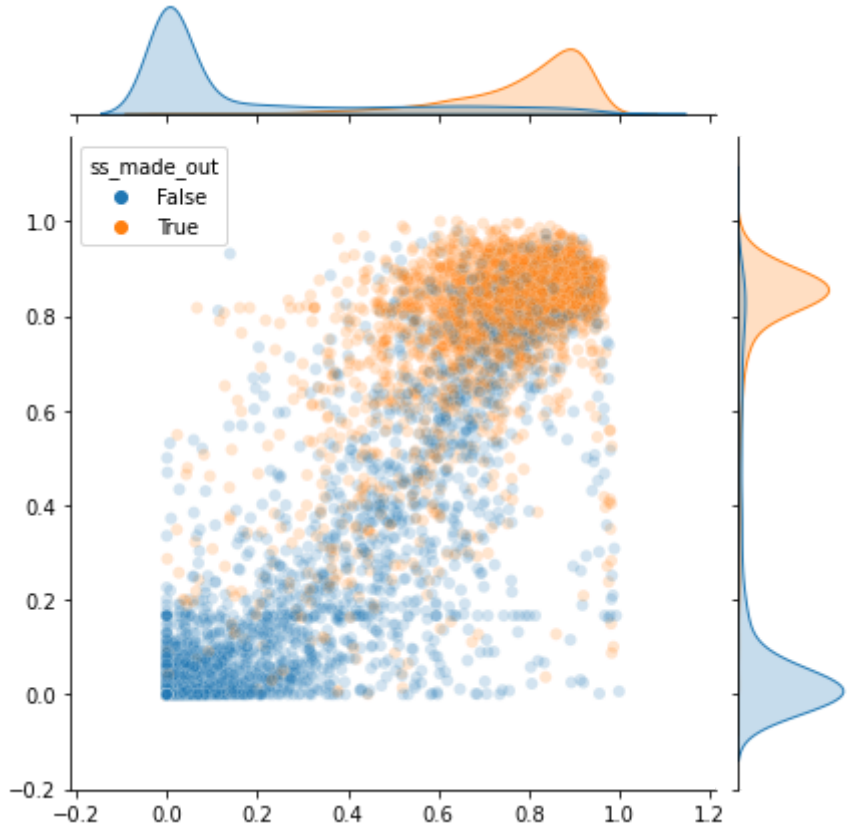
I honestly can't see any pattern from just inspecting the biggest dissagreements.

Scatterplot of predictions from both models

In [105…

```
sns.jointplot(x=glmm_preds_proba, y=svm_preds_proba[:, 1], hue=df['ss_made_out']
plt.xlabel('Multilevel Logistic Regression')
plt.ylabel('SVM-Poly')
plt.plot()
```

Out[105… []



The first thing I notice is that at around 17% for the SVM (y-axis) there is a straight line across the x-axis, which would indicate the same pathology of a disproportionate amount of density being placed on the same exact prediction as the SVM-RBF. Looking back up at the histogram of predictions for that model, I can now see that was something I missed above, as it's smaller than the SVM-RBF, but still there. There's another similar spike in density on the converse side of the prediction space, at around 83%, also noticable on the histogram of predictions show in the previous section. For that reason, plus the fact that the fully Bayesian Multilevel Logistic Regression model give me uncertainty estimates, I'll go with the Logistic Regression model.

In [ ]:

4 - Generate OAA Leaderboard

Now that we've chosen our model we can use the predicted probabilities to calculate the OAA metric for each player. The Questionnaire also asks to count the number of "opportunities" so I will interpret that to mean the number of reasonable chances for the SS to make the play, and I will output that as well.

To determine "opportuntities" I'll pick a cut-off in predicted probability that is very very low, so that in theory making the play was extremely unlikely but not impossible. For this I'll choose 0.5%, so that the play is made in 1/200 opportunities.

First I'll use the fully bayesian output instead of just the mean parameter estimates

In [161…

```
params = model.params  # return a dictionary of arrays
bias = params['bias']
slope1 = params['slope1']
slope2 = params['slope2']
slope3 = params['slope3']
slope4 = params['slope4']
slope5 = params['slope5']
slope6 = params['slope6']
slope7 = params['slope7']
slope8 = params['slope8']
slope9 = params['slope9']
slope10 = params['slope10']
level_param = params['shortstop_effect']

# get predictions
preds_proba = expit(
    bias.reshape(-1,1) \
    + slope1.reshape(-1,1)*X_t[:,0].reshape(1,-1) \
    + slope2.reshape(-1,1)*X_t[:,1].reshape(1,-1) \
    + slope3.reshape(-1,1)*X_t[:,2].reshape(1,-1) \
    + slope4.reshape(-1,1)*X_t[:,3].reshape(1,-1) \
    + slope5.reshape(-1,1)*X_t[:,4].reshape(1,-1) \
    + slope6.reshape(-1,1)*X_t[:,5].reshape(1,-1) \
    + slope7.reshape(-1,1)*X_t[:,6].reshape(1,-1) \
    + slope8.reshape(-1,1)*X_t[:,7].reshape(1,-1) \
    + slope9.reshape(-1,1)*X_t[:,8].reshape(1,-1) \
    + slope10.reshape(-1,1)*X_t[:,9].reshape(1,-1) \
    + np.stack([level_param[:, l-1] for l in levels]).T
)
```

Then i'll combine that data in an xarray Dataset

I have to use an xarray Dataset so that I can pass in the (samples, observations) 2D array as data for ss_out_probability, OAA, and opportunities columns so I can first groupby playerid and aggregate across observations, and only at the end take the mean and std across samples.

In [207…

```
import xarray as xr

def summarize_outs_above_average(df, y, preds_proba, min_prob_for_opportunity =
    """
    This function prepares a dataframe that summarizes the OAA metric, given a m
    """

    model_output = xr.Dataset({
        'ss_out_probability':(['samples','observations'], preds_proba),
        'OAA': (['samples','observations'], y.values-preds_proba),
        'opportunities': (['samples', 'observations'], (preds_proba > min_prob_f
        'observed_out': ('observations', y),
        'playerid': ('observations', df['playerid'].values)
    })

    return model_output
```

In [221…

```
oaa = summarize_outs_above_average(df, y, preds_proba)
oaa
```

Out[221… xarray.Dataset

▶ Dimensions: (observations: 9362, samples: 2000)

▶ Coordinates: (0)

▼ Data variables:

ss_out_probabili…	(samples, observations)	float64	0.9111 0.7825 … 0.09254 0.…
OAA	(samples, observations)	float64	0.08889 -0.7825 … -0.0925…
opportunities	(samples, observations)	int64	1 1 1 1 1 1 0 1 … 1 1 0 1 1 1 1 1
observed_out	(observations)	int64	1 0 1 1 1 0 0 1 … 0 1 0 1 0 1 0 0
playerid	(observations)	int64	11742 9425 5419 … 161551 …

▶ Attributes: (0)

In [239…

```
# aggregate across observations to get player-level stats
player_summary = oaa.groupby('playerid')\
                    .sum(dim='observations')

# aggregate across samples to get bayesian flavour of metrics
player_summary['OAA_mean'] = player_summary.OAA.mean(dim='samples')
player_summary['OAA_std'] = player_summary.OAA.std(dim='samples')
player_summary['opportunities'] = player_summary.opportunities.mean(dim='samples

# convert to pandas dataframe
player_summary = player_summary[['opportunities','OAA_mean','OAA_std']].to_dataf
player_summary['OAA_per_Opp'] = player_summary['OAA_mean'] / player_summary['opp

player_summary
```

Out[239…

playerid	opportunities	OAA_mean	OAA_std	OAA_per_Opp
162066	227.94	11.09	4.83	0.05
162648	196.57	9.15	4.04	0.05
197513	90.34	4.46	1.91	0.05
154448	225.88	4.34	3.78	0.02
9742	97.64	4.22	1.80	0.04
…	…	…	…	…
160570	182.68	-4.08	3.27	-0.02
164881	117.42	-4.11	2.32	-0.03
6619	57.08	-4.80	1.40	-0.08
171806	145.64	-5.18	3.16	-0.04
171885	96.81	-10.07	2.72	-0.10

107 rows × 4 columns

In [240…
```
# prep leaderboard with rank
player_summary = player_summary.reset_index().reset_index().rename(columns={'ind
player_summary.loc[:, 'rank'] = player_summary['rank'] + 1
player_summary = player_summary.set_index('rank')
```

In [241…
```
format_1d = "{0:.1f}".format
format_2d = "{0:.2f}".format
format_3d =  "{0:.3f}".format

player_summary[['opportunities']] = player_summary[['opportunities']].applymap(f
player_summary[['OAA_mean','OAA_std']] = player_summary[['OAA_mean','OAA_std']].
player_summary[['OAA_per_Opp']] = player_summary[['OAA_per_Opp']].applymap(forma

player_summary[:20] # top 20
```

Out[241…

rank	playerid	opportunities	OAA_mean	OAA_std	OAA_per_Opp
1	162066	227.9	11.09	4.83	0.049
2	162648	196.6	9.15	4.04	0.047
3	197513	90.3	4.46	1.91	0.049
4	154448	225.9	4.34	3.78	0.019
5	9742	97.6	4.22	1.80	0.043
6	2950	151.2	4.11	2.73	0.027
7	168314	178.1	4.07	3.10	0.023
8	5495	226.8	3.98	3.78	0.018
9	9148	155.7	2.88	2.65	0.018
10	9074	181.0	2.81	3.19	0.016
11	162294	50.0	2.67	1.13	0.053
12	132551	200.5	2.65	3.50	0.013
13	161551	231.2	2.36	3.51	0.010
14	171164	13.0	2.02	0.24	0.155
15	5419	163.9	2.02	2.74	0.012
16	7580	121.8	1.64	1.97	0.013
17	167746	26.6	1.48	0.56	0.056
18	11742	214.5	1.22	3.15	0.006
19	184486	9.0	1.17	0.25	0.129
20	121615	31.8	1.13	0.60	0.036

In [282…
```
player_summary.to_csv('../data/ss_OAA.csv')
```

In [245…
```
# for markdown doc
```

```
print(player_summary[:20].to_markdown())
```

rank	playerid	opportunities	OAA_mean	OAA_std	OAA_per_Opp
1	162066	227.9	11.09	4.83	0.049
2	162648	196.6	9.15	4.04	0.047
3	197513	90.3	4.46	1.91	0.049
4	154448	225.9	4.34	3.78	0.019
5	9742	97.6	4.22	1.8	0.043
6	2950	151.2	4.11	2.73	0.027
7	168314	178.1	4.07	3.1	0.023
8	5495	226.8	3.98	3.78	0.018
9	9148	155.7	2.88	2.65	0.018
10	9074	181	2.81	3.19	0.016
11	162294	50	2.67	1.13	0.053
12	132551	200.5	2.65	3.5	0.013
13	161551	231.2	2.36	3.51	0.01
14	171164	13	2.02	0.24	0.155
15	5419	163.9	2.02	2.74	0.012
16	7580	121.8	1.64	1.97	0.013
17	167746	26.6	1.48	0.56	0.056
18	11742	214.5	1.22	3.15	0.006
19	184486	9	1.17	0.25	0.129
20	121615	31.8	1.13	0.6	0.036

In [ ]:

For kicks let's compare to the player-level variable intercept in the heirarchical model

In [277…

```
ss_effect_intercept_mean = level_param.mean(axis=0)
ss_effect_intercept_std = level_param.std(axis=0)
```

In [278…

```
encoded_levels = np.array(range(len(ss_effect_intercept_mean)))
ss_effect_playerids = oe.inverse_transform(encoded_levels.reshape(-1,1)).reshape
```

In [283…

```
glmm_results = pd.DataFrame({
    'playerid':ss_effect_playerids,
    'ss_effect_intercept_mean': ss_effect_intercept_mean,
    'ss_effect_intercept_std': ss_effect_intercept_std
}).sort_values('ss_effect_intercept_mean', ascending=False)

glmm_results[:20]
```

Out[283…

	playerid	ss_effect_intercept_mean	ss_effect_intercept_std
56	162066	0.21	0.15
59	162648	0.18	0.15
87	197513	0.09	0.14
32	9742	0.09	0.13
4	2950	0.09	0.13
46	154448	0.08	0.12
14	5495	0.08	0.12
70	168314	0.08	0.12
57	162294	0.06	0.15
26	9148	0.06	0.12
25	9074	0.05	0.12
44	132551	0.05	0.12
75	171164	0.05	0.15
55	161551	0.05	0.11
13	5419	0.04	0.12
21	7580	0.04	0.12
66	167746	0.04	0.15
82	184486	0.03	0.16
43	121615	0.03	0.14
11	4955	0.03	0.15

Close, but they order is slightly different! My guess is the difference is that the variable intercept captures the 'skill' of the player to add incremental probabilty of making an out independant of the kinds of opportunities they actually saw. In contrast, the observed difference of making an out vs the probability of making an out (OAA metric) is sensitive to the opportunities a player actually gets, in addition to their skill.

In [284…

```
glmm_results.to_csv('../data/ss_defense_glmm.csv', index=False)
```

Thanks! I enjoyed this :)

In [6]: