Springboard - DSCT
Capstone Project 2 - Final Report

David Buehler

February, 2020

Introduction

Baseball is like a chess match between the batter and pitcher. What pitch will the pitcher throw next? Where in the zone will he throw it? Is the batter able to make an educated guess on what is coming next, or is he totally clueless and just reacting to what is coming? Sports are beginning to adopt data science and analytics to answer questions like these, and this is what I want to do for my 2nd capstone project. I want to build models that can be used to make predictions of pitches coming from certain pitchers based on factors such as count, score of the game, how many runners on base, if a righty or lefty is batting, and many others that could go into making predictions like this. Using these models, I can analyze what factors were the most important in determining what pitches were thrown. I'm also able to choose specific pitchers, so comparing Cy Young award winners with pitchers that belong to the bottom-of-the-rotation could be interesting to look at. I would guess the Cy Young winners will be less predictable than pitchers who are not at this level.

The client for this project would be the managers, pitching, and hitting coaches who are interested in these questions for decision-making, as well as the players themselves. Any leg up they can get on the opposition is a good thing and if there is any way to reliably predict pitches coming, that would be an advantage for the hitters. Hitting coaches would be interested in

these models as well. They can coach their guys to jump on a fastball on a certain count or sit back on an off-speed pitch in certain situations. Models like the ones I built in this project could be even be good for baserunners, once the hitters get on base. If they and the coaches had an idea of what pitches were coming, baserunners could steal on off-speed stuff as it will give them a better chance to swipe the base.

For this project I decided to use classification models as the target variable represents the four types of pitches: fastball, breaking ball, off-speed, andr other. I ended up testing six different types of machine learning models: logistic regression, ridge classification, random forest classification, gradient-boosted trees for classification, k-nearest neighbors, and adaptively-boosted classification (Ada-Boost). Gradient boosting performed the best out of these models, and was the model I went on a deeper analysis with.

This was an imbalanced classification problem, with fastballs being the majority class, breaking balls having around half as many fastballs, then off-speed pitches having about a third of the fastball numbers. This meant resampling had to be performed, and I used a combination of under-sampling and oversampling (specifically, I used the SMOTE oversampling technique.) Combining oversampling with gradient boosting netted some interesting results. These models consistently performed well on the larger classes, and dropped off in performance on smaller classes. For gradient boosting without resampling, the majority class (fastballs) had a recall of 0.95, precision of 0.57 and an f1-score of 0.71. The next largest class, breaking balls, only had a recall of 0.09 and precision of 0.47 with an f1-score of 0.15. The next largest class, off-speed pitches, had a precision of 0.71, but a recall of 0. The model wasn't able to correctly predict any off-speed pitches, probably because it was such a low minority. It had about 4.5x less samples

than the fastball class. This was a pattern consistently seen throughout the predictive modeling analysis, even with resampling; majority classes would get predicted well, minority classes would not be.

After looking more deeply into gradient boosting, I got into using neural network classifiers, one through scikit-learn and a deeper neural network through Keras and TensorFlow. These models came up with a similar score as the gradient boosting did without resampling.

Approach to the Problem

Data acquisition and wrangling

I pulled the data from Kaggle, and it all came clean already. There was very little cleaning I had to do to get it in a good shape for analysis and visualization. What I did do though was a couple things, first I noticed there were a few pitches that occured with 4 balls in the count, and since you can have no more than 3 balls in the count before a ball is thrown, those needed to be dropped. Next I wanted to know what pitch in the at-bat each and every pitch was, so I looped through every pitch and was able to do just that. I added that as a column to the data frame. From there, I needed to group pitches into 4 groups: fastballs, breaking balls, offspeed pitches, and other pitches. Fortunately Pandas has an easy function to do that, and was able to accomplish that with relative ease. Finally, I wanted to know what pitcher was throwing each pitch, and the Kaggle site I got my data from had another CSV file with each

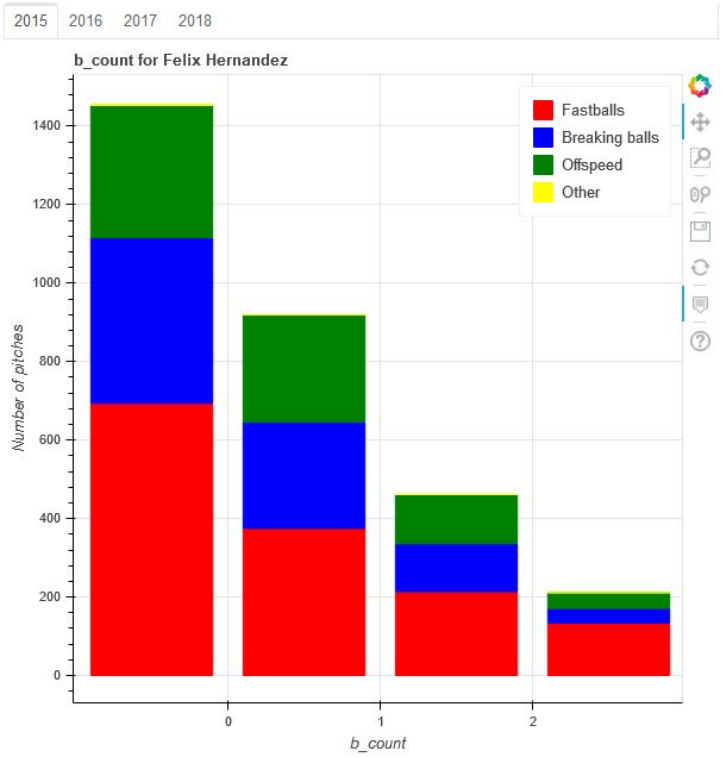
player ID and their corresponding first and last name. With a simple merge call, I made two new columns with a pitcher's first and last name in separate columns and my data wrangling was done after putting the wrangled data frames into CSV files to use next.

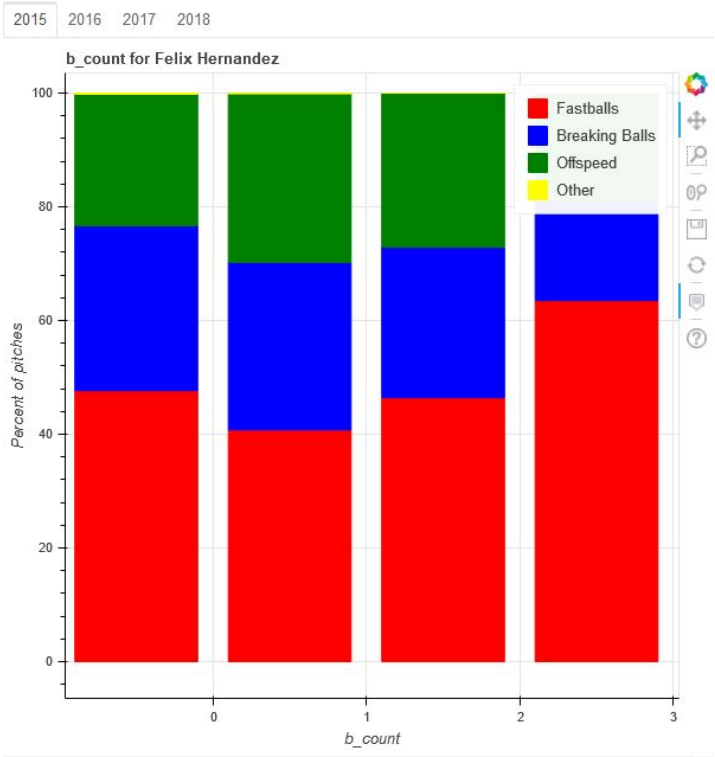
Data Storytelling

For the storytelling portion of my project, I was able to get deep into the data. I wanted to use the Bokeh library because I knew that could be a powerful tool in data visualization, and I was able to do so after coming to grips on what data I wanted to show. First I needed to create a couple functions that would pull out the needed data. I was able to pull out how many fastballs, breaking balls, offspeed and other pitches that pitchers threw based on the variables in the data frames I'm working with. For example, I used a function that would return a data frame that tells me how many, and the percentage of fastballs/breaking balls/offspeed/other pitches a certain pitcher threw with so many balls in the count, broken up by year. An example of this would be this data frame for Felix Hernandez:

	count	year	number_of_FB	number_of_BB	number_of_OS	number_of_OT	percent_FB	percent_BB	percent_OS	percent_OT
0	0	2015	694	422	337	3	47.664835	28.983516	23.145604	0.206044
1	1	2015	375	271	273	1	40.760870	29.456522	29.673913	0.108696
2	2	2015	214	122	125	0	46.420824	26.464208	27.114967	0.000000
3	3	2015	134	37	40	0	63.507109	17.535545	18.957346	0.000000
4	0	2016	592	295	211	0	53.916211	26.867031	19.216758	0.000000
5	1	2016	313	201	211	0	43.172414	27.724138	29.103448	0.000000
6	2	2016	195	100	138	0	45.034642	23.094688	31.870670	0.000000
7	3	2016	124	33	40	0	62.944162	16.751269	20.304569	0.000000
8	0	2017	298	215	140	0	45.635528	32.924962	21.439510	0.000000
9	1	2017	165	115	121	0	41.147132	28.678304	30.174564	0.000000
10	2	2017	86	59	71	0	39.814815	27.314815	32.870370	0.000000
11	3	2017	62	23	26	0	55.855856	20.720721	23.423423	0.000000
12	0	2018	503	411	264	2	42.627119	34.830508	22.372881	0.169492
13	1	2018	272	232	213	1	37.883008	32.311978	29.665738	0.139276
14	2	2018	185	132	113	0	43.023256	30.697674	26.279070	0.000000
15	3	2018	138	67	28	0	59.227468	28.755365	12.017167	0.000000

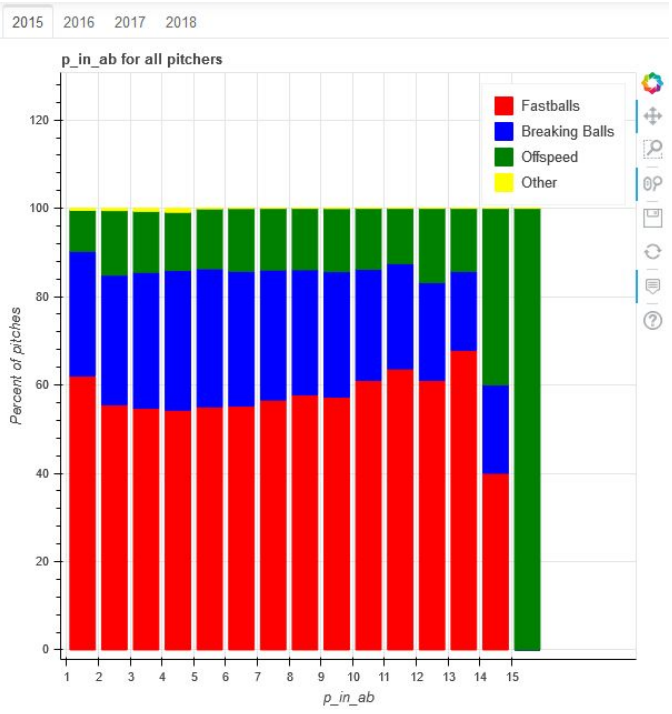
This data frame would then get used to make a plot of this data using Bokeh, and would turn out like this:





The top graph being the total number of types of pitches thrown with each number of balls in the count, and the bottom being the percentage of types of pitches thrown. The percentage graph usually gives more information, as trends are able to be seen better than looking at the total number of pitches.

Those graphs were made using the function that took in fixed number variables (ball count, strike count, batter side etc.) I was also able to put together a function that took in variable number features and plot them for me, like innings and what the pitch in the at-bat is. Unfortunately I wasn't able to look at this for specific pitchers, so it's looking at every pitcher in the data frames. The percentage pitch in at-bat graph looks like the graph below:



The graph shown is only for the 2015 season, but some trends were noticed as the years went on. First pitch fastball occurs 62% of the time in 2015, and actually decreases over time.

Down to 60% in 2018. It's not a huge variation but certainly notable. Breaking balls have seem to become more prevalent as first pitches to batters as time has gone on, rising from 28% in 2015 to 32% in 2018. Pitchers seem to agree that the best time for an off-speed pitch, most commonly a changeup, is the 2nd pitch of an at bat, and has stayed at around 14% of all types of pitches. These were just a few things that were noticed in an otherwise information filled graph.

Next I wanted to take a look at some Cy Young award winners, and see how their pitches stacked up against the rest of the MLB. Starting in 2015, Jake Arrieta of the Chicago Cubs won the National League Cy Young award, and Dallas Keuchel of the Houston Astros who won the American League Cy Young award. These two pitchers were at the top of their game in 2015, and the data shows why. Finally, I thought it would be interesting to look at the pitches thrown with the number of runners on base, so I was able to make a column in the data frame that had the total number of runners on base, but not where they were, and made my Bokeh plots on those.

Inferential Statistics

This data did not come with many continuous variables, mainly batter score and pitcher score, almost all the rest of them are categorical. Looking at batter score and pitcher score though, I thought it would be interesting to see how significant the score differential was on the type of pitch thrown. Creating the score differential column was easy enough, then I created a function that makes the pitch type I want to look at a 1, then all the other pitches a 0 in the pitch_type column of the data frame. Then created a bootstrap test function, that takes the

target pitch type as a variable, and created histograms with it, and performed a p-test from scipy's ttest_ind function. I operated under the null hypothesis of: "On average, pitchers will throw the same amount of fastballs/breaking balls/offspeed pitches no matter the score differential" and performed the bootstrap test at a 95% confidence interval. Testing all four pitch types, all of them came under the 0.05 threshold, so I was able to reject the null hypothesis and conclude that the computed differential is statistically significant. While doing this analysis, something really interesting came up while looking at the "other" pitch type. A lot of other pitches, which include pitch outs that usually lead to intentional walks, were used where the score differential was around 1. Which means pitchers were intentionally walking hitters that could either tie the game, or put the hitting team up with one swing of the bat, and I found that to be really interesting.

Next I wanted to take a look at how different variables affected what pitch was thrown using statsmodels' logit function. This would build a very basic logistic regression model, but would only look at the summary from the fit() method and not make any predictions. First up was ball count and strike count for each type of pitch. For fastballs, as ball count goes up, it's more likely that a pitcher would throw a fastball, and as the strike count went up it was less likely that a fastball would be thrown. Next up was breaking balls, which came with some information that hasn't matched any trends seen up to this point. As both the ball and strike count went up, pitchers were less likely to throw a breaking ball. However the graphs and the data have consistently shown that as strike count goes up, breaking balls are more likely to be thrown. An interesting discrepancy there for sure. Next up came runners on, as that was only looked at briefly in the data storytelling. Statsmodels confirmed what the graph showed. As

more runners are on, fastball usage is likely to go up, and breaking balls and offspeed usage are likely to go down.

Baseline Modeling

In this project I am dealing with a multi-class classification problem, and I decided on using scikit-learn's default Logistic Regression (default uses Ridge regularization with the parameter equals to 1.0), and also combined with optimized Ridge regularization as the baseline models.

I decided to use these models for a few reasons, first and foremost this being a classification problem so I needed classification models. Logistic regression being the "base" classification model, then I took a look through scikit-learn's library of models and found the ridge classifier. This model seemed useful for the problem as it gave a different perspective on the data modeling. This model essentially took all the classes, turned them into numeric values then performed a multi-output regression model on the data set. It was useful to perform the modeling with two different classifiers, to see how differently they modeled the data and how accurate they were.

Before actually modeling the data, I needed to split up the data first. However this was not as trivial as just using train_test_split and that being adequate. I needed to keep the class representation of the main data set in the training and testing data for the coming modeling. Fortunately, train_test_split comes with the stratify keyword argument that does just that. I also needed to get all of the noise and features of the data that was not known before the pitch happened. These included all of the "id" columns, the object columns that were just categorical

info, and all other columns that happened after the pitch. This data was also able to be split up by the pitcher too, just by specifying the pitcher's first and last name.

After splitting the data, I needed to do some hyperparameter tuning. For the logistic regression model, the parameter of interest was the C value, and for the ridge classifier, the alpha value needed to be found. Since I was planning on looking at specific pitchers, these parameters needed to be specific for every pitcher. I ran a GridSearchCV to cross-validate and find the best parameter for each model and plugged those parameters into the model that I would eventually be fitting and predicting. Next I wanted to see how accurate these models were. I made another function that took in the best hyperparameters and put those into a logistic regression and ridge classifier model to fit the data and give me an accuracy score of the data. But with classification problems, using accuracy alone to assess the performance of classification models is not enough. I needed to use the classification report derived from the confusion matrix to get a good gauge on how well the model was performing for each class. I was able to build a final function that did just this, that utilized all the splitting data and hyperparameter tuning from before, to fit and predict the data and build confusion matrices and give classification reports.

Extended modeling

After all the baseline modeling was done, I wanted to get into some deeper machine learning models. After doing some research on some of the best models for imbalanced classification problems, I decided on using four different models for an exploratory approach, then choose which one did the best and go more in-depth with that model. The four models I

used were random forest, gradient boosting, ada boosting, and k-nearest neighbors. These gave me a range of methods used to model the data, and were all different enough to give me results on which one would be the best choice. I performed two trials to compare these models against one another, both with only a sample of the full data set. One trial had 10,000 samples and the other had 100,000. Both came out with similar scores for all four models, and showed that gradient boosting was the best model for an in-depth analysis on the project. After determining which was the best model to use, I put that model to work on the entire data set, as I wanted to get classification reports and feature importances for this data, and see how well the gradient boosting model performed overall.

Since I am dealing with an imbalanced classification problem, resampling of the data set would need to be done in the form of over or undersampling the classes. I used a SMOTE oversampling method to better take advantage of a common resampling technique. This allowed me to control different levels of class representation by tuning the minority class to different fractions of the majority class. I started with 75% breaking balls and 50% offspeed pitches, then went to 87.5% breaking balls and 75% offspeed pitches, then finally an even number of all three classes and compared them all against one another.

Neural Networks

Finally I wanted to get into some basic neural networks. I built two different networks, one simple one from scikit-learn (MLPClassifier), then deeper network using Keras and TensorFlow. For both neural networks, I used a 1,000,000 sample from my original data set

because of time considerations. Using all 2.8 million data points would have taken a long time here. For the MLPClassifier, I went with a hidden_layer_size of eight layers consisting of 15 neurons each. I went with 15 because that was the number of features I had in my data set. After performing a fit and predict on the data set with the original class representation, I went into more resampling of the data set, this time with a more tried and true method. Using SMOTE to resample the data set, I performed three different tests with three different levels of class representation. The final test being an even split of all three classes.

The Keras model required some more setup than the scikit-learn model. For the classifier, I added four Dense layers. The model expects rows of data with 15 variables (number of features of the data set), the first hidden layer has 12 nodes and uses a ReLU activation function. The second hidden layer has 15 nodes and also uses a ReLU activation function, then the final output layer has three nodes and uses the softmax activation function. I needed to tell the Keras model how many classes I had, so using the three nodes at the end will allow it to do that. Finally, I used a “categorical_crossentropy” loss function, with an “Adam” optimizer when compiling the model. I used this neural network to model four different class representations; the original representation, as well as the same three resampling tests I did with the MLPClassifier before.

Findings

Model	Precision	Recall	F1-Score	Accuracy	Test Set Size
Logistic Regression	FB: 0.57 BB: 0.42 OS: 0.00	FB: 0.95 BB: 0.08 OS: 0.00	FB: 0.71 BB: 0.14 OS: 0.00	55.61%	713,803
Ridge Classifier	FB: 0.57 BB: 0.42 OS: 0.00	FB: 0.94 BB: 0.09 OS: 0.00	FB: 0.71 BB: 0.14 OS: 0.00	55.56%	713,803
Gradient Boosting	FB: 0.57 BB: 0.47 OS: 0.71	FB: 0.95 BB: 0.09 OS: 0.00	FB: 0.71 BB: 0.15 OS: 0.00	56.14%	713,803
MLPClassifier	FB: 0.57 BB: 0.47 OS: 0.00	FB: 0.95 BB: 0.10 OS: 0.00	FB: 0.71 BB: 0.16 OS: 0.00	56.20%	250,000
Keras NN	FB: 0.56 BB: 0.47 OS: 0.00	FB: 1.00 BB: 0.00 OS: 0.00	FB: 0.72 BB: 0.00 OS: 0.00	56.07%	250,000

Overall, none of the models did very well. They all had fastballs, the majority class, being the better t predicted class. For breaking balls, the next largest class, the models had a tough time predicting. In these tests, there were about half as many breaking balls as fastballs. Then for the off-speed pitches, most of them did not even get predicted. The gradient boosting looks like it tried to predict some off-speed, but did not get any of them correct. Accuracies were not great either. The MLP Classifier seemed to perform the best out of all the models, but it is basically a tie between all five seeing as the spread of the accuracy is not very high.

Logistic Regression and Ridge Classifier

Model	Pitcher	Precision	Recall	F1-Score	Accuracy
Logistic Regression	Felix Hernandez	FB: 0.48 BB: 0.39 OS: 0.39	FB: 0.91 BB: 0.07 OS: 0.13	FB: 0.63 BB: 0.11 OS: 0.19	46.80%
	Justin Verlander	FB: 0.62 BB: 0.52 OS: 0.00	FB: 0.89 BB: 0.22 OS: 0.00	FB: 0.73 BB: 0.31 OS: 0.00	60.26%
	Clayton Kershaw	FB: 0.61 BB: 0.61 OS: 0.00	FB: 0.61 BB: 0.62 OS: 0.00	FB: 0.61 BB: 0.61 OS: 0.00	60.88%
	Edwin Diaz	FB: 0.70 BB: 0.59 OS: 0.00	FB: 0.91 BB: 0.26 OS: 0.00	FB: 0.79 BB: 0.36 OS: 0.00	68.53%
	Aroldis Chapman	FB: 0.77 BB: 0.00 OS: 0.00	FB: 1.00 BB: 0.00 OS: 0.00	FB: 0.87 BB: 0.00 OS: 0.00	76.86%
Ridge Classifier	Felix Hernandez	FB: 0.48 BB: 0.37 OS: 0.37	FB: 0.90 BB: 0.14 OS: 0.06	FB: 0.63 BB: 0.11 OS: 0.20	46.60%
	Justin Verlander	FB: 0.62 BB: 0.52 OS: 0.00	FB: 0.89 BB: 0.22 OS: 0.00	FB: 0.73 BB: 0.31 OS: 0.00	60.16%
	Clayton Kershaw	FB: 0.61 BB: 0.61 OS: 0.00	FB: 0.61 BB: 0.62 OS: 0.00	FB: 0.61 BB: 0.62 OS: 0.00	60.96%
	Edwin Diaz	FB: 0.70 BB: 0.56 OS: 0.00	FB: 0.90 BB: 0.26 OS: 0.00	FB: 0.79 BB: 0.35 OS: 0.00	67.77%
	Aroldis Chapman	FB: 0.77 BB: 0.00 OS: 0.00	FB: 1.00 BB: 0.00 OS: 0.00	FB: 0.87 BB: 0.00 OS: 0.00	76.86%

Taking a look at the baseline models first, they performed at about the same rate as each other for every test I performed. Accuracies of the models were almost the same, about 0.05% off, which is not much at all. Even getting into looking at specific pitchers, the two models performed the same for every pitcher I looked at. From the table above, I saw that these models did not predict any off-speed pitches, since there were so few of them compared to the amount of fastballs in the data set. However, when getting into looking at specific pitchers, the models were able to finally pick out some of the off-speed pitches. For Felix Hernandez, he threw about half as many off-speed pitches as he did fastballs, and about 70% as many breaking balls as fastballs. The models were able to correctly predict some of those breaking balls and off-speed pitches, though not nearly at the rate as fastballs. For logistic regression, the recall on breaking balls and off-speed was 0.07 and 0.13 respectively, where the fastballs were at a recall of 0.91. For the ridge classifier, the numbers were nearly identical. 0.06 and 0.14 for breaking ball and off-speed recall respectively, and 0.90 for fastball recall.

Looking into two other starting pitchers, Justin Verlander and Clayton Kershaw, they had somewhat similar results to Hernandez's. For Verlander, he mainly throws two pitches, fastballs and breaking balls, with off-speed stuff trickled in every now and then. In this data set, he threw about 60% breaking balls as compared to fastballs. Then so few off-speed pitches the model was not able to pick any out, like the overall data set. For both models, the recall on Verlander's fastballs and breaking balls were 0.89 and 0.22 respectively. A small improvement, it seems as though as numbers of types of pitches thrown get closer together, the model is able to better predict the different pitches. That is the pattern I saw for the rest of my machine learning analysis.

Starting with Clayton Kershaw, he threw an about even number of fastballs and breaking balls, and both models were able to predict the pitches at about the same rate. Recall for fastballs was 0.61 and breaking balls was 0.62. Finally, looking at two closers in the MLB, Edwin Diaz and Aroldis Chapman, Diaz was fairly similar to Verlander, throwing about half as many breaking balls as fastballs. He had a breaking ball recall of 0.26 and a fastball recall of 0.91 on both models. For Chapman, he had a large majority on fastballs, throwing more than three times as many fastballs as any other pitch. Both models only predicted fastballs, and did not make any other predictions.

Overall I was starting to see a pattern with the models. As class representation got more similar, models tended to do better. A bigger spread in class representation, like Chapman's fastball count being at least three times larger than any other class, would result in less accurate models as only one class was being predicted, and the others were left out. For pitchers like Kershaw, who threw an even number of fastballs and breaking balls, the models were able to predict both classes at the same rate, and gave the highest accuracy out of the three starting pitchers that were looked at. This trend continued into the more in-depth models as well.

In-depth Modeling

Without Resampling

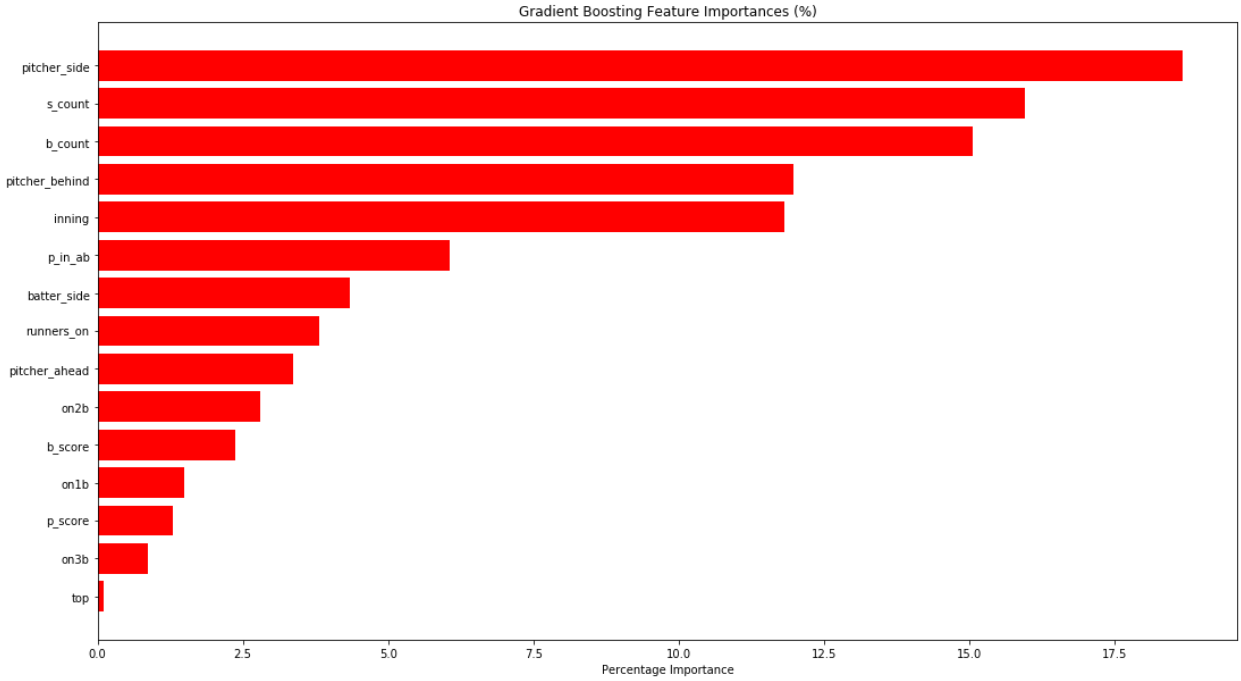
To start the extended modeling process, I wanted to try a variety of different machine learning models to see which performed the best on a sample of the data set, then choose the

best one and go in-depth with it from there. I tested random forest classifier, gradient boosting classifier, ada boosting classifier and k-nearest neighbors classifier. I used a small sample of the data set, 100,000 samples so as to not take too much time, and below are the results.

Model	Precision	Recall	F1-Score	Accuracy
Random Forest	FB: 0.58 BB: 0.37 OS: 0.19	FB: 0.71 BB: 0.30 OS: 0.09	FB: 0.64 BB: 0.33 OS: 0.12	0.50
Gradient Boost	FB: 0.57 BB: 0.44 OS: 0.26	FB: 0.94 BB: 0.10 OS: 0.00	FB: 0.71 BB: 0.16 OS: 0.00	0.56
Ada Boost	FB: 0.57 BB: 0.41 OS: 0.00	FB: 0.95 BB: 0.07 OS: 0.00	FB: 0.71 BB: 0.12 OS: 0.00	0.56
K-Neighbors	FB: 0.58 BB: 0.37 OS: 0.25	FB: 0.81 BB: 0.25 OS: 0.01	FB: 0.67 BB: 0.30 OS: 0.02	0.53

From these results it seems like a tie between gradient boosting and ada boosting. I chose gradient boosting because that performed just a little bit better on the minority classes, and it even tried to predict off-speed pitches where the ada boost model did not.

After choosing gradient boosting, I put it to work on the entire data set. The results that came out of it were very similar to the results in the table above, the only difference being the precision on off-speed pitches went up to 0.71. Everything else stayed essentially the same. The biggest thing I wanted to look at though was feature importances, and below is the graph for them.



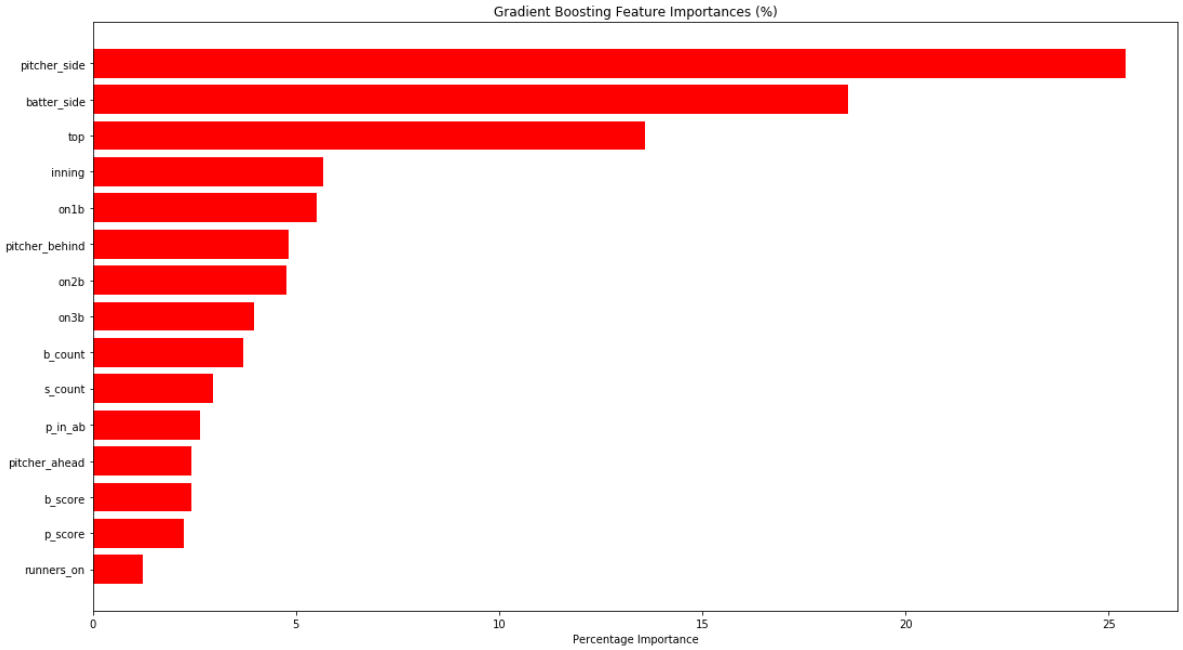
The most important feature according to gradient boosting was what side the pitcher was pitching from. I thought that was interesting as I would have expected what side the batter was on would have made more of a difference on predictions. Having strike count and ball count be the next two most important features makes sense though, as it was found in the exploratory data analysis section that those certainly did make a difference in what pitch was coming next. Overall the order in which the feature importances come in definitely make sense for the context of the game.

With Resampling

Getting into the SMOTE resampling methods on the gradient boosting model, there were some patterns seen here that would continue on into the neural network portion of the modeling. Most commonly that accuracy overall, fastball recall, and fastball f1 score would all drop, but all the other metrics would continually increase. The minority classes

precision/recall/f1 score would all increase as the class representation got closer together. For example: the classification report for the original class representation in the data set had fastball precision/recall/f1 0.57/0.95/0.71. For breaking balls it was 0.47/0.09/0.15, and for off speed pitches it was 0.57/0.00/0.00. With an even split of the data, the accuracy of the model dropped from 56% down to around 49%, but it did a better job at being able to predict minority classes. However precision/recall/f1 scores changed quite a bit. For fastballs, the scores changed to 0.62/0.59/0.60, for breaking balls the scores are 0.41/0.38/0.39, and for offspeed pitches they are 0.22/0.30/0.25. Overall most of the scores increased as the model got fed more information for what situations breaking balls and offspeed pitches were thrown in. However, the model got progressively worse at correctly predicting fastballs, with its recall dropping from 0.95 down to 0.59.

I also took a look at the feature importances for these resampled models, and they all gave a similar graph, seen below.



Pitcher side, along with being the most important feature in the base gradient boosting model, was also the most important feature in the resampled model. However there were some changes at the top of the graph. In the base model with the original class representation, strike count and ball count were the second and third most important feature, but here with the resampled model, those changed to batter side and top of the inning. Top of the inning is a surprise here, as it was the least important feature for the original gradient boosting model, but with the resampled model it is the third most important feature.

Neural Network

MLP Classifier

Jumping straight into the resampling portion of the MLP Classifier, as the non-resampling portion gave the same results as the gradient boosting model, I noticed the same pattern for resampling with this classifier as the gradient boosting classifier. For example: the first resampling test I did had ⅔ as many breaking balls as fastballs, and half as many off-speed pitches as fastballs. The classification report for the predicted test set looks like this:

	precision	recall	f1-score	support
BB	0.44	0.23	0.31	79195
FB	0.59	0.78	0.67	140182
OS	0.24	0.19	0.21	30623
accuracy			0.53	250000
macro avg	0.42	0.40	0.40	250000
weighted avg	0.50	0.53	0.50	250000

Definitely a step up from the original class representation. The overall accuracy fell, and the values for the fastball class fell, but all values for the other classes went up, meaning the neural

network got better at picking out breaking balls and off-speed pitches. This trend continued with a training set class representation of 80% breaking balls and 66.6667% offspeed pitches. Then when doing an even split of all three classes on the training set, the classification report

	precision	recall	f1-score	support
BB	0.41	0.46	0.43	79195
FB	0.67	0.41	0.51	140182
OS	0.20	0.48	0.28	30623
accuracy			0.44	250000
macro avg	0.42	0.45	0.41	250000
weighted avg	0.53	0.44	0.46	250000

for the predicted test set looks like this:

Throughout all three tests, the recall and f1-score of fastballs fell, from 0.78/0.67 on the first test, down to 0.41/0.51 on the third test. Interestingly, the precision of the fastballs actually continued to rise from 0.59 to 0.67. For breaking balls and off-speed pitches, the recall and f1-score continued to climb while the precision slowly fell. Recall/f1 for breaking balls climbed from 0.23/0.31 to 0.46/0.43, and for off-speed pitches they climbed from 0.19/0.21 to 0.48/0.28. However, precision for both classes slowly fell, for breaking balls from 0.44 to 0.41 and for off-speed pitches from 0.24 to 0.20. As resampling went on, this neural network got better in some ways, and worse in others. Overall accuracy came down, and it progressively got worse at predicting the majority class in the testing set, but got much better at predicting minority classes.

Keras and TensorFlow

Finally getting into the results of the last model, the deep learning neural network from Keras and TensorFlow. I saw a similar pattern as the models before, but with some small

changes. Starting with the original data set with the original class representation. Here is the

	precision	recall	f1-score	support
0	0.56	0.99	0.72	140182
1	0.49	0.03	0.05	79195
2	0.00	0.00	0.00	30623
accuracy			0.56	250000
macro avg	0.35	0.34	0.25	250000
weighted avg	0.47	0.56	0.42	250000

classification report for this model:

It predicted fastballs better than the other models did, with a 0.99 recall compared to the 0.95 recalls seen before. However it did worse overall on breaking balls and offspeed pitches, with it not predicting offspeed at all. I used the same SMOTE resampling technique that I did with the previous models, and saw very similar results. With fastball recall/f1 score and overall model accuracy coming down, but everything else increasing. For the even split class representation model, the classification report can be seen in the figure below.

Just like the previous models, this Keras model did better on the minority classes when it was given more information about them.

	precision	recall	f1-score	support
0	0.68	0.37	0.48	140182
1	0.41	0.45	0.43	79195
2	0.19	0.54	0.28	30623
accuracy			0.42	250000
macro avg	0.43	0.45	0.40	250000
weighted avg	0.53	0.42	0.44	250000

Conclusions

After looking at all the models built, and looking at feature importances for the gradient boosting models, it seems like the side the pitcher is pitching from, how many balls in the count there are, how many strikes in the count there are, and what side the batter is hitting from are the most important variables for predicting the next pitch that is coming. Both the original gradient boosting model with the normal class representation, as well as all the resampled versions of the same model had pitcher side on top as the most important feature for predicting pitches. This certainly makes sense in the context of a game. Right handed pitchers are going to throw different pitches in different situations than left handed pitchers. Looking at Clayton Kershaw (a lefty) vs. Felix Hernandez or Justin Verlander (both righties), Kershaw threw many more breaking balls than either Hernandez or Verlander. Left handers and right handers match up differently against batters and will throw pitches based on these matchups. A right handed batter vs. left handed pitcher will match up differently than a left handed batter vs. left handed pitcher, and different pitches will be chosen based on these matchups. How many balls and strikes in the count will also dictate what pitch is thrown. Pitchers with more strikes in the count will be looking to stay ahead of batters and throw more breaking balls or off speed pitches. While pitchers who are behind in the count (more balls) will want to throw more fastballs to catch up. Ultimately all of these factors go into choosing what pitch is coming next.

Future Work

- Delve deeper into the deep learning neural networks and extract feature importances from them. I was unable to get feature importances from the deep learning models, so getting more information in that regard could tell us more about what goes into predicting pitches
- Adapting these models for real time data and adjusting hyperparameters accordingly could be a good way to use these models for use during a game. Using one of these models to predict what pitch is coming next was the point of building them, so putting them to use during a game would serve their intended purpose.