Table of Contents

---

1 - Introduction

In this project, we seek to investigate the efficacy of umpires in calling balls and strikes. Given that there is an easily-defined regulation strike zone, it is trivial to take the measured location data of a pitch and directly determine if that measurement is within the regluation strike zone. Consequently, we take a slightly different focus in `Section 2`, centering two questions:

Question 1: Would a pseudo-robo-umpire (i.e. a model trained on real umpire data) be more or less accurate than an actual umpire?

Question 2: How large of an overall impact does an individual umpire have on training a pseudo-robo-umpire?

Note that in `Section 2`, each model will consider at most three pieces of information for each pitch: the horizontal pitch location, the vertical pitch location, and/or the umpire making the call.

Given the expectation that no human umpire will be perfect at calling balls and strikes, the following is a natural follow-up question:

Question 3: What factors (beyond pitch location) most impact an umpire making an incorrect ball/strike call?

We investigate `Question 3` in `Section 3`, where we will consider many more features than the two (non-`umpire`) features of `Section 2`.

1.1 - A Note on Metrics

In this case, our primary metric of interest will be accuracy, as incorrectly predicting a called ball as a called strike is similar to incorrectly predicting a called strike as a called ball.

Beyond accuracy, we will also track F1-score (the mean of recall and precision) and PR-AUC (area under the precision-recall curve).

1.2 - Data Collection

We will be looking at pitches from the entirety of the 2023 season that were called balls or strikes by an umpire (this does not include swinging strikes, for example). Our source of pitch data is Baseball Savant, via pybaseball. Since Baseball Savant no longer provides umpire information, we used boxscores on Baseball Reference to compile data on home plate umpires for every game, which we then joined to the Baseball Savant pitch data.

We go into further details in Appendix A.

1.3 - Importing Necessary Packages

In [1]:
```
# General
import pandas as pd
import numpy as np

# Early Data Analysis
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_style("whitegrid")

# Images
from IPython.display import Image

# General Modeling
from sklearn.compose import ColumnTransformer
from sklearn.metrics import accuracy_score, precision_recall_curve, auc, f1_score
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Specific Models
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
```

---

2 - Predicting Ball and Strike Calls Using Only Pitch Location and Umpire Information

Recall that in this section, we focus on the following two questions from `Section 1`:

Question 1: Would a pseudo-robo-umpire (i.e. a model trained on real umpire data) be more or less accurate than an actual umpire?

Question 2: How large of an overall impact does an individual umpire have on training a pseudo-robo-umpire?

We begin by reading in the data for this section.

In [2]:
```
core = pd.read_csv('small_model_data.csv')
```

The columns of this data set are:

- `ball/strike` - the umpire's call on the field of if that pitch was a ball or strike ^
- `binary_bs` - a binary version of `ball/strike`
- `true_ball/strike` - an assessment of if the pitch was a ball or called strike based on the regulation strike zone
- `correct_call` - checks if the umpire's call matches the correct call according to the regulation strike zone
- `zone` - the Gameday Zone of the pitch ^
- `hscw` - records if the pitch was in the heart, shadow, chase, or waste region of the strike zone
- `plate_x` - the horizontal location of the pitch (relative to the middle of home plate) as the pitch crossed home plate ^
- `plate_x_mag` - absolute value of `plate_x`
- `plate_x_dir` - sign of `plate_x`
- `plate_z` - the height of the pitch ^
- `sz_top` - the top height of the regulation strike zone (changes per batter) ^
- `sz_bot` - the bottom height of the regulation strike zone (changes per batter) ^
- `umpire` - identity of home plate umpire, as scraped from Baseball Reference

^ as provided by Baseball Savant, see the documentation for more details

2.1 - Early Data Analysis

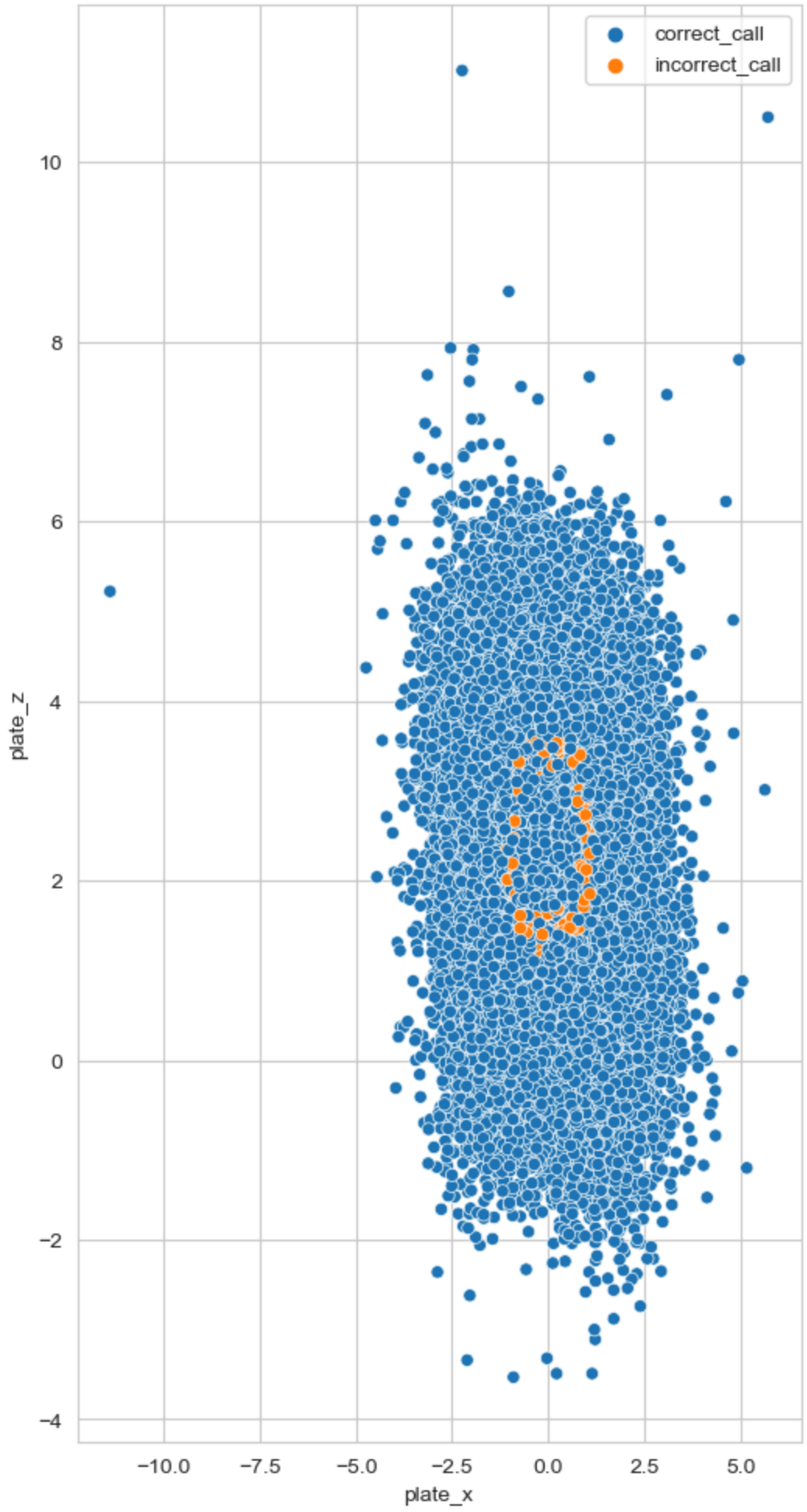
We begin at the most basic level: what percentage of calls are made correctly?

In [3]:
```
core.correct_call.value_counts(normalize = True)
```

Out[3]:
```
correct_call      0.917176
incorrect_call    0.082824
Name: correct_call, dtype: float64
```

We see that approximately 91.7% of ball/strike calls were made correctly. Let's now visualize the pitches in terms of where they crossed the plate, with the blue dots representing pitches that umpires called correctly and orange dots representing pitches that were called incorrectly.

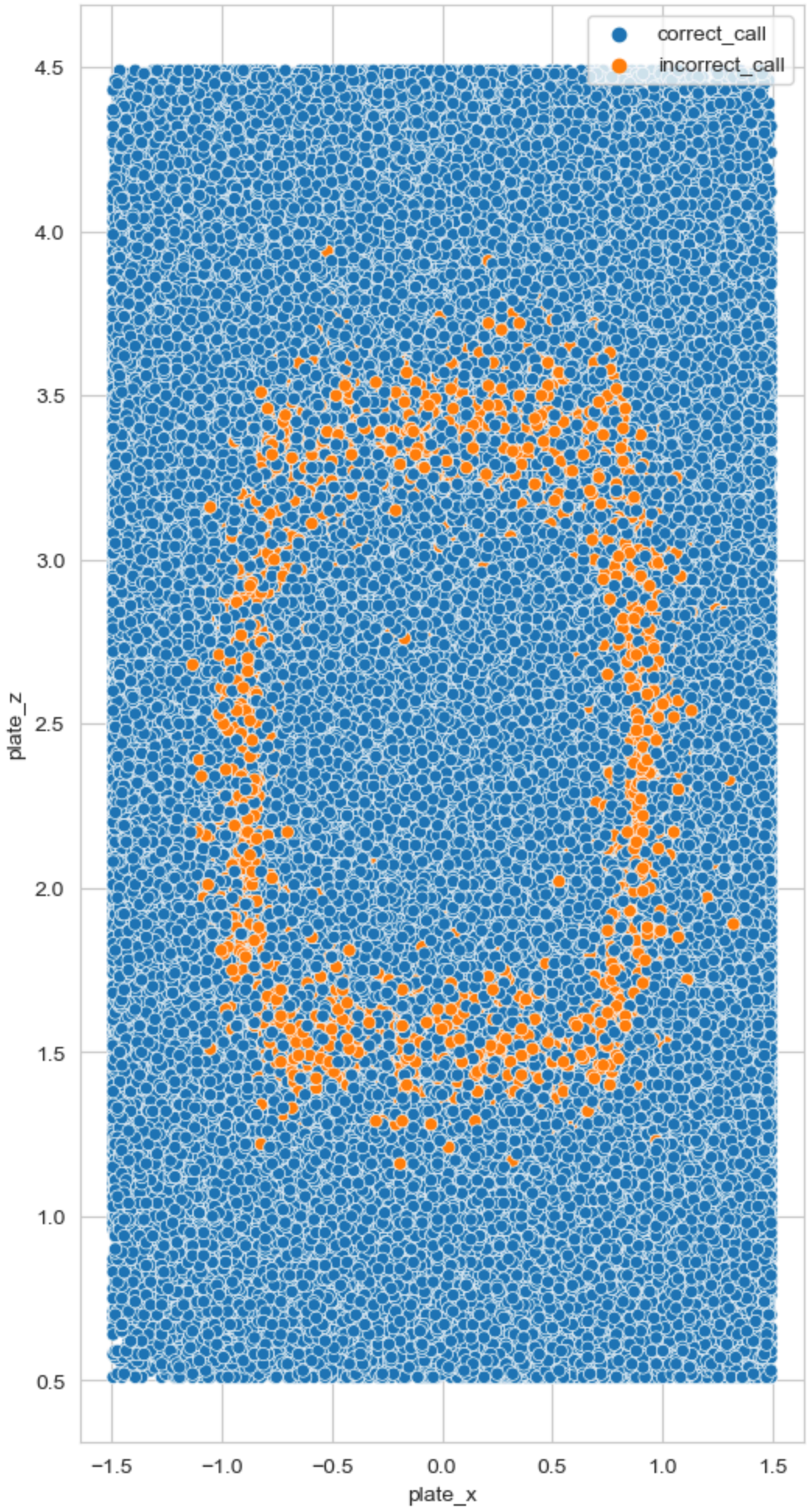
In [4]:
```
plt.figure(figsize=(6,12))
sns.scatterplot(x='plate_x', y='plate_z', data=core, hue='correct_call')
plt.legend(loc='upper right')
plt.show()
```


correct_call
incorrect_call
plate_z
plate_x
10
8
6
4
2
0
−2
−4
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0

This plot is consistent with the fact that about 91.7% of calls were made correctly.

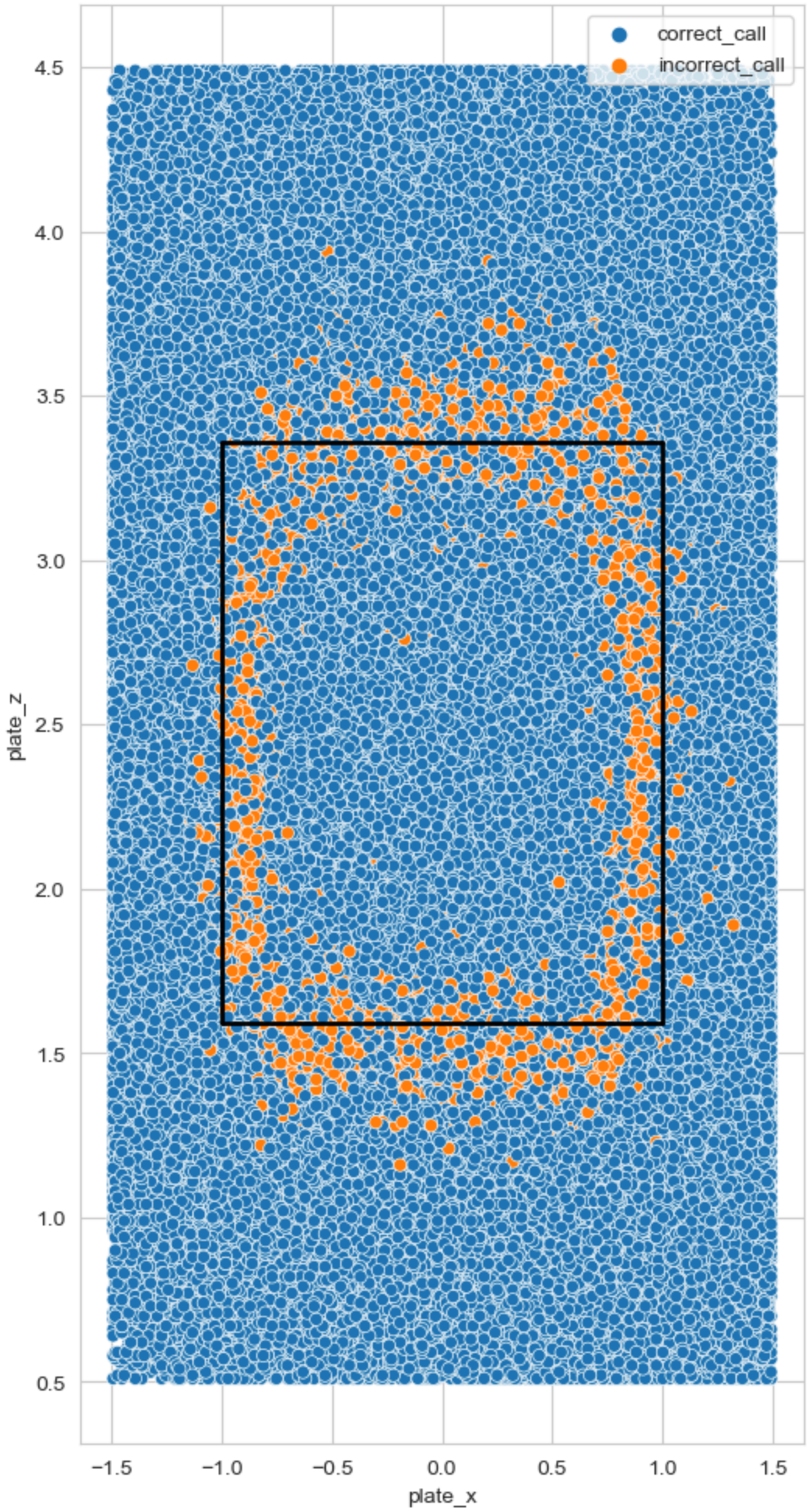
We now re-create this picture, but more centered on the strike zone. As a reminder, the width of the regulation stike zone is constant (it goes from -0.833 ft to 0.833 ft); the height of the strike zone depends on the batter. We will restrict to pitches with a `plate_x` value between -1.5 and 1.5 and with a `plate_z` value between 0.5 and 4.5.

In [5]:
```
zoom = core[(-1.5 < core['plate_x']) & (1.5 > core['plate_x']) & (0.5 < core['plate_z'

plt.figure(figsize=(6,12))
sns.scatterplot(x='plate_x', y='plate_z', data=zoom, hue='correct_call')
plt.legend(loc='upper right')
plt.show()
```


correct_call
incorrect_call
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
plate_z
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
plate_x

We now recreate this picture with lines defining the strike zone width and (average) strike zone height to highlight common knowledge: most missed calls occur near the boundary of the strike zone.

In [6]:
```
plt.figure(figsize=(6,12))
sns.scatterplot(x='plate_x', y='plate_z', data=zoom, hue='correct_call')
plt.legend(loc='upper right')
avg_top = core.sz_top.mean()
avg_bot = core.sz_bot.mean()
plt.plot([-1, 1], [avg_top, avg_top], color='black', linewidth=2)
plt.plot([-1, 1], [avg_bot, avg_bot], color='black', linewidth=2)
plt.plot([-1, -1], [avg_bot, avg_top], color='black', linewidth=2)
plt.plot([1, 1], [avg_bot, avg_top], color='black', linewidth=2)
plt.show()
```


correct_call
incorrect_call
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
plate_z
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
plate_x

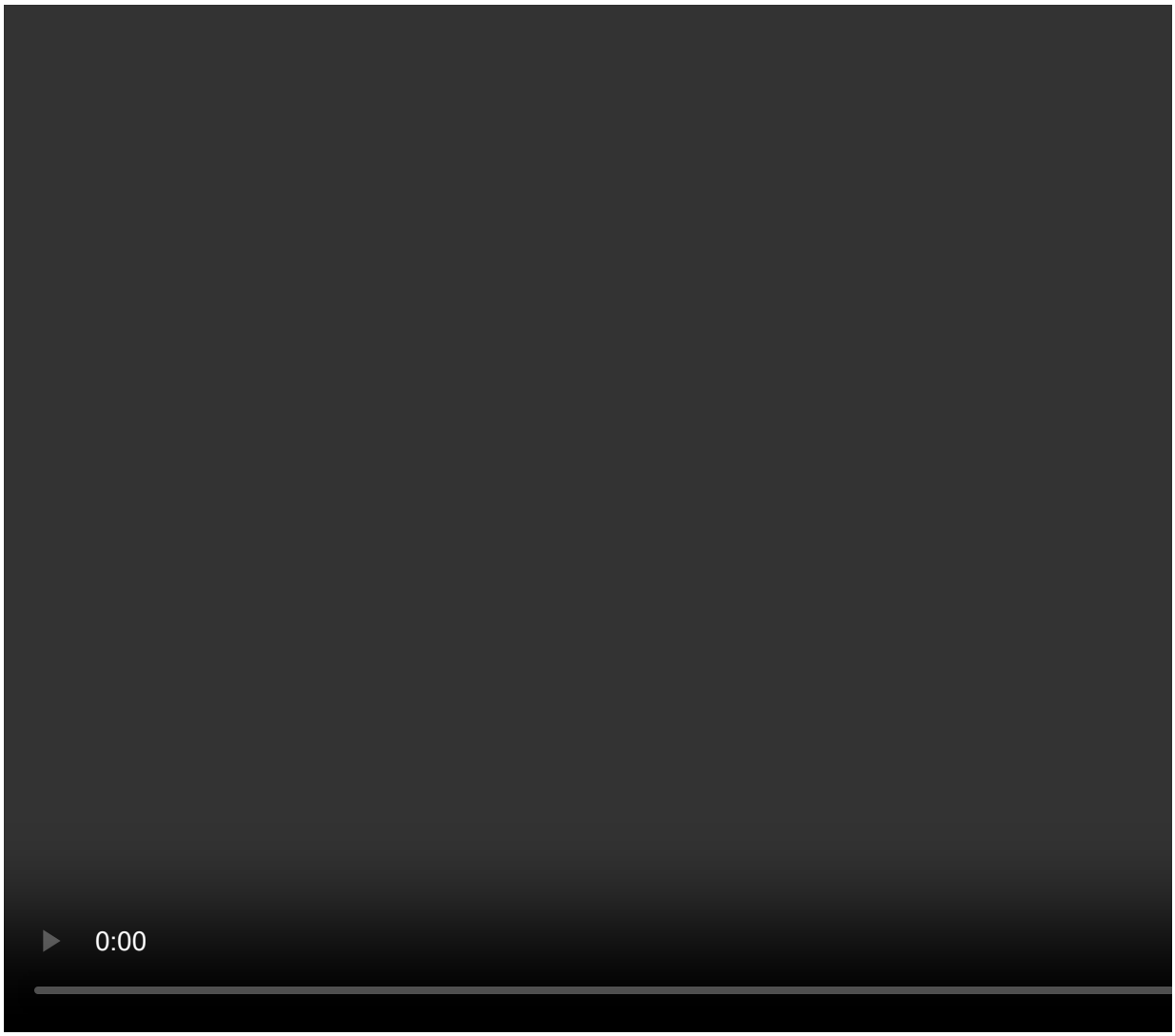
2.1.1 - Four Regions of the Zones: Heart, Shadow, Chase, Waste (HSCW)

Given that most missed calls happen near the edge of the zone, we will take note of a standard method of breaking the pitching plane into four zones: the heart, shadow, chase, and waste zones. This is given concretely in the picture below (originally found at https://tangotiger.net/strikezone/zone%20chart.png, which was linked to from Baseball Savant), but we give a short written description as well.

- Pitches in the heart of the strike zone are always regulation strikes; it is the dead center of the strike zone.
- Pitches in the shadow zone are close to the border of the regulation zone; this region incorporates both true strikes and true balls.
- Pitches in the chase portion of the zone are all true balls, but pitches that a hitter may still reasonably swing at (such as a curveball with a large break on an 0-2 count).
- Finally, pitches in the waste zone are balls that are not close at all to being strikes.

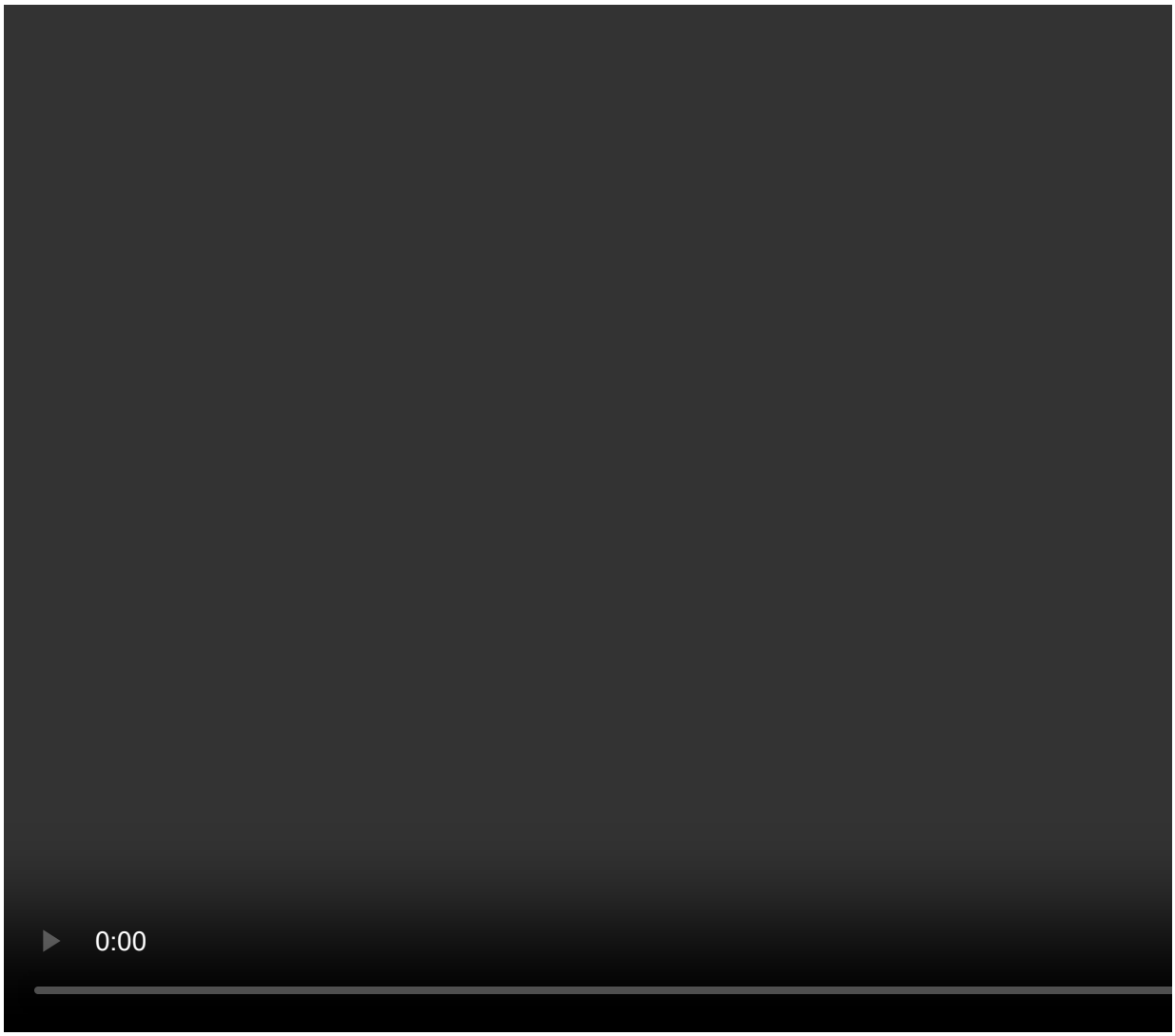
We also include the following video clips from Baseball Savant as specific examples:

Heart Example



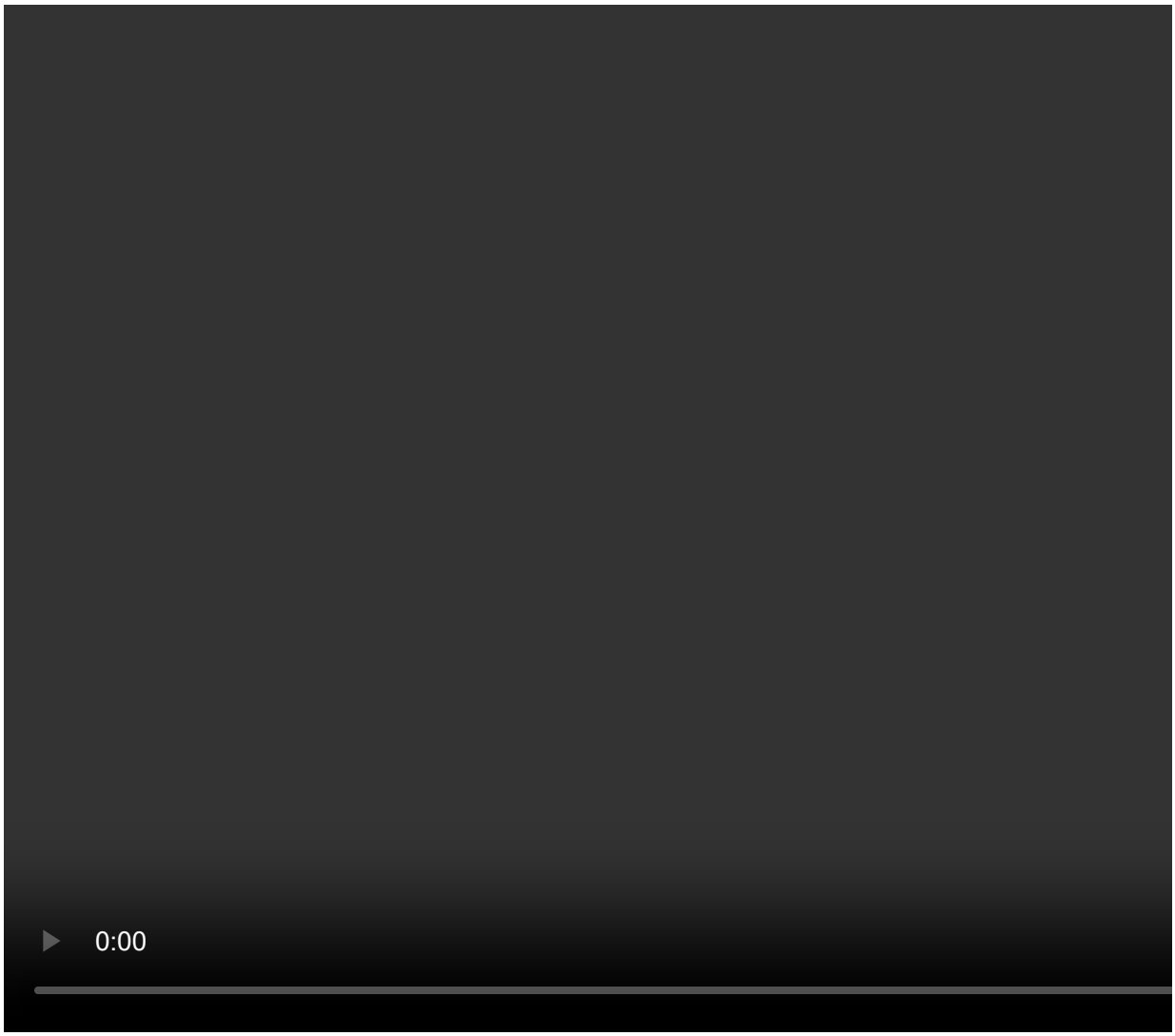
Heart Video Source

Shadow Example



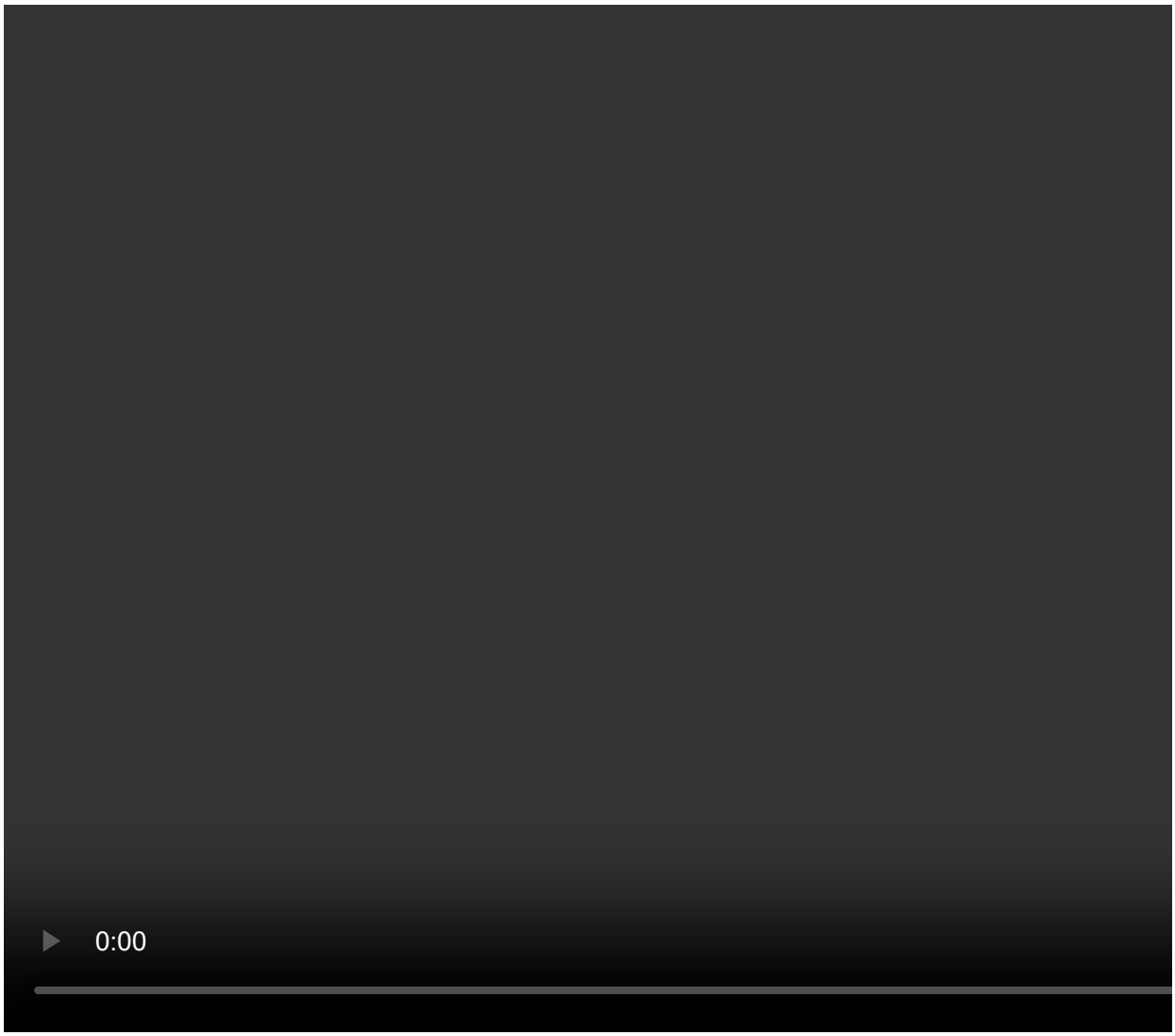
Shadow Video Source

Chase Example



Chase Video Source

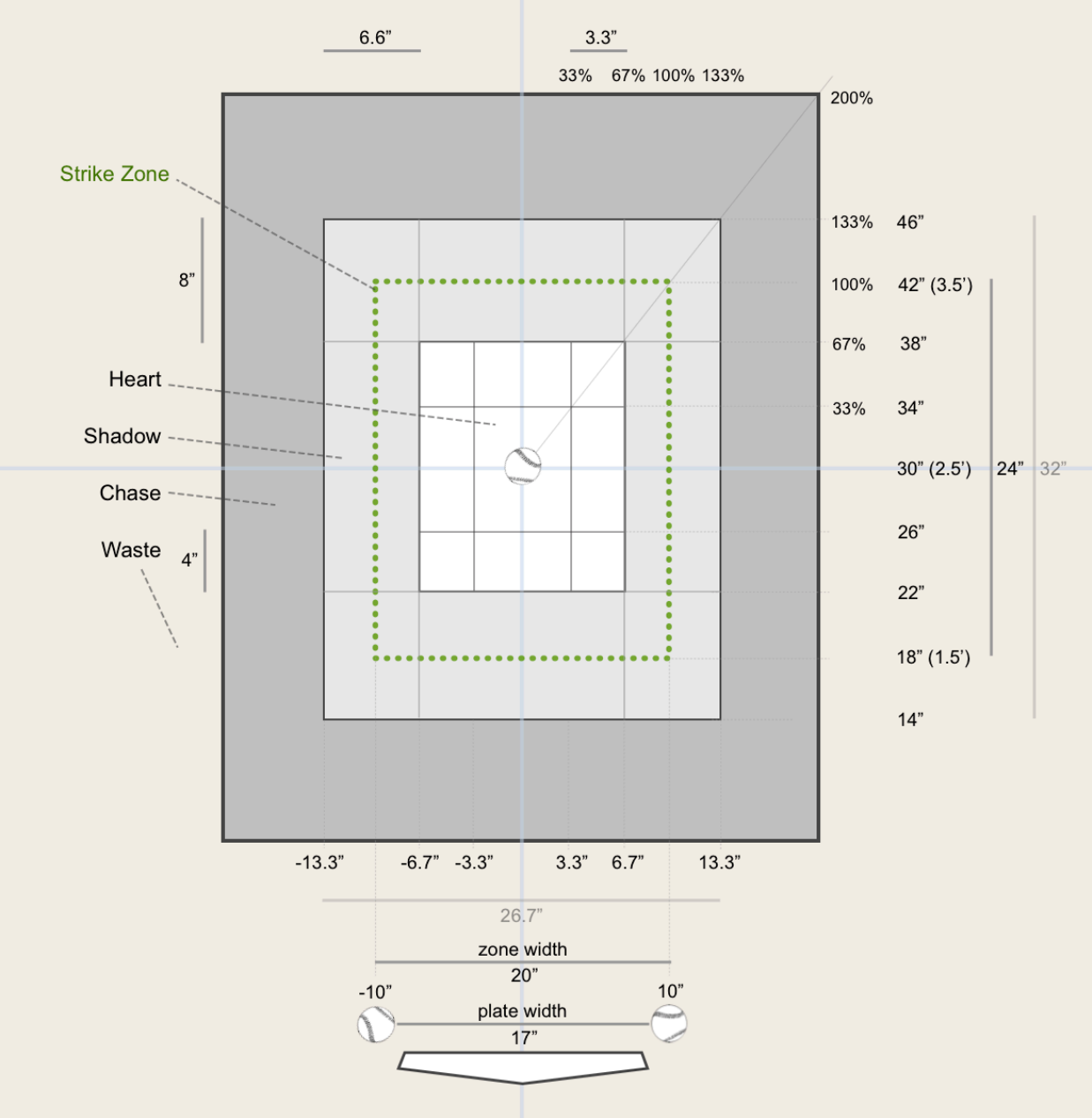
Waste Example



Waste Video Source

In [7]:
```
Image("heart_shadow_chase_waste.png")
```

Out[7]:

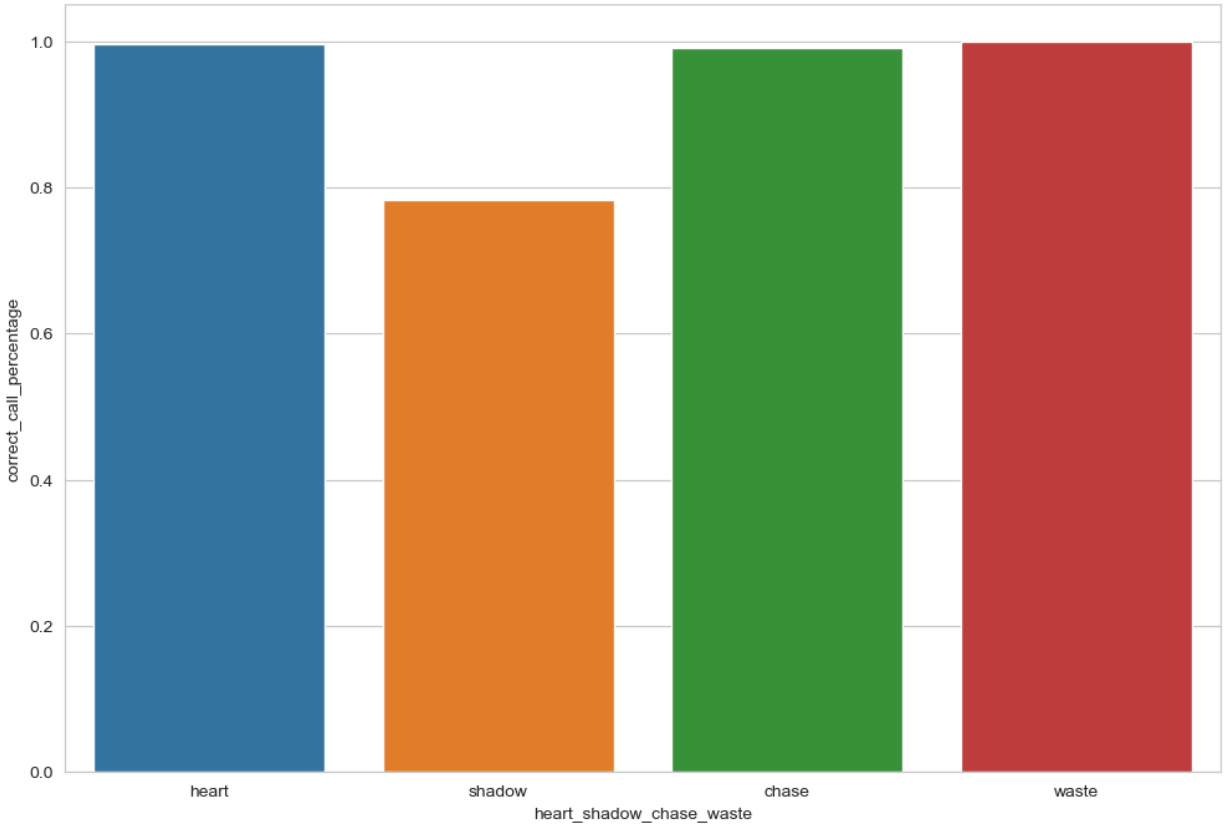


Now, let's examine umpire accuracy in these four regions. As can be seen below, umpire accuracy is fairly high overall, but is very high in the heart, chase, and waste zones. The shadow of the zone is the most difficult region for umpires to call correctly.

In [8]:

```
hscw_names = ['heart', 'shadow', 'chase', 'waste']
hscw_correct = []
hscw_incorrect = []

for string in hscw_names:
    acc_list = core[core['hscw']==string].correct_call.value_counts(normalize=True).to
    hscw_correct.append(acc_list[0])
    hscw_incorrect.append(acc_list[1])

hscw_eda = pd.DataFrame({
    'heart_shadow_chase_waste':hscw_names,
    'correct_call_percentage':hscw_correct,
    'incorrect_per':hscw_incorrect
    }
```

```
)

plt.figure(figsize=(12,8))
sns.barplot(hscw_eda, x='heart_shadow_chase_waste', y='correct_call_percentage')
plt.show()
```

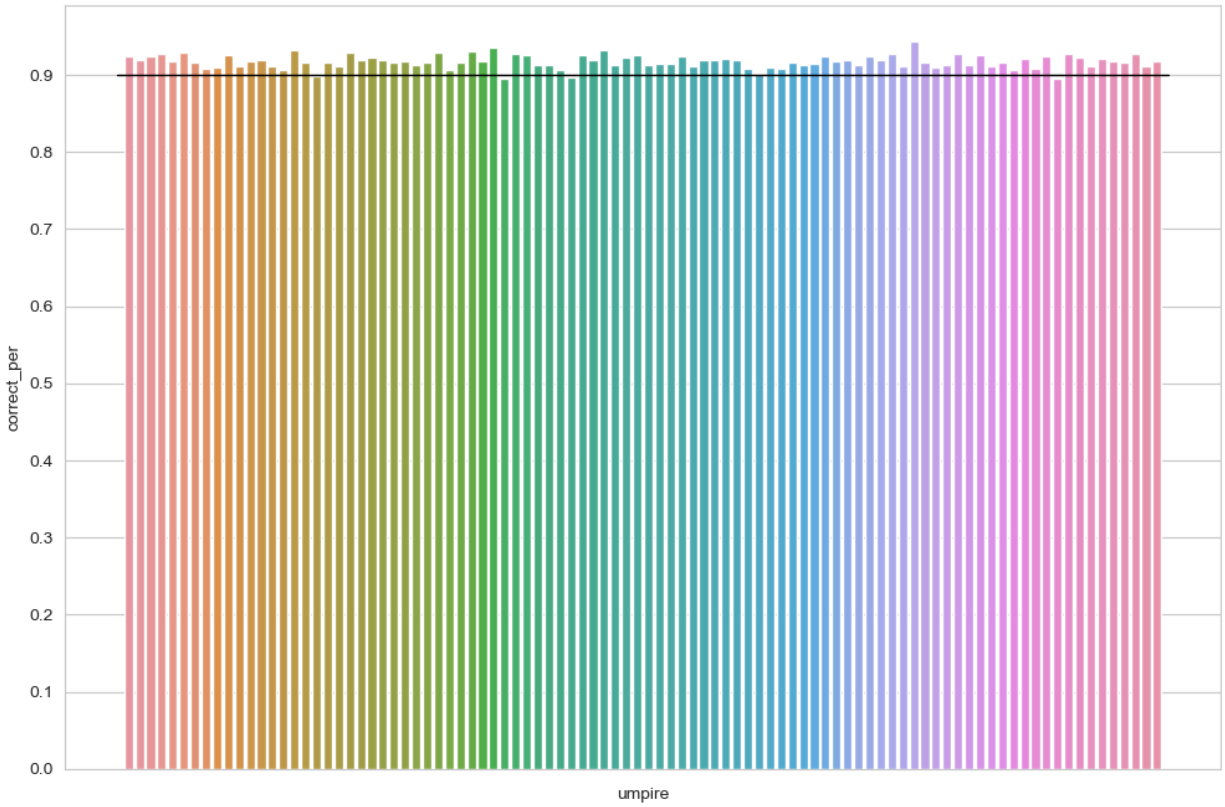


2.1.2 - Accuracy By Umpire

We will also look at some of these metrics by umpire. In the bar plot below, we see that umpire accuracy on these pitches is fairly uniform - most are above 90%, for example.

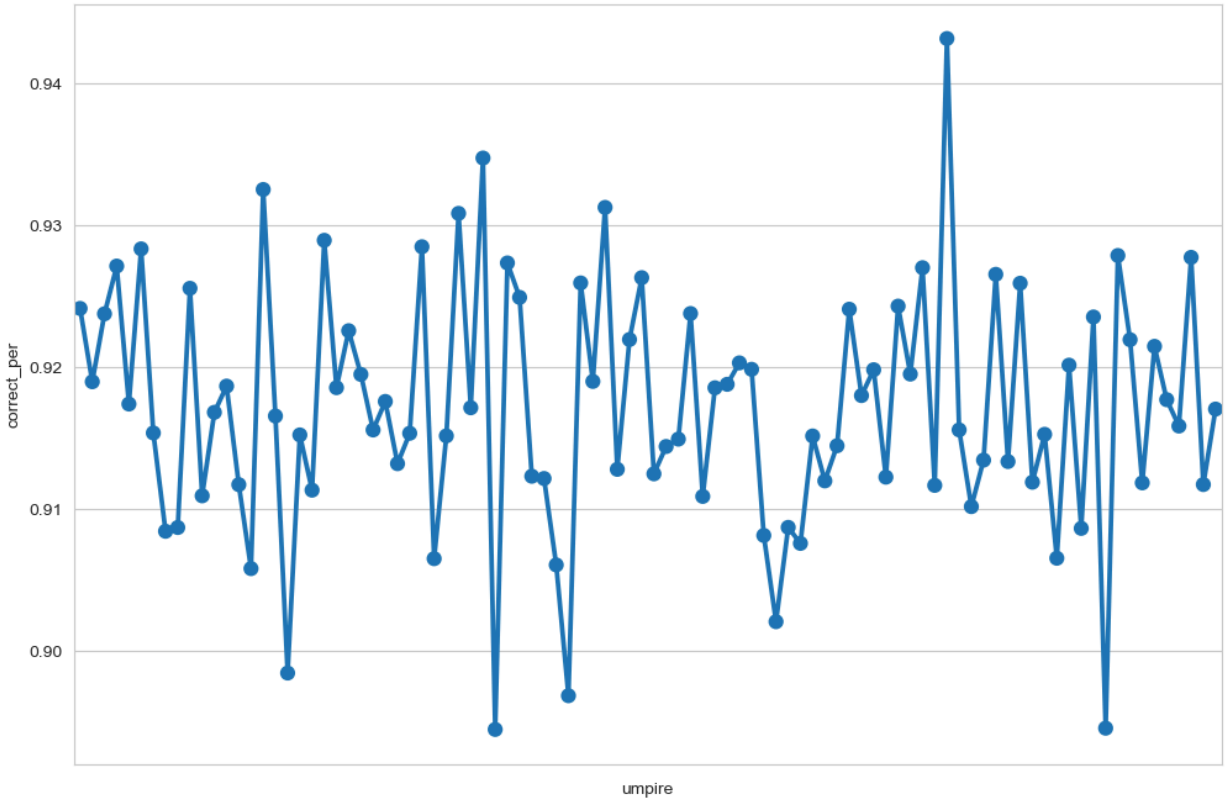
In [9]:
```
umpire_list = list(core.umpire.unique())

umpire_list.sort()

umpires_correct = []
umpires_incorrect = []

for umpire in umpire_list:
    acc_list = core[core['umpire']==umpire].correct_call.value_counts(normalize=True).
    umpires_correct.append(acc_list[0])
    umpires_incorrect.append(acc_list[1])

umpires_eda = pd.DataFrame({'umpire':umpire_list, 'correct_per':umpires_correct, 'incc

plt.figure(figsize=(12,8))

sns.barplot(umpires_eda, x='umpire', y='correct_per')
plt.gca().set_xticklabels([])
plt.gca().set_yticks([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
plt.gca().set_yticklabels([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
```

```
plt.plot([-1, 94], [0.9, 0.9], color='black', linewidth=1)
plt.show()
```



Nonetheless, we can use a `pointplot` to zoom in on the the differences.

In [10]:
```
plt.figure(figsize=(12,8))
sns.pointplot(umpires_eda, x='umpire', y='correct_per')
plt.gca().set_xticklabels([])
plt.show()
```



We can sort `umpires_eda` by correct call percentage to highlight the range:

In [11]:
```
umpires_eda.sort_values(by='correct_per', ascending=False)
```

Out[11]:

	umpire	correct_per	incorrect_per
71	Pat Hoberg	0.943174	0.056826
33	Derek Thomas	0.934751	0.065249
15	Brock Ballou	0.932533	0.067467
43	Jansen Visconti	0.931271	0.068729
31	David Arrieta	0.930851	0.069149
...	...	...	...
57	Larry Vanover	0.902074	0.097926
17	CB Bucknor	0.898455	0.101545
40	Hunter Wendelstedt	0.896856	0.103144
84	Scott Barry	0.894567	0.105433
34	Doug Eddings	0.894477	0.105523

94 rows × 3 columns

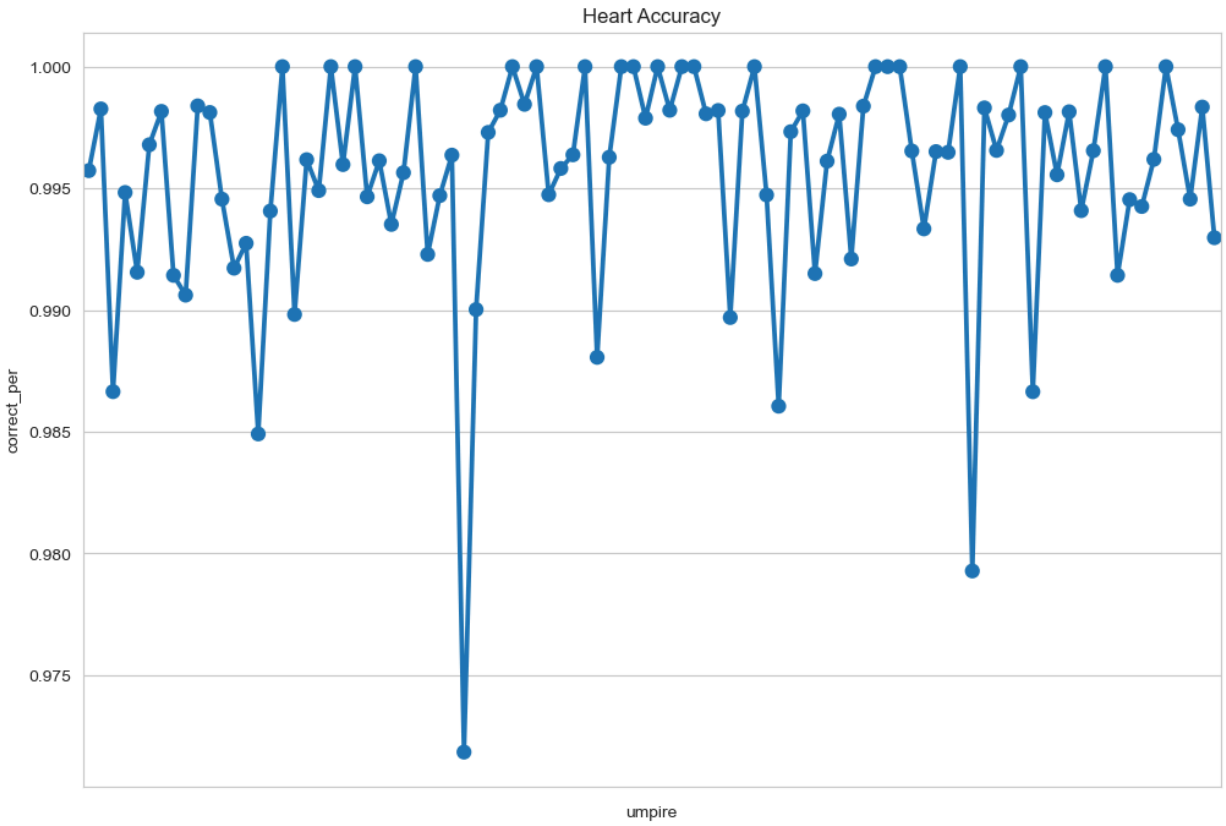
Notably, Pat Hoberg has the highest percentage of correct calls (approximately 94.3%) and Doug Eddings has the lowest percentage of correct calls (approximately 89.4%).

2.1.3 - Heart, Shadow, Chase, Waste Accuracy By Umpire

We will also examine individual umpire performance in the heart, shadow, chase, and waste zones. We will first look at a `pointplot` of the accuracy in the heart of the zone.

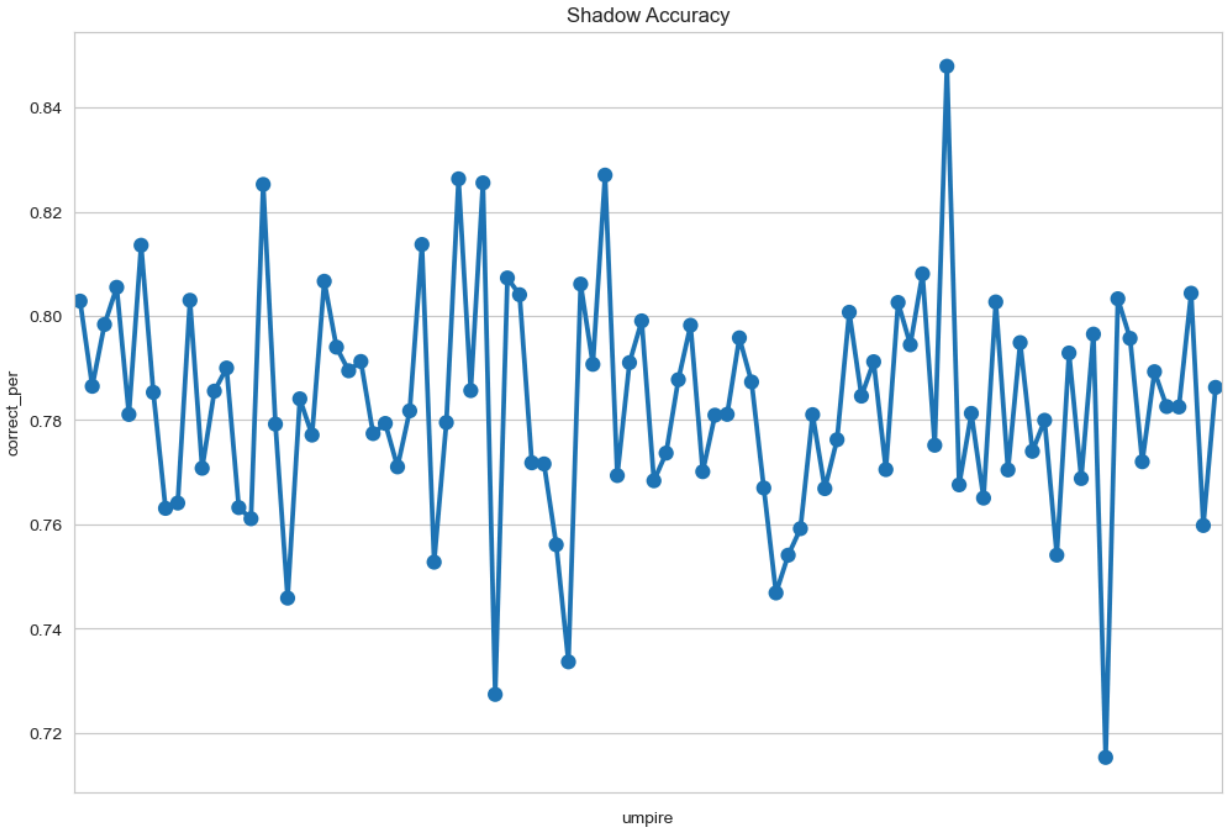
In [12]:
```
uhcc = core.groupby(['umpire', 'hscw']).correct_call.value_counts().unstack(fill_value
    columns={0:'count'}
)

umpires_heart = []
umpires_shadow = []
umpires_chase = []
umpires_waste = []

for umpire in umpire_list:
    h_cor = uhcc.at[(umpire, 'heart', 'correct_call'), 'count']
    h_inc = uhcc.at[(umpire, 'heart', 'incorrect_call'), 'count']
    umpires_heart.append( h_cor / (h_cor + h_inc) )
    s_cor = uhcc.at[(umpire, 'shadow', 'correct_call'), 'count']
    s_inc = uhcc.at[(umpire, 'shadow', 'incorrect_call'), 'count']
    umpires_shadow.append( s_cor / (s_cor + s_inc) )
    c_cor = uhcc.at[(umpire, 'chase', 'correct_call'), 'count']
    c_inc = uhcc.at[(umpire, 'chase', 'incorrect_call'), 'count']
    umpires_chase.append( c_cor / (c_cor + c_inc) )
    w_cor = uhcc.at[(umpire, 'waste', 'correct_call'), 'count']
    w_inc = uhcc.at[(umpire, 'waste', 'incorrect_call'), 'count']
    umpires_waste.append( w_cor / (w_cor + w_inc) )

umpires_heart_df = pd.DataFrame({'umpire':umpire_list, 'correct_per':umpires_heart})
umpires_shadow_df = pd.DataFrame({'umpire':umpire_list, 'correct_per':umpires_shadow})
umpires_chase_df = pd.DataFrame({'umpire':umpire_list, 'correct_per':umpires_chase})
umpires_waste_df = pd.DataFrame({'umpire':umpire_list, 'correct_per':umpires_waste})
```

In [13]:
```
plt.figure(figsize=(12,8))
sns.pointplot(umpires_heart_df, x='umpire', y='correct_per').set_title('Heart Accuracy
plt.gca().set_xticklabels([])
plt.show()
```



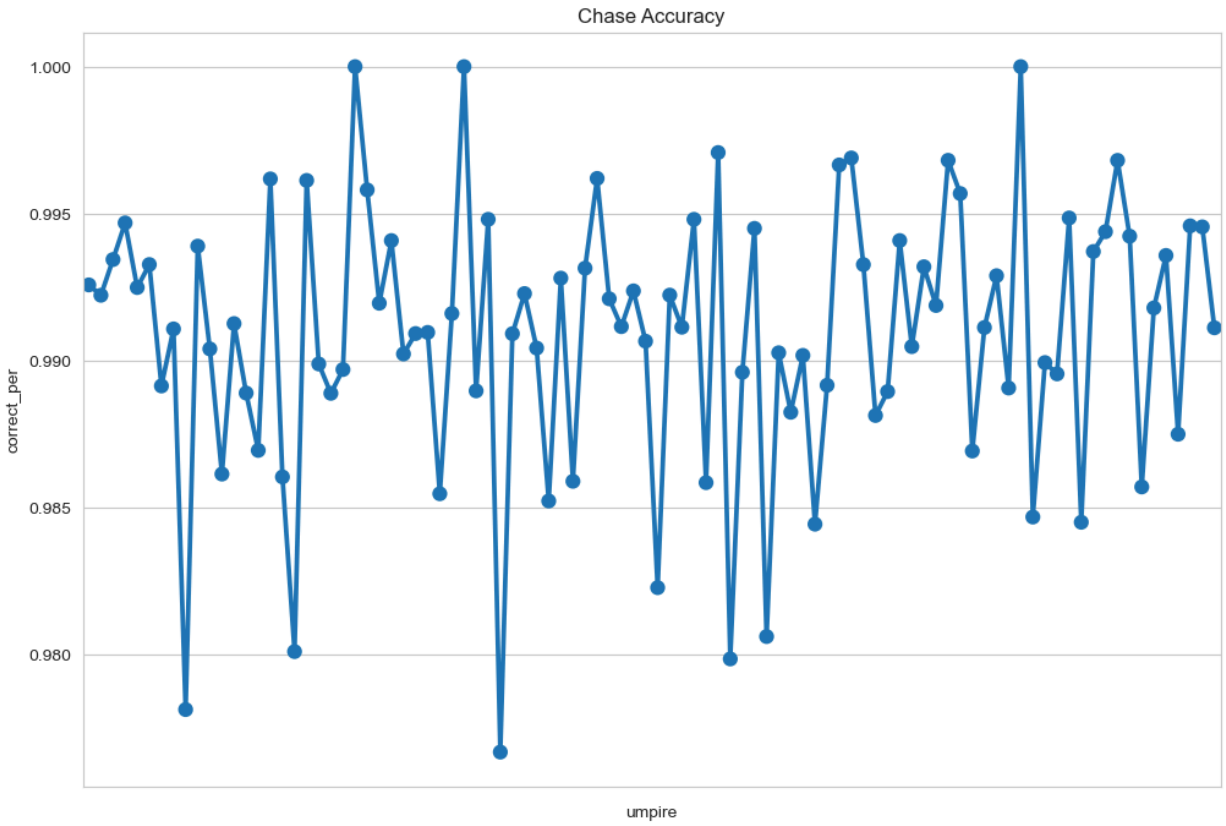
In the heart of the zone, everyone does well; umpires called pitches correctly between 97% and 100% of the time. As noted earlier, the lowest levels of accuracy occur in the shadow zone:

In [14]:
```
plt.figure(figsize=(12,8))
sns.pointplot(umpires_shadow_df, x='umpire', y='correct_per').set_title('Shadow Accura
plt.gca().set_xticklabels([])
plt.show()
```



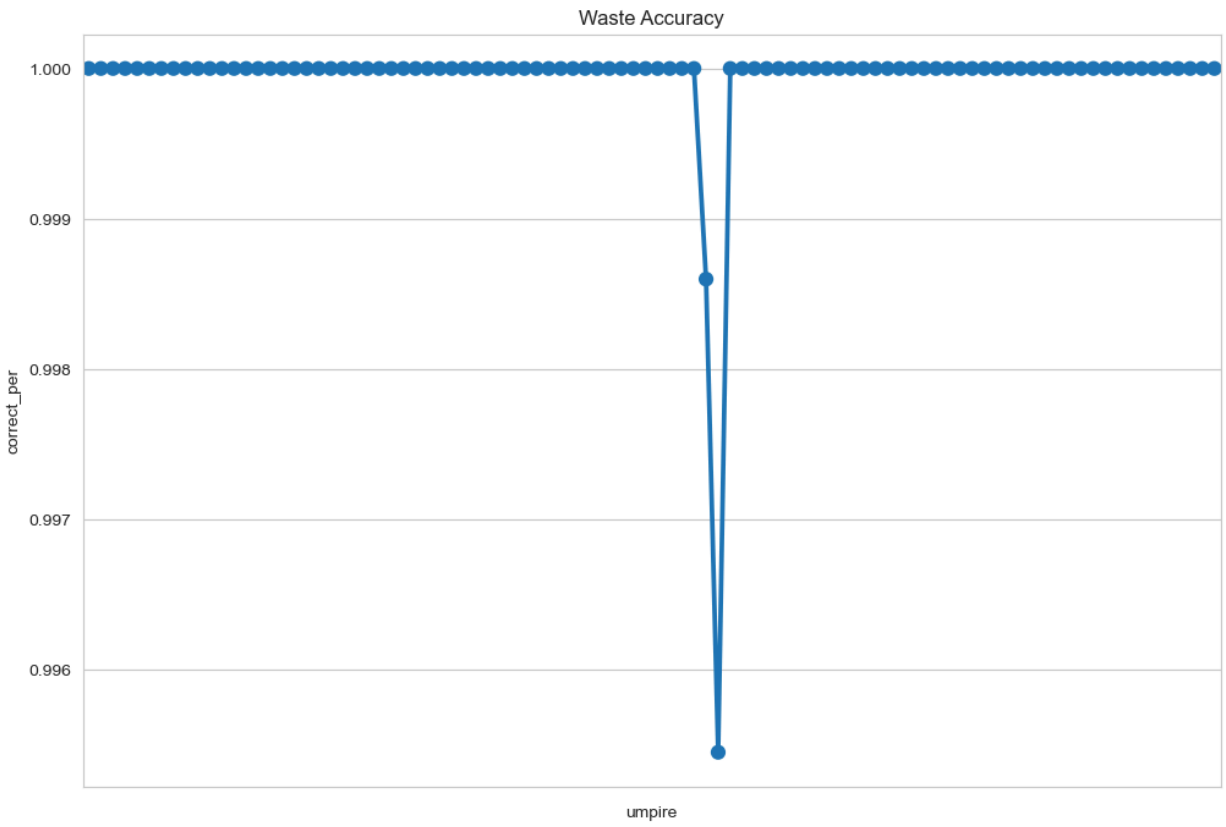
Accuracy in the chase zone is similar to in the heart:

In [15]:
```
plt.figure(figsize=(12,8))
sns.pointplot(umpires_chase_df, x='umpire', y='correct_per').set_title('Chase Accuracy
plt.gca().set_xticklabels([])
plt.show()
```



In the waste zone, we see almost perfect accuracy, as would be expected:

In [16]:
```
plt.figure(figsize=(12,8))
sns.pointplot(umpires_waste_df, x='umpire', y='correct_per').set_title('Waste Accuracy
plt.gca().set_xticklabels([])
plt.show()
```



Upon inspection, one can see that the only umpires who missed calls in the waste zone are Jordan Baker (3 missed calls) and John Tumpane (1 missed call).

In [17]:
```
umpires_waste_df.sort_values('correct_per')
```

Out[17]:

	umpire	correct_per
52	Jordan Baker	0.995448
51	John Tumpane	0.998597
0	Adam Beck	1.000000
67	Nate Tomlinson	1.000000
66	Mike Muchlinski	1.000000
...	...	...
27	D.J. Reyburn	1.000000
26	Cory Blaser	1.000000
25	Clint Vondrak	1.000000
34	Doug Eddings	1.000000
93	Will Little	1.000000

94 rows × 2 columns

In [18]:
```
core[(core['umpire']=='Jordan Baker') & (core['hscw']=='waste') & (core['correct_call'
```

Out[18]:

	ball/strike	binary_bs	true_ball/strike	correct_call	zone	hscw	plate_x	plate_x_mag	plate_x
74903	S	1	true_ball	incorrect_call	12.0	waste	0.90	0.90	
74904	S	1	true_ball	incorrect_call	12.0	waste	0.40	0.40	
74910	S	1	true_ball	incorrect_call	12.0	waste	0.64	0.64	

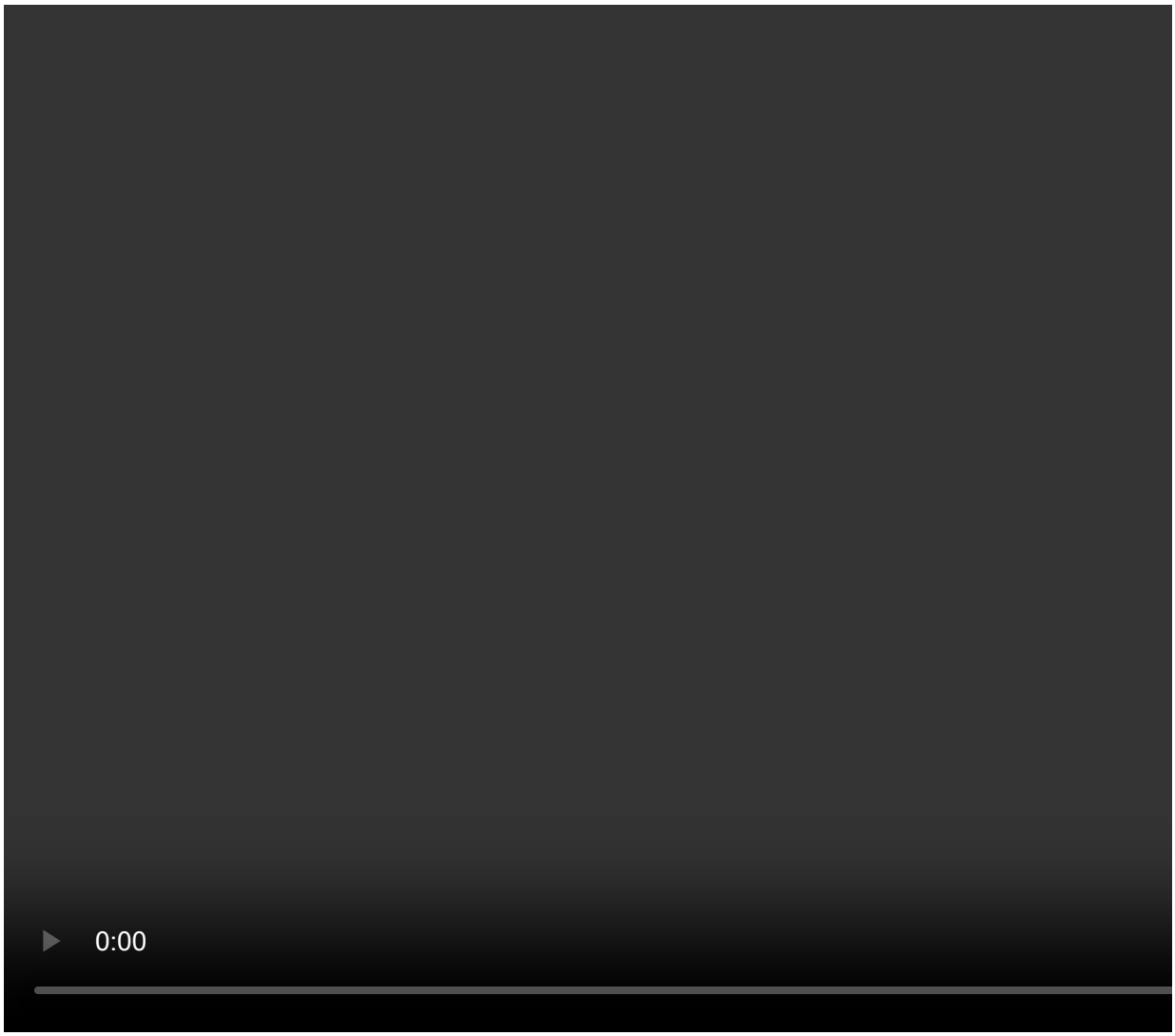
In [19]:
```
core[(core['umpire']=='John Tumpane') & (core['hscw']=='waste') & (core['correct_call'
```

Out[19]:

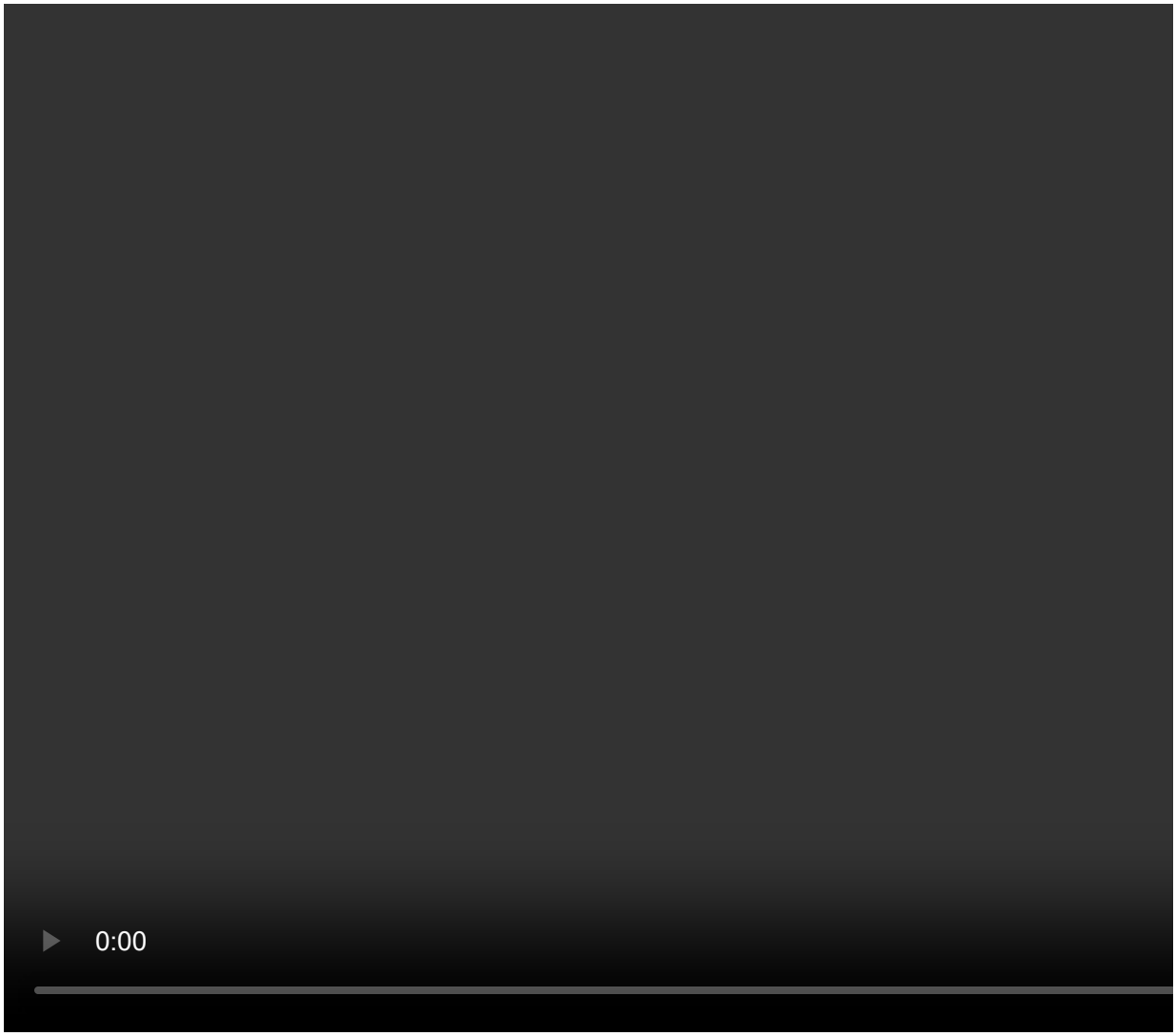
	ball/strike	binary_bs	true_ball/strike	correct_call	zone	hscw	plate_x	plate_x_mag	plate_
287843	S	1	true_ball	incorrect_call	12.0	waste	0.58	0.58	

These are irregular pitches, as can be seen from the following videos (provided by Baseball Savant). In each case, it is a position player lobbing a slow, looping pitch which drops drastically.

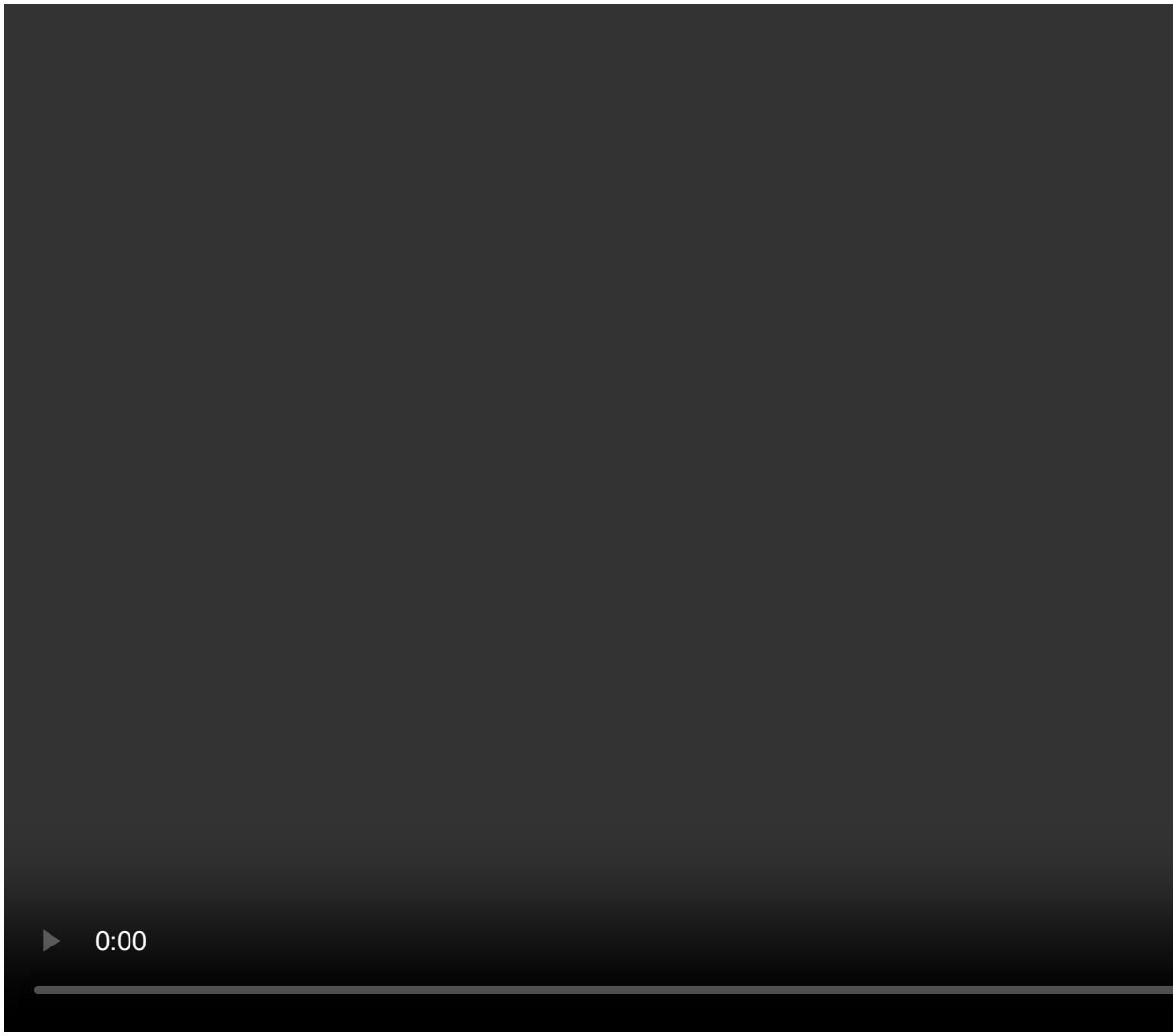
For Jordan Baker, these three pitches are by Ryan McKenna (primarily an outfielder):



Baker Video 1 Source

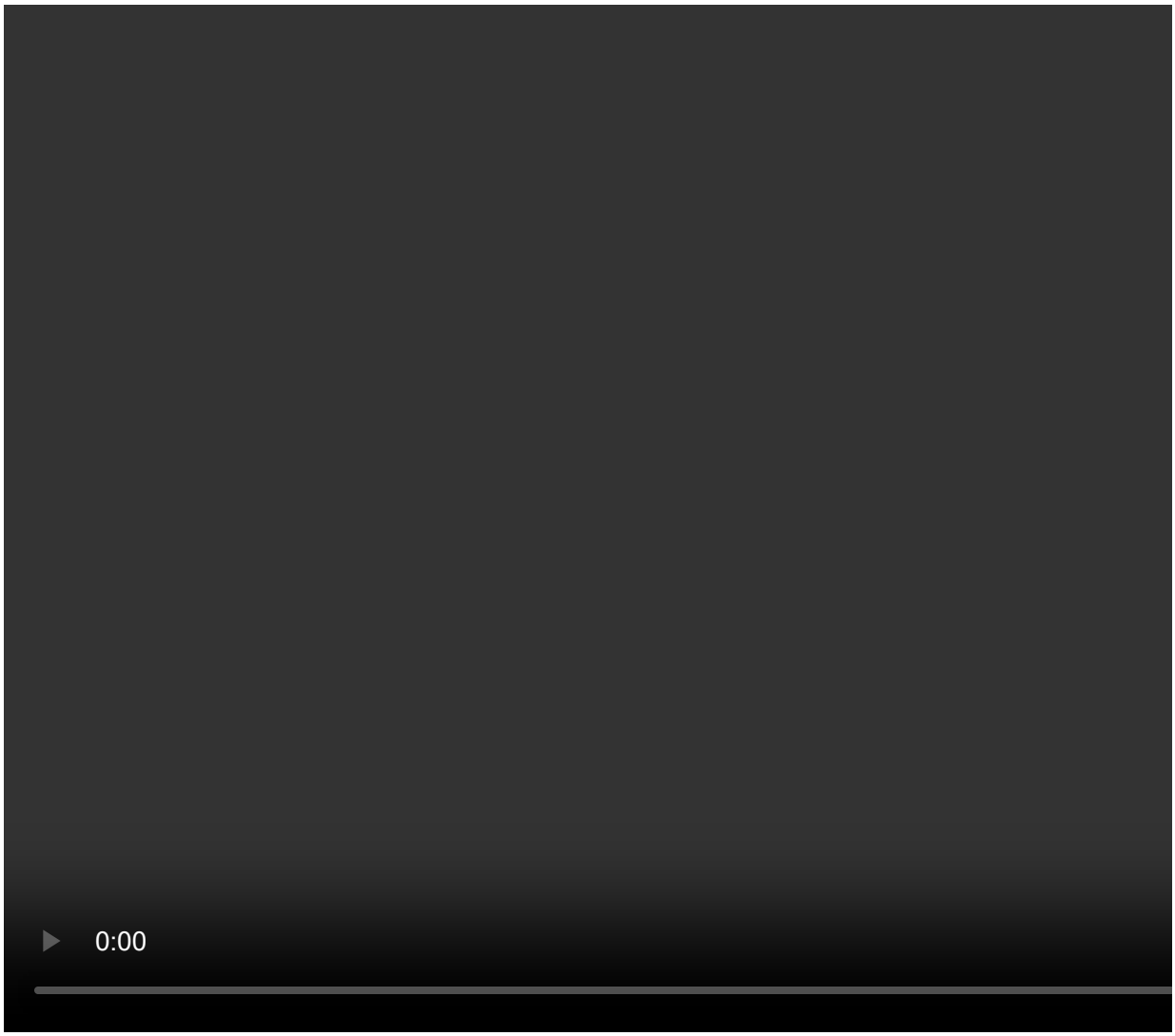


Baker Video 2 Source



Baker Video 3 Source

For John Tumpane, this pitch is by Tucker Barnhart (primarily a catcher):



Tumpane Video Source

2.2 - Modeling with Horizontal Location Only: Predicting Called Balls and Strikes

Having examined the relevant features for this section, we proceed to modeling.

As we will address later, there are subtleties in dealing with vertical pitch location (the `plate_z` feature), as the strike zone height varies by batter. As such, we will begin by only considering horizontal pitch location. Further, instead of using the standard `plate_x` feature directly, we will instead use `plate_x_mag`. While `plate_x` records the signed distance from the plate (with one direction being positive and the other negative), `plate_x_mag` is the absolute value of `plate_x` (with sign recorded in the `plate_x_dir` feature).

Since we will not yet be using the height for any pitches, we will create the `horizontal` data frame that contains only pitches that are either: 1) true strikes (pitches with both `plate_x` and `plate_z` values within the regulation zone), or 2) true balls with both `plate_x` and `plate_z` values outside of the regulation zone range.

Otherwise, we would be including information which umpires would be able to differentiate that this model cannot.

Finally, recall that `plate_x` and `plate_z` are given in feet; the width of the regulation strike zone is from -10in to 10in, which is -0.83ft to 0.83 ft.

In [20]:
```
true_strikes = core[core['true_ball/strike']=='true_strike']

left_down = core[ (core['plate_x'] < -0.83) & (core['plate_z'] < core['sz_bot']) ]
right_down = core[ (core['plate_x'] > 0.83) & (core['plate_z'] < core['sz_bot']) ]

left_up = core[ (core['plate_x'] < -0.83) & (core['plate_z'] > core['sz_top']) ]
right_up = core[ (core['plate_x'] > 0.83) & (core['plate_z'] > core['sz_top']) ]

merge_list = [true_strikes, left_down, right_down, left_up, right_up]

horizontal = pd.concat(merge_list, ignore_index=True)
```

2.2.1 - Train-Test Splits

We begin by separating into training data and test data.

In [21]:
```
hor_train, hor_test = train_test_split(horizontal,
                                       test_size=0.2,
                                       shuffle=True,
                                       random_state=630,
                                       stratify=horizontal['ball/strike'])
```

We stratify based on balls and strikes see similar percentages for called balls and called strikes in both `horizontal` and `hor_train`.

In [22]:
```
horizontal['ball/strike'].value_counts(normalize=True)
```

Out[22]:
```
S    0.560544
B    0.439456
Name: ball/strike, dtype: float64
```

In [23]:
```
hor_train['ball/strike'].value_counts(normalize=True)
```

Out[23]:
```
S    0.560544
B    0.439456
Name: ball/strike, dtype: float64
```

2.2.2 - Baseline Model for `Horizontal` Data Frame

For our baseline model with the `horizontal` data frame, we randomly predict balls and strikes according to the overall distribution in the data frame (i.e. with a 44% chance of a given pitch being called a strike). We will use 1000 random samples in our baseline model.

In [24]:
```
hor_baseline_accs = []
hor_baseline_f1s = []
```

```
hor_baseline_praucs = []

np.random.seed(329)

for obs in range(1000):
    rand_draw = np.random.binomial(n=1, p=0.44, size=len(hor_train))
    hor_baseline_accs.append( accuracy_score(hor_train.binary_bs.values, rand_draw) )
    precision, recall, _ = precision_recall_curve(hor_train.binary_bs.values, rand_dra
    hor_baseline_praucs.append( auc(recall, precision) )
    hor_baseline_f1s.append( f1_score(hor_train.binary_bs.values, rand_draw) )

hor_baseline_stats = pd.DataFrame(data={
                                'hor_baseline_accuracy':hor_baseline_accs,
                                'hor_baseline_f1':hor_baseline_f1s,
                                'hor_baseline_pr_auc':hor_baseline_praucs
                                })

hor_baseline_stats.describe()
```

Out[24]:

	hor_baseline_accuracy	hor_baseline_f1	hor_baseline_pr_auc
count	1000.000000	1000.000000	1000.000000
mean	0.492673	0.492914	0.657168
std	0.001333	0.001590	0.001058
min	0.488969	0.488043	0.654059
25%	0.491775	0.491841	0.656444
50%	0.492705	0.492906	0.657203
75%	0.493562	0.493968	0.657905
max	0.496775	0.497711	0.660372

2.2.3 - Logistic Regression - Horizontal, No Umpires

We begin in the simplest case, using only `plate_x_mag` as a predictor. The first step is setting up k-fold cross validation. We will always use k=10 in our k-fold cross validation.

In [25]:
```
kfold_splits = 10

hor_no_ump_log_reg_kfold_rand_state = 721

hor_no_ump_log_reg_kfold = StratifiedKFold(kfold_splits, shuffle=True, random_state=ho
```

Our logistic regression model object requires uses both a penalty (either l2 or none) and a maximum number of iterations. The following code uses `GridSearchCV` to determine optimal choices for both of these options. Given the run time, we provide commented-out code to run the search, export the results as a CSV, and read-in the results as a data frame.

There are several options which seem to provide the best test score. Given the similarity in scores, we pick no penalty and a maximum of 10,000 iterations.

In [26]:
```
#hor_no_ump_log_reg_preprocessor = ColumnTransformer(
#    transformers=[
#            ('continuous', StandardScaler(), ['plate_x_mag'])
#        ],
#    remainder='passthrough')

## Setting up the normalization pipeline
#hor_no_ump_log_reg_pipeline = Pipeline([
#    ('log_reg_preprocessor', hor_no_ump_log_reg_preprocessor),
#    ('log_reg', LogisticRegression())
#])

## Creating the grid
#hor_no_ump_log_reg_grid = GridSearchCV(
#    hor_no_ump_log_reg_pipeline,
#    param_grid={
#        'log_reg__penalty':['l2', None],
#        'log_reg__max_iter':[100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000
#    },
#    scoring='accuracy',
#    cv=5
#)

## Fitting the grid
#hor_no_ump_log_reg_grid.fit(hor_train[['plate_x_mag']], hor_train.binary_bs)
```

In [27]:
```
#hor_no_ump_log_reg_grid_results = pd.DataFrame(data=hor_no_ump_log_reg_grid.cv_result

#hor_no_ump_log_reg_grid_results_order = ['rank_test_score', 'std_test_score', 'param_
#hor_no_ump_log_reg_grid_results_order.extend(['param_log_reg__max_iter', 'mean_fit_ti
#hor_no_ump_log_reg_grid_results_order.extend(['std_score_time', 'split0_test_score',
#hor_no_ump_log_reg_grid_results_order.extend(['split2_test_score', 'split3_test_score

#hor_no_ump_log_reg_grid_results = hor_no_ump_log_reg_grid_results[hor_no_ump_log_reg_

#hor_no_ump_log_reg_grid_results.to_csv('hor_no_ump_log_reg_grid_results.csv', index=F
```

In [28]:
```
#hor_no_ump_log_reg_grid_results_csv = pd.read_csv('hor_no_ump_log_reg_grid_results.cs

#hor_no_ump_log_reg_grid_results_csv
```

Having determined our parameters for the model, we now run the model and observe the relevant metrics. As mentioned above, our primary metric is accuracy, with F1-score and PR-AUC serving as additional metrics. We also record the number of pitches being predicted as strikes.

In [29]:
```
hor_no_ump_log_reg_accs = []
hor_no_ump_log_reg_f1s = []
hor_no_ump_log_reg_praucs = []

hor_no_ump_log_reg_preds = []
hor_no_ump_log_reg_pred_strikes = []

feat = ['plate_x_mag']

hor_no_ump_log_reg_preprocessor = ColumnTransformer(
    transformers=[
        ('continuous', StandardScaler(), ['plate_x_mag'])
```

```
    ],
    remainder='passthrough'
)


for train_index, test_index in hor_no_ump_log_reg_kfold.split(hor_train, hor_train.bir
    # Splitting up our data using the indices from our kfold split
    split_train = hor_train.iloc[train_index]
    split_test = hor_train.iloc[test_index]

    # Setting up the normalization pipeline
    hor_no_ump_log_reg_pipeline = Pipeline([
        ('hor_no_ump_log_reg_preprocessor', hor_no_ump_log_reg_preprocessor),
        ('hor_no_ump_log_reg', LogisticRegression(penalty=None, max_iter=10000))
    ])

    # Fitting the pipeline
    hor_no_ump_log_reg_pipeline.fit(split_train[feat], split_train.binary_bs)

    # Predictions
    split_pred = hor_no_ump_log_reg_pipeline.predict(split_test[feat])
    hor_no_ump_log_reg_preds.append(split_pred)
    split_pred_dict = {1:0}
    for entry in split_pred:
        if entry==1:
            split_pred_dict[1] += 1
    hor_no_ump_log_reg_pred_strikes.append(split_pred_dict[1])

    # Evaluation metrics
    hor_no_ump_log_reg_accs.append( accuracy_score(split_test.binary_bs, split_pred) )
    hor_no_ump_log_reg_f1s.append( f1_score(split_test.binary_bs, split_pred) )
    hor_no_ump_log_reg_prec, hor_no_ump_log_reg_rec, _ = precision_recall_curve(split_
    hor_no_ump_log_reg_praucs.append(auc(hor_no_ump_log_reg_rec, hor_no_ump_log_reg_pr

hor_no_ump_log_reg_stats = pd.DataFrame(data={
    'hor_no_ump_pred_strikes':hor_no_ump_log_reg_pred_strikes,
    'hor_no_ump_log_reg_accuracy':hor_no_ump_log_reg_accs,
    'hor_no_ump_log_reg_f1':hor_no_ump_log_reg_f1s,
    'hor_no_ump_log_reg_pr_auc':hor_no_ump_log_reg_praucs
    })

hor_no_ump_log_reg_stats.describe()
```

Out[29]:

	hor_no_ump_pred_strikes	hor_no_ump_log_reg_accuracy	hor_no_ump_log_reg_f1	hor_no_ump_log
count	10.000000	10.000000	10.000000	
mean	7823.500000	0.938307	0.945313	
std	27.889664	0.002819	0.002488	
min	7779.000000	0.932023	0.939809	
25%	7804.250000	0.937519	0.944673	
50%	7827.000000	0.938445	0.945422	
75%	7835.500000	0.939822	0.946617	
max	7881.000000	0.942760	0.949277	

We now compare metrics between this model and the baseline model:

In [30]:

```
hor_no_ump_log_reg_combo_stats = pd.concat([hor_baseline_stats.describe(), hor_no_ump_

hor_no_ump_log_reg_combo_stats_order = ['hor_baseline_accuracy', 'hor_no_ump_log_reg_a
hor_no_ump_log_reg_combo_stats_order.extend(['hor_no_ump_log_reg_f1', 'hor_baseline_pr
hor_no_ump_log_reg_combo_stats = hor_no_ump_log_reg_combo_stats[hor_no_ump_log_reg_com

hor_no_ump_log_reg_combo_stats
```

Out[30]:

	hor_baseline_accuracy	hor_no_ump_log_reg_accuracy	hor_baseline_f1	hor_no_ump_log_reg_f1	h
count	1000.000000	10.000000	1000.000000	10.000000	
mean	0.492673	0.938307	0.492914	0.945313	
std	0.001333	0.002819	0.001590	0.002488	
min	0.488969	0.932023	0.488043	0.939809	
25%	0.491775	0.937519	0.491841	0.944673	
50%	0.492705	0.938445	0.492906	0.945422	
75%	0.493562	0.939822	0.493968	0.946617	
max	0.496775	0.942760	0.497711	0.949277	

We see a vast increase in accuracy for predicting umpire calls of balls and strikes using only the magnitude of the horizontal distance from the middle of the plate over the baseline model, as expected. These increases also occur for F1-score and PR-AUC.

2.2.4 - Support Vector Machines - Horizontal, No Umpires

For another point of comparison, we will use a `LinearSVC` model which also only inputs `plate_x_mag` as a feature. Again, we start with setting up our cross validation.

In [31]:
```
kfold_splits = 10

hor_no_ump_svm_kfold_rand_state = 722

hor_no_ump_svm_kfold = StratifiedKFold(kfold_splits, shuffle=True, random_state=hor_nc
```

Our `LinearSVC` model object requires a hyperparameter `C` and a maximum number of iterations. The following code uses `GridSearchCV` to determine optimal choices for both of these options. Given the run time, we provide commented-out code to run the search, export the results as a CSV, and read-in the results as a data fra,e.

There are several options which seem to provide the best test score. Given the similarity in scores, we pick `C=0.0001` and a maximum of 5,000 iterations.

In [32]:
```
#feat = ['plate_x_mag']

#hor_no_ump_svm_preprocessor = ColumnTransformer(
#    transformers=[
#            ('continuous', StandardScaler(), ['plate_x_mag'])
#        ],
#    remainder='passthrough')

## Setting up the normalization pipeline
#hor_no_ump_svm_pipeline = Pipeline([
#    ('svm_preprocessor', hor_no_ump_svm_preprocessor),
#    ('svm', LinearSVC())
#])

## Creating the grid
#hor_no_ump_grid = GridSearchCV(
#    hor_no_ump_svm_pipeline,
#    param_grid={
#        'svm__C':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
#        'svm__max_iter':[1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000]
#    },
#    scoring='accuracy',
#    cv=5
#)

## Fitting the grid
#hor_no_ump_grid.fit(hor_train[feat], hor_train.binary_bs)
```

In [33]:
```
#hor_no_ump_grid_results = pd.DataFrame(data=hor_no_ump_grid.cv_results_)

#hor_no_ump_grid_results_order = ['rank_test_score', 'std_test_score', 'param_svm__C',
#hor_no_ump_grid_results_order.extend(['mean_fit_time', 'std_fit_time', 'mean_score_ti
#hor_no_ump_grid_results_order.extend(['split1_test_score', 'split2_test_score', 'spli

#hor_no_ump_grid_results = hor_no_ump_grid_results[hor_no_ump_grid_results_order].sort

#hor_no_ump_grid_results.to_csv('hor_no_ump_svm_grid_results.csv', index=False)
```

In [34]:
```
#hor_no_ump_grid_results_csv = pd.read_csv('hor_no_ump_svm_grid_results.csv')

#hor_no_ump_grid_results_csv
```

We now run the model with our chosen parameters and observe the relevant metrics.

In [35]:
```
hor_no_ump_svm_accs = []
hor_no_ump_svm_f1s = []
hor_no_ump_svm_praucs = []
hor_no_ump_svm_pred_strikes = []

feat = ['plate_x_mag']

hor_no_ump_svm_preprocessor = ColumnTransformer(
    transformers=[
        ('continuous', StandardScaler(), ['plate_x_mag'])
        ],
    remainder='passthrough')


for train_index, test_index in hor_no_ump_svm_kfold.split(hor_train, hor_train.binary_
    # Splitting up our data using the indices from our kfold split
    split_train = hor_train.iloc[train_index]
    split_test = hor_train.iloc[test_index]

    # Setting up the normalization pipeline
    svm_pipeline = Pipeline([
        ('svm_preprocessor', hor_no_ump_svm_preprocessor),
        ('svm', LinearSVC(C=0.0001, max_iter=5000))
    ])

    # Fitting the pipeline
    svm_pipeline.fit(split_train[feat], split_train.binary_bs)

    # Predictions
    split_pred = svm_pipeline.predict(split_test[feat])
    split_pred_dict = {1:0}
    for entry in split_pred:
        if entry==1:
            split_pred_dict[1] += 1
    hor_no_ump_svm_pred_strikes.append(split_pred_dict[1])

    # Evaluation metrics
    hor_no_ump_svm_accs.append( accuracy_score(split_test.binary_bs, split_pred) )
    hor_no_ump_svm_f1s.append( f1_score(split_test.binary_bs, split_pred) )
    hor_no_ump_svm_prec, hor_no_ump_svm_rec, _ = precision_recall_curve(split_test.bir
    hor_no_ump_svm_praucs.append(auc(hor_no_ump_svm_rec, hor_no_ump_svm_prec))

hor_no_ump_svm_stats = pd.DataFrame(data={
    'hor_no_ump_svm_pred_strikes':hor_no_ump_svm_pred_strikes,
    'hor_no_ump_svm_accuracy':hor_no_ump_svm_accs,
    'hor_no_ump_svm_f1':hor_no_ump_svm_f1s,
    'hor_no_ump_svm_pr_auc':hor_no_ump_svm_praucs
    })

hor_no_ump_svm_stats.describe()
```

Out[35]:

	hor_no_ump_svm_pred_strikes	hor_no_ump_svm_accuracy	hor_no_ump_svm_f1	hor_no_ump_svm
count	10.000000	10.000000	10.000000	10
mean	8024.400000	0.944785	0.951680	0
std	28.791202	0.002450	0.002124	0
min	7988.000000	0.940660	0.948050	0
25%	8003.000000	0.943506	0.950511	0
50%	8019.500000	0.945625	0.952353	0
75%	8036.750000	0.946588	0.953274	0
max	8074.000000	0.947838	0.954259	0

Next, we compare metrics between this model and the baseline model:

In [36]:
```
hor_no_ump_svm_combo_stats = pd.concat([hor_baseline_stats.describe(), hor_no_ump_svm_

hor_no_ump_svm_combo_stats_order = ['hor_baseline_accuracy', 'hor_no_ump_svm_accuracy'
hor_no_ump_svm_combo_stats_order.extend(['hor_no_ump_svm_f1', 'hor_baseline_pr_auc', '
hor_no_ump_svm_combo_stats = hor_no_ump_svm_combo_stats[hor_no_ump_svm_combo_stats_ord

hor_no_ump_svm_combo_stats
```

Out[36]:

	hor_baseline_accuracy	hor_no_ump_svm_accuracy	hor_baseline_f1	hor_no_ump_svm_f1	hor_bas
count	1000.000000	10.000000	1000.000000	10.000000	
mean	0.492673	0.944785	0.492914	0.951680	
std	0.001333	0.002450	0.001590	0.002124	
min	0.488969	0.940660	0.488043	0.948050	
25%	0.491775	0.943506	0.491841	0.950511	
50%	0.492705	0.945625	0.492906	0.952353	
75%	0.493562	0.946588	0.493968	0.953274	
max	0.496775	0.947838	0.497711	0.954259	

This `LinearSVC` model using only `plate_x_mag` performs slightly better than the `LogisiticRegression` model using only `plate_x_mag`, but the gains relative to the baseline model are very similar.

2.3 - Modeling with Horizontal Location Only: Comparing True Balls and Strikes

Recall that we are training our models to predict whether an umpire will call a given pitch a ball or a strike, not if the pitch actually is a ball or a strike.

As another point of comparison, we will now examine how often the model predictions for pitches match the regulation strike zone compared to the umpire's call. We do this now by comparing the percentage of correct predictions (percent of model predictions that match the regulation strike zone) with the percentage of correct calls (percent of umpire calls that match the regulation strike zone).

Given the similarity in accuracy between our two models, we only do this comparison for the `LogisticRegression` model.

In [37]:
```
# Note that this requires hor_no_ump_log_reg_kfold, which is created and run in Sectio
# The code below requires that code to be run first anyway, so there should be no issu
# We reference the call in 2.2.3 so that the random states are always the same

hor_split_data_frames = []

# This fills the list created above with the split data frames
for train_index, test_index in hor_no_ump_log_reg_kfold.split(hor_train, hor_train.bir
    hor_split_data_frames.append(hor_train.iloc[test_index])


# Next, we go through the split data frames, add the predictions, and reduce to the re
for counter in range(len(hor_split_data_frames)):
    temp_df = hor_split_data_frames[counter]
    temp_df = temp_df.assign(pred_bs=hor_no_ump_log_reg_preds[counter])
    temp_df = temp_df[['true_ball/strike', 'pred_bs', 'binary_bs', 'correct_call']]
    hor_split_data_frames[counter] = temp_df



# We again go through the split data frames, check whether the predictions were correc
# and store that information
for counter in range(len(hor_split_data_frames)):
    correct_preds = []
    temp_df = hor_split_data_frames[counter]
    for ind in temp_df.index:
        pred = temp_df.at[ind, 'pred_bs']
        ump_call = temp_df.at[ind, 'binary_bs']
        cor_incorr = temp_df.at[ind, 'correct_call']
        if pred == ump_call:
            if cor_incorr == 'correct_call':
                correct_preds.append('correct_pred')
            else:
                correct_preds.append('incorrect_pred')
        else:
            if cor_incorr == 'correct_call':
                correct_preds.append('incorrect_pred')
            else:
                correct_preds.append('correct_pred')
    temp_df = temp_df.assign(correct_pred=correct_preds)
    hor_split_data_frames[counter] = temp_df

# Finally, we make a data frame comparing the umpire calls to the predicted calls
correct_call_test_list = []
```

```
correct_pred_test_list = []

for counter in range(len(hor_split_data_frames)):
    correct_call_test_list.append(hor_split_data_frames[counter].correct_call.value_co
    correct_pred_test_list.append(hor_split_data_frames[counter].correct_pred.value_co

hor_no_ump_log_reg_call_pred_stats = pd.DataFrame(data={
        'hor_no_ump_correct_call_percent':correct_call_test_list,
        'hor_no_ump_correct_pred_percent':correct_pred_test_list
        }
    ).describe()

hor_no_ump_log_reg_call_pred_stats
```

Out[37]:

	hor_no_ump_correct_call_percent	hor_no_ump_correct_pred_percent
count	10.000000	10.000000
mean	0.951358	0.963241
std	0.001892	0.001976
min	0.948128	0.960102
25%	0.950635	0.962239
50%	0.951540	0.963292
75%	0.952390	0.964796
max	0.954222	0.965760

Despite being trained on umpire's calls, we see that the `LogisiticRegression` model predicts balls and strikes more accurately than the umpires call balls and strikes.

Additionally, we note that the umpires correct call percentage in the `horizontal` data frame (95.1%) is higher than in the `core` data frame (91.7%). Recall that we obtained the `horizontal` data frame from the `core` data frame by dropping pitches with exactly one of their `plate_x` values and `plate_z` values within the regulation strike zone, which explains why the correct call percentage is noticeably higher.

In the next subsection, we will create a data frame that does not throw out these pitches and tackles the problem associated to a varying strike zone.

2.4 - Investigating Vertical Location

Recall that the height of the regulation strike zone (and thus whether a pitch is truly a ball or strike) depends on the height of the batter, whereas the width of the strike zone does not. As a result, the `plate_z` feature is the height of the pitch from the ground and is not a suitable feature for our purposes.

One approach would be to mimic the `plate_x_mag` feature by computing the mean of the middle of the regulation strike zone and recording the distance of the pitch from this mean.

However, the distance from the middle of the regulation strike zone to the edge varies as well.

Instead, we use the breakdown of the regulation zone into the heart, shadow, chase, and waste regions for each pitch (see early data analysis above for a visual description), which are based on percentiles, rather than absolute measurements. The shadow region of the zone is the only region which incorporates both balls and strikes, so we split it into two subregions.

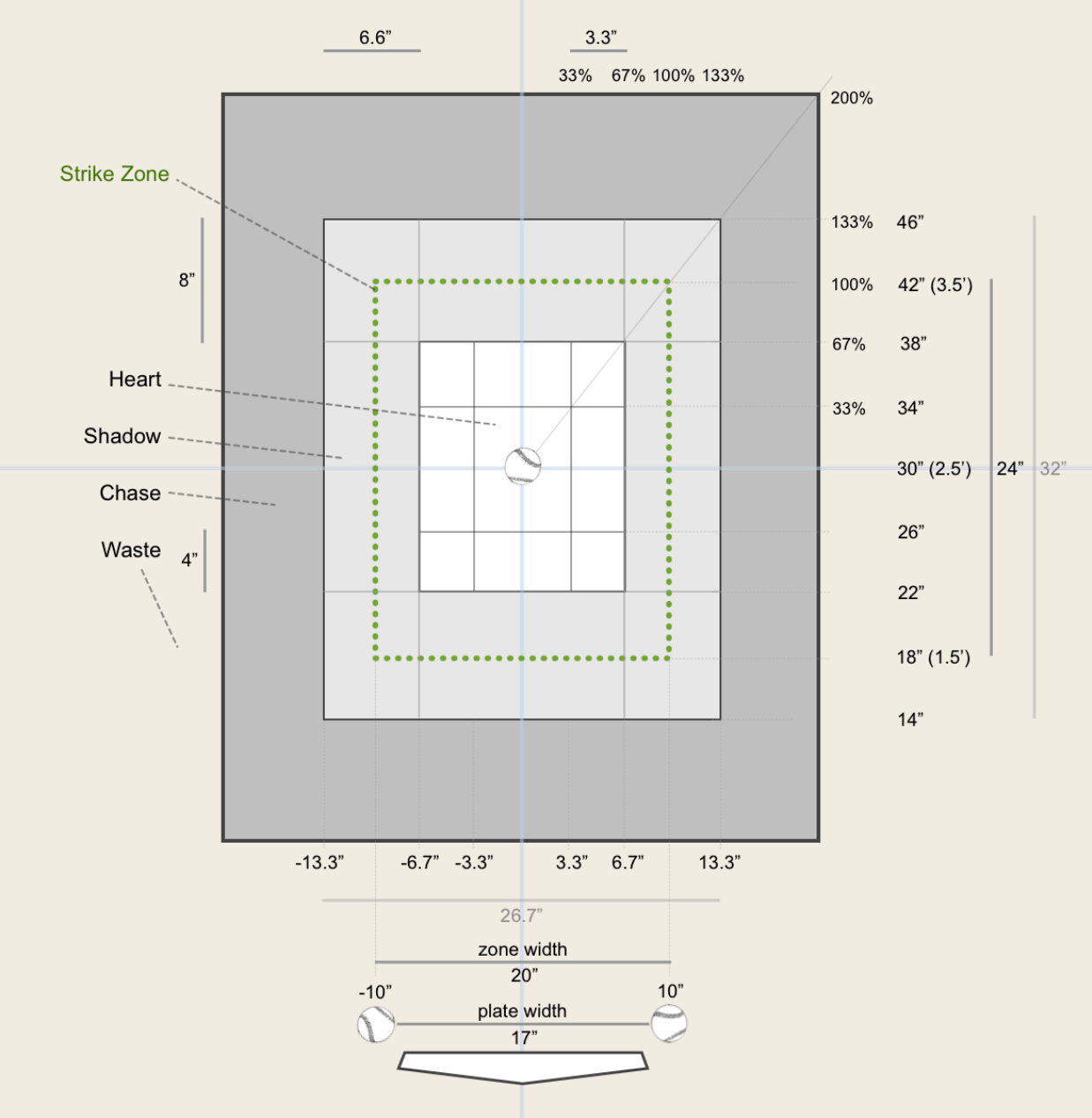
2.5 - Modeling with Added Vertical Location: Predicting Called Balls and Strikes

Instead of considering `plate_z` as a continuous variable, we will determine if the height of the pitch is in the vertical heart, vertical shadow, vertical chase, or vertical waste portion of the zone. However, the shadow zone incorporates both balls and strikes. For this purpose, we will split the shadow zone into two parts.

We begin by recalling the heart / shadow / chase / waste breakdown:

In [38]:
```
Image("heart_shadow_chase_waste.png")
```

Out[38]:



Now, we create the feature `plate_z_hscw` to record whether the pitch was in the heart portion of the zone, a strike in the shadow portion of the zone, a ball in the shadow portion of the zone, a ball in the chase portion of the zone, or a ball in the waste portion of the zone.

In [39]:

```
vert_cat_list = []

for counter in core.index:
    z = core.at[counter, 'plate_z']
    top = core.at[counter, 'sz_top']
    bot = core.at[counter, 'sz_bot']
    mid = (top+bot)/2
    unit = top-mid
    heart_adj = (2/3)*unit
    shadow_adj = (4/3)*unit
    chase_adj = 2*unit
    if (mid-heart_adj <= z <= mid+heart_adj):
        vert_cat_list.append('vertical_heart')
    elif (mid-unit <= z <= mid+unit):
```

```
        vert_cat_list.append('vertical_shadow_strike')
    elif (mid-shadow_adj <= z <= mid+shadow_adj):
        vert_cat_list.append('vertical_shadow_ball')
    elif (mid-chase_adj <= z <= mid+chase_adj):
        vert_cat_list.append('vertical_chase')
    else:
        vert_cat_list.append('vertical_waste')

vertical = core.assign(plate_z_hscw=vert_cat_list)
```

Note that in the `horizontal` data frame, we restricted to pitches that were either 1) true strikes, or 2) both the horizontal and vertical pitch location are outside of the strike zone. Since we now have a categorical version of the `plate_z` feature, we do not need to use the pitch location restrictions used in creating the `horizontal` data frame for the `vertical` data frame.

2.5.1 - Train-Test Splits

We stratify by balls and strikes and verify the similar perecentages between the full data frame and the test data frame.

In [40]:
```
vert_train, vert_test = train_test_split(vertical,
                                         test_size=0.2,
                                         shuffle=True,
                                         random_state=530,
                                         stratify=vertical['ball/strike'])
```

In [41]:
```
vertical['ball/strike'].value_counts(normalize=True)
```

Out[41]:
```
B    0.671135
S    0.328865
Name: ball/strike, dtype: float64
```

In [42]:
```
vert_train['ball/strike'].value_counts(normalize=True)
```

Out[42]:
```
B    0.671136
S    0.328864
Name: ball/strike, dtype: float64
```

2.5.2 - Baseline Model for `Vertical` Data Frame

Given the difference in balls and strikes between the `horizontal` and `vertical` data frames, we will create a separate baseline model, now using a 33% chance of a random pitch being a strike.

In [43]:
```
vert_baseline_accs = []
vert_baseline_f1s = []
vert_baseline_praucs = []

np.random.seed(923)

for obs in range(1000):
```

```
    rand_draw = np.random.binomial(n=1, p=0.33, size=len(vert_train))
    vert_baseline_accs.append( accuracy_score(vert_train.binary_bs.values, rand_draw)
    precision, recall, _ = precision_recall_curve(vert_train.binary_bs.values, rand_dr
    vert_baseline_praucs.append( auc(recall, precision) )
    vert_baseline_f1s.append( f1_score(vert_train.binary_bs.values, rand_draw) )

vert_baseline_stats = pd.DataFrame(data={
    'vert_baseline_accuracy':vert_baseline_accs,
    'vert_baseline_f1':vert_baseline_f1s,
    'vert_baseline_pr_auc':vert_baseline_praucs
    }
)

vert_baseline_stats.describe()
```

Out[43]:

	vert_baseline_accuracy	vert_baseline_f1	vert_baseline_pr_auc
count	1000.000000	1000.000000	1000.000000
mean	0.558159	0.329416	0.439587
std	0.000877	0.001326	0.001085
min	0.555453	0.325689	0.436540
25%	0.557592	0.328507	0.438845
50%	0.558159	0.329420	0.439586
75%	0.558748	0.330347	0.440346
max	0.561238	0.334000	0.443423

2.5.3 - Logistic Regression - Vertical, No Umpires

We begin by setting up our cross-validation.

In [44]:
```
kfold_splits = 10

vert_no_ump_log_reg_kfold_rand_state = 412

vert_no_ump_log_reg_kfold = StratifiedKFold(kfold_splits, shuffle=True, random_state=v
```

Next, we again use `GridSearchCV` to determine our penalty and maximum number of iterations.

In [45]:
```
#vert_no_ump_log_reg_feat = ['plate_x_mag', 'plate_z_hscw']

#vert_no_ump_log_reg_preprocessor = ColumnTransformer(
#    transformers=[
#            ('categorical', OneHotEncoder(), ['plate_z_hscw']),
#            ('continuous', StandardScaler(), ['plate_x_mag'])
#        ],
#    remainder='passthrough')

## Setting up the normalization pipeline
#vert_no_ump_log_reg_pipeline = Pipeline([
```

```
#    ('log_reg_preprocessor', vert_no_ump_log_reg_preprocessor),
#    ('log_reg', LogisticRegression())
#])

## Creating the grid
#vert_no_ump_log_reg_grid = GridSearchCV(
#    vert_no_ump_log_reg_pipeline,
#    param_grid={
#        'log_reg__penalty':['l2', None],
#        'log_reg__max_iter':[1000, 5000, 10000, 50000, 100000, 500000, 1000000, 50000
#    },
#    scoring='accuracy',
#    cv=5
#)

## Fitting the grid
#vert_no_ump_log_reg_grid.fit(vert_train[vert_no_ump_log_reg_feat], vert_train.binary_
```

In [46]:
```
#vert_no_ump_log_reg_grid_results = pd.DataFrame(data=vert_no_ump_log_reg_grid.cv_resu

#vert_no_ump_log_reg_grid_results_order = ['rank_test_score', 'std_test_score', 'param
#vert_no_ump_log_reg_grid_results_order.extend(['mean_fit_time', 'std_fit_time', 'mean
#vert_no_ump_log_reg_grid_results_order.extend(['split1_test_score', 'split2_test_scor

#vert_no_ump_log_reg_grid_results = vert_no_ump_log_reg_grid_results[vert_no_ump_log_r

#vert_no_ump_log_reg_grid_results

#vert_no_ump_log_reg_grid_results.to_csv('vert_no_ump_log_reg_grid_results.csv', index
```

In [47]:
```
#vert_no_ump_log_reg_grid_results_csv = pd.read_csv('vert_no_ump_log_reg_grid_results.

#vert_no_ump_log_reg_grid_results_csv
```

As can be verified above, there are several options which seem to provide the best test score. Given the similarity in scores, we pick 1000 iterations and no penalty for this model.

In [48]:
```
vert_no_ump_log_reg_preds = []
vert_no_ump_log_reg_pred_strikes = []

vert_no_ump_log_reg_accs = []
vert_no_ump_log_reg_f1s = []
vert_no_ump_log_reg_praucs = []


vert_no_ump_log_reg_feat = ['plate_x_mag', 'plate_z_hscw']

vert_no_ump_log_reg_preprocessor = ColumnTransformer(
    transformers=[
        ('categorical', OneHotEncoder(), ['plate_z_hscw']),
        ('continuous', StandardScaler(), ['plate_x_mag'])
    ],
    remainder='passthrough'
)


for train_index, test_index in vert_no_ump_log_reg_kfold.split(vert_train, vert_train.
    # Splitting up our data using the indices from our kfold split
```

```
    split_train = vert_train.iloc[train_index]
    split_test = vert_train.iloc[test_index]

    # Setting up the normalization pipeline
    vert_no_ump_log_reg_pipeline = Pipeline([
        ('log_reg_preprocessor', vert_no_ump_log_reg_preprocessor),
        ('log_reg', LogisticRegression(penalty=None, max_iter=1000))
    ])

    # Fitting the pipeline
    vert_no_ump_log_reg_pipeline.fit(split_train[vert_no_ump_log_reg_feat], split_trai

    # Predictions
    split_pred = vert_no_ump_log_reg_pipeline.predict(split_test[vert_no_ump_log_reg_f
    vert_no_ump_log_reg_preds.append(split_pred)
    split_pred_dict = {1:0}
    for entry in split_pred:
        if entry==1:
            split_pred_dict[1] += 1
    vert_no_ump_log_reg_pred_strikes.append(split_pred_dict[1])

    # Evaluation metrics
    vert_no_ump_log_reg_accs.append( accuracy_score(split_test.binary_bs, split_pred)
    vert_no_ump_log_reg_f1s.append( f1_score(split_test.binary_bs, split_pred) )
    vert_no_ump_log_reg_prec, vert_no_ump_log_reg_rec, _ = precision_recall_curve(spli
    vert_no_ump_log_reg_praucs.append(auc(vert_no_ump_log_reg_rec, vert_no_ump_log_reg

vert_no_ump_log_reg_stats = pd.DataFrame(data={
    'vert_no_ump_pred_strikes':vert_no_ump_log_reg_pred_strikes,
    'vert_no_ump_log_reg_accuracy':vert_no_ump_log_reg_accs,
    'vert_no_ump_log_reg_f1':vert_no_ump_log_reg_f1s,
    'vert_no_ump_log_reg_pr_auc':vert_no_ump_log_reg_praucs
    }
)

vert_no_ump_log_reg_stats.describe()
```

Out[48]:

	vert_no_ump_pred_strikes	vert_no_ump_log_reg_accuracy	vert_no_ump_log_reg_f1	vert_no_ump_
count	10.000000	10.000000	10.000000	
mean	9390.100000	0.914877	0.870294	
std	58.833947	0.002244	0.003510	
min	9263.000000	0.911186	0.864618	
25%	9365.250000	0.914328	0.868710	
50%	9405.000000	0.914917	0.870549	
75%	9420.000000	0.916436	0.872875	
max	9474.000000	0.917535	0.874436	

We now compare these metrics with the baseline model.

In [49]:
```
vert_no_ump_log_reg_combo_stats = pd.concat([vert_baseline_stats.describe(), vert_no_u
```

```
vert_no_ump_log_reg_combo_stats_order = ['vert_baseline_accuracy', 'vert_no_ump_log_re
vert_no_ump_log_reg_combo_stats_order.extend(['vert_no_ump_log_reg_f1', 'vert_baseline
vert_no_ump_log_reg_combo_stats = vert_no_ump_log_reg_combo_stats[vert_no_ump_log_reg_

vert_no_ump_log_reg_combo_stats
```

Out[49]:

	vert_baseline_accuracy	vert_no_ump_log_reg_accuracy	vert_baseline_f1	vert_no_ump_log_reg_f1
count	1000.000000	10.000000	1000.000000	10.000000
mean	0.558159	0.914877	0.329416	0.870294
std	0.000877	0.002244	0.001326	0.003510
min	0.555453	0.911186	0.325689	0.864618
25%	0.557592	0.914328	0.328507	0.868710
50%	0.558159	0.914917	0.329420	0.870549
75%	0.558748	0.916436	0.330347	0.872875
max	0.561238	0.917535	0.334000	0.874436

Recall that in the `horizontal` data frame, we restricted to pitches that were either 1) true strikes, or 2) both the horizontal and vertical pitch location are outside of the strike zone.

In this model, we are using the `vertical` data frame which includes all pitches, as we now have categorical vertical pitch location data. Consequently, it is natural to see both

- the increase in accuracy of the baseline model, as strike percentage has dropped from 0.44 to 0.33, and
- the slight decrease in mean accuracy from logistic regression model on the `horizontal` data frame to the logistic regression model on the `vertical` data frame.

2.5.4 - Support Vector Machines - Vertical, No Umpires

We start by preparing for cross validation for this model.

In [50]:
```
kfold_splits = 10

vert_no_ump_svm_kfold_rand_state = 462

vert_no_ump_svm_kfold = StratifiedKFold(kfold_splits, shuffle=True, random_state=vert_
```

For `LinearSVC`, we have used `GridSearchCV` for tuning the hyperparameter `C` and considering the number of max iterations. This will also use cross validation on `core_train`.

Given the run time, we have commented out the code below and saved the results as a CSV, which we import for verification. We pick `C=0.0001` and 500,000 iterations.

In [51]:
```
#vert_no_ump_svm_feat = ['plate_x_mag', 'plate_z_hscw']
```

```
#vert_no_ump_svm_preprocessor = ColumnTransformer(
#    transformers=[
#            ('categorical', OneHotEncoder(), ['plate_z_hscw']),
#            ('continuous', StandardScaler(), ['plate_x_mag'])
#        ],
#    remainder='passthrough')

## Setting up the normalization pipeline
#vert_no_ump_svm_pipeline = Pipeline([
#    ('svm_preprocessor', vert_no_ump_svm_preprocessor),
#    ('svm', LinearSVC())
#])

## Creating the grid
#vert_no_ump_svm_grid = GridSearchCV(
#    vert_no_ump_svm_pipeline,
#    param_grid={
#        'svm__C':[0.00001, 0.0001, 0.001, 0.01, 0.1],
#        'svm__max_iter':[1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000]
#    },
#    scoring='accuracy',
#    cv=5
#)

## Fitting the grid
#vert_no_ump_svm_grid.fit(vert_train[vert_no_ump_svm_feat], vert_train.binary_bs)
```

In [52]:
```
#vert_no_ump_svm_grid_results = pd.DataFrame(data=vert_no_ump_svm_grid.cv_results_)

#vert_no_ump_svm_grid_results_order = ['rank_test_score', 'std_test_score', 'param_svm
#vert_no_ump_svm_grid_results_order.extend(['mean_fit_time', 'std_fit_time', 'mean_sco
#vert_no_ump_svm_grid_results_order.extend(['split1_test_score', 'split2_test_score',

#vert_no_ump_svm_grid_results = vert_no_ump_svm_grid_results[vert_no_ump_svm_grid_resu

#vert_no_ump_svm_grid_results

#vert_no_ump_svm_grid_results.to_csv('vert_no_ump_svm_grid_results.csv', index=False)
```

In [53]:
```
#vert_no_ump_svm_grid_results_csv = pd.read_csv('vert_no_ump_svm_grid_results.csv')

#vert_no_ump_svm_grid_results_csv
```

We now run the model with the chosen hyperparameters.

In [54]:
```
vert_no_ump_svm_preds = []
vert_no_ump_svm_pred_strikes = []

vert_no_ump_svm_accs = []
vert_no_ump_svm_f1s = []
vert_no_ump_svm_praucs = []

vert_no_ump_svm_feat = ['plate_x_mag', 'plate_z_hscw']

vert_no_ump_svm_preprocessor = ColumnTransformer(
    transformers=[
        ('categorical', OneHotEncoder(), ['plate_z_hscw']),
        ('continuous', StandardScaler(), ['plate_x_mag'])
```

```
    ],
    remainder='passthrough'
)


for train_index, test_index in vert_no_ump_svm_kfold.split(vert_train, vert_train.bina
    # Splitting up our data using the indices from our kfold split
    split_train = vert_train.iloc[train_index]
    split_test = vert_train.iloc[test_index]

    # Setting up the normalization pipeline
    vert_no_ump_svm_pipeline = Pipeline([
        ('svm_preprocessor', vert_no_ump_svm_preprocessor),
        ('svm', LinearSVC(C=0.0001, max_iter=500000))
    ])

    # Fitting the pipeline
    vert_no_ump_svm_pipeline.fit(split_train[vert_no_ump_svm_feat], split_train.binary

    # Predictions
    split_pred = vert_no_ump_svm_pipeline.predict(split_test[vert_no_ump_svm_feat])
    vert_no_ump_svm_preds.append(split_pred)
    split_pred_dict = {1:0}
    for entry in split_pred:
        if entry==1:
            split_pred_dict[1] += 1
    vert_no_ump_svm_pred_strikes.append(split_pred_dict[1])

    # Evaluation metrics
    vert_no_ump_svm_accs.append( accuracy_score(split_test.binary_bs, split_pred) )
    vert_no_ump_svm_f1s.append( f1_score(split_test.binary_bs, split_pred) )
    vert_no_ump_svm_prec, vert_no_ump_svm_rec, _ = precision_recall_curve(split_test.b
    vert_no_ump_svm_praucs.append(auc(vert_no_ump_svm_rec, vert_no_ump_svm_prec))

vert_no_ump_svm_stats = pd.DataFrame(data={
    'vert_no_ump_pred_strikes':vert_no_ump_svm_pred_strikes,
    'vert_no_ump_svm_accuracy':vert_no_ump_svm_accs,
    'vert_no_ump_svm_f1':vert_no_ump_svm_f1s,
    'vert_no_ump_svm_pr_auc':vert_no_ump_svm_praucs
    }
)

vert_no_ump_svm_stats.describe()
```

Out[54]:

	vert_no_ump_pred_strikes	vert_no_ump_svm_accuracy	vert_no_ump_svm_f1	vert_no_ump_svm_pr
count	10.000000	10.000000	10.000000	10.00
mean	9121.200000	0.913674	0.866559	0.89
std	48.271915	0.001923	0.002860	0.00
min	9057.000000	0.911573	0.863432	0.88
25%	9089.250000	0.912312	0.864563	0.88
50%	9115.000000	0.913052	0.865482	0.89
75%	9155.500000	0.914430	0.867666	0.89
max	9197.000000	0.917393	0.871939	0.89

As always, we compare the metrics for this model with the relevant baseline model.

In [55]:
```
vert_no_ump_svm_combo_stats = pd.concat([vert_baseline_stats.describe(), vert_no_ump_s

vert_no_ump_svm_combo_stats_order = ['vert_baseline_accuracy', 'vert_no_ump_svm_accura
vert_no_ump_svm_combo_stats_order.extend(['vert_no_ump_svm_f1', 'vert_baseline_pr_auc'
vert_no_ump_svm_combo_stats = vert_no_ump_svm_combo_stats[vert_no_ump_svm_combo_stats_

vert_no_ump_svm_combo_stats
```

Out[55]:

	vert_baseline_accuracy	vert_no_ump_svm_accuracy	vert_baseline_f1	vert_no_ump_svm_f1	vert_b
count	1000.000000	10.000000	1000.000000	10.000000	
mean	0.558159	0.913674	0.329416	0.866559	
std	0.000877	0.001923	0.001326	0.002860	
min	0.555453	0.911573	0.325689	0.863432	
25%	0.557592	0.912312	0.328507	0.864563	
50%	0.558159	0.913052	0.329420	0.865482	
75%	0.558748	0.914430	0.330347	0.867666	
max	0.561238	0.917393	0.334000	0.871939	

We see similar performance between the `LinearSVC` model and the `LogisticRegression` model compared to the `vertical` baseline.

2.6 - Modeling with Added Vertical Location: Comparing with True Balls and Strikes

Just as in `Section 2.3`, we compare the correct call percentage with the correct prediction percentage.

In [56]:

```
# Note that this requires vert_no_ump_log_reg_kfold, which is created and run in Secti
# The code below requires that code to be run first anyway, so there should be no issu
# We reference the call in 2.5.2 so that the random states are always the same

vert_split_data_frames = []

# This fills the list created above with the split data frames
for train_index, test_index in vert_no_ump_log_reg_kfold.split(vert_train, vert_train.
    vert_split_data_frames.append(vert_train.iloc[test_index])


# Next, we go through the split data frames, add the predictions, and reduce to the re
for counter in range(len(vert_split_data_frames)):
    temp_df = vert_split_data_frames[counter]
    temp_df = temp_df.assign(pred_bs=vert_no_ump_log_reg_preds[counter])
    temp_df = temp_df[['true_ball/strike', 'pred_bs', 'binary_bs', 'correct_call']]
    vert_split_data_frames[counter] = temp_df

# We again go through the split data frames, check whether the predictions were correc
# and store that information
for counter in range(len(vert_split_data_frames)):
    correct_preds = []
    temp_df = vert_split_data_frames[counter]
    for ind in temp_df.index:
        pred = temp_df.at[ind, 'pred_bs']
        ump_call = temp_df.at[ind, 'binary_bs']
        cor_incorr = temp_df.at[ind, 'correct_call']
        if pred == ump_call:
            if cor_incorr == 'correct_call':
                correct_preds.append('correct_pred')
            else:
                correct_preds.append('incorrect_pred')
        else:
            if cor_incorr == 'correct_call':
                correct_preds.append('incorrect_pred')
            else:
                correct_preds.append('correct_pred')
    temp_df = temp_df.assign(correct_pred=correct_preds)
    vert_split_data_frames[counter] = temp_df

# Finally, we make a data frame comparing the umpire calls to the predicted calls
correct_call_test_list = []
correct_pred_test_list = []

for counter in range(len(vert_split_data_frames)):
    correct_call_test_list.append(vert_split_data_frames[counter].correct_call.value_c
    correct_pred_test_list.append(vert_split_data_frames[counter].correct_pred.value_c

vert_no_ump_log_reg_call_pred_stats = pd.DataFrame(data={
    'vert_no_ump_correct_call_percent':correct_call_test_list,
    'vert_no_ump_correct_pred_percent':correct_pred_test_list
    }
    ).describe()

vert_no_ump_log_reg_call_pred_stats
```

Out[56]:

	vert_no_ump_correct_call_percent	vert_no_ump_correct_pred_percent
count	10.000000	10.000000
mean	0.917635	0.947204
std	0.001877	0.001432
min	0.914990	0.945638
25%	0.916670	0.946100
50%	0.917168	0.946842
75%	0.918605	0.947880
max	0.921406	0.949578

We see that for the `vertical` data frame and the `plate_x_mag` feature, our `LogisticRegression` model correctly predicts balls and strikes at a better rate than umpires. Furthermore, the difference correct prediction percentage and correct call percentage for the `vertical` data frame (about 3%) is larger than for the `horizontal` data frame (about 1.2%).

2.7 - Modeling with Horizontal Location and Umpires: Predicting Called Balls and Strikes

In `Section 2.2`, the models only used the feature `plate_x_mag` with the horizontal data frame. We will now re-create all of the models in `Section 2.2` with `umpire` as an additional feature.

2.7.1 - Train-Test Splits - Horizontal with Umpires

Now that we will be including the `umpire` feature, we will create new training and test datasets so that we can stratify by this feature.

In [57]:
```
hor_ump_train, hor_ump_test = train_test_split(horizontal,
                                               test_size=0.2,
                                               shuffle=True,
                                               random_state=543,
                                               stratify=horizontal['umpire'])
```

Note that despite stratifying by `umpire`, we have very similar proportions of balls and strikes in our training data

In [58]:
```
horizontal['ball/strike'].value_counts(normalize=True)
```

Out[58]:
```
S    0.560544
B    0.439456
Name: ball/strike, dtype: float64
```

In [59]:
```
hor_ump_train['ball/strike'].value_counts(normalize=True)
```

Out[59]:
```
S    0.56021
B    0.43979
Name: ball/strike, dtype: float64
```

2.7.2 - Logistic Regression - Horizontal with Umpires

We start by preparing our cross validation.

In [60]:
```
kfold_splits = 10

hor_ump_log_reg_kfold_rand_state = 620

hor_ump_log_reg_kfold = StratifiedKFold(kfold_splits, shuffle=True, random_state=hor_u
```

Once again, we use `GridSearchCV` to determine our penalty and maximum number of iterations for our logistic regression. We will use no penalty and a maximum of 1000 iterations.

In [61]:
```
#hor_ump_log_reg_feat = ['plate_x_mag', 'umpire']

#hor_ump_log_reg_preprocessor = ColumnTransformer(
#    transformers=[
#            ('categorical', OneHotEncoder(), ['umpire']),
#            ('continuous', StandardScaler(), ['plate_x_mag'])
#        ],
#    remainder='passthrough')

## Setting up the normalization pipeline
#hor_ump_log_reg_pipeline = Pipeline([
#    ('log_reg_preprocessor', hor_ump_log_reg_preprocessor),
#    ('log_reg', LogisticRegression())
#])

## Creating the grid
#hor_ump_log_reg_grid = GridSearchCV(
#    hor_ump_log_reg_pipeline,
#    param_grid={
#        'log_reg__penalty':['l2', None],
#        'log_reg__max_iter':[100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000
#    },
#    scoring='accuracy',
#    cv=5
#)

## Fitting the grid
#hor_ump_log_reg_grid.fit(hor_ump_train[hor_ump_log_reg_feat], hor_train.binary_bs)
```

In [62]:
```
#hor_ump_log_reg_grid_results = pd.DataFrame(data=hor_ump_log_reg_grid.cv_results_)

#hor_ump_log_reg_grid_results_order = ['rank_test_score', 'std_test_score', 'param_log
#hor_ump_log_reg_grid_results_order.extend(['param_log_reg__max_iter', 'mean_fit_time'
#hor_ump_log_reg_grid_results_order.extend(['std_score_time', 'split0_test_score', 'sp
#hor_ump_log_reg_grid_results_order.extend(['split2_test_score', 'split3_test_score',

#hor_ump_log_reg_grid_results = hor_ump_log_reg_grid_results[hor_ump_log_reg_grid_resu

#hor_ump_log_reg_grid_results.to_csv('hor_ump_log_reg_grid_results.csv', index=False)
```

In [63]:
```
#hor_ump_log_reg_grid_results_csv = pd.read_csv('hor_ump_log_reg_grid_results.csv')

#hor_ump_log_reg_grid_results_csv
```

We now run the model with our chosen hyperparameters.

In [64]:
```
hor_ump_log_reg_accs = []
hor_ump_log_reg_f1s = []
hor_ump_log_reg_praucs = []

hor_ump_log_reg_preds = []
hor_ump_log_reg_pred_strikes = []

hor_ump_log_reg_feat = ['plate_x_mag', 'umpire']

hor_ump_log_reg_preprocessor = ColumnTransformer(
    transformers=[
        ('categorical', OneHotEncoder(), ['umpire']),
        ('continuous', StandardScaler(), ['plate_x_mag'])
    ],
    remainder='passthrough'
)


for train_index, test_index in hor_ump_log_reg_kfold.split(hor_ump_train, hor_ump_trai
    # Splitting up our data using the indices from our kfold split
    split_train = hor_ump_train.iloc[train_index]
    split_test = hor_ump_train.iloc[test_index]

    # Setting up the normalization pipeline
    hor_ump_log_reg_pipeline = Pipeline([
        ('log_reg_preprocessor', hor_ump_log_reg_preprocessor),
        ('log_reg', LogisticRegression(penalty=None, max_iter=1000))
    ])

    # Fitting the pipeline
    hor_ump_log_reg_pipeline.fit(split_train[hor_ump_log_reg_feat], split_train.binary

    # Predictions
    split_pred = hor_ump_log_reg_pipeline.predict(split_test[hor_ump_log_reg_feat])
    hor_ump_log_reg_preds.append(split_pred)
    split_pred_dict = {1:0}
    for entry in split_pred:
        if entry==1:
            split_pred_dict[1] += 1
    hor_ump_log_reg_pred_strikes.append(split_pred_dict[1])
```

```
    # Evaluation metrics
    hor_ump_log_reg_accs.append( accuracy_score(split_test.binary_bs, split_pred) )
    hor_ump_log_reg_f1s.append( f1_score(split_test.binary_bs, split_pred) )
    hor_ump_log_reg_prec, hor_ump_log_reg_rec, _ = precision_recall_curve(split_test.b
    hor_ump_log_reg_praucs.append(auc(hor_ump_log_reg_rec, hor_ump_log_reg_prec))

hor_ump_log_reg_stats = pd.DataFrame(data={
    'hor_ump_pred_strikes':hor_ump_log_reg_pred_strikes,
    'hor_ump_log_reg_accuracy':hor_ump_log_reg_accs,
    'hor_ump_log_reg_f1':hor_ump_log_reg_f1s,
    'hor_ump_log_reg_pr_auc':hor_ump_log_reg_praucs
    })

hor_ump_log_reg_stats.describe()
```

Out[64]:

	hor_ump_pred_strikes	hor_ump_log_reg_accuracy	hor_ump_log_reg_f1	hor_ump_log_reg_pr_auc
count	10.000000	10.000000	10.000000	10.000000
mean	7854.400000	0.939068	0.946080	0.958985
std	32.792953	0.001768	0.001533	0.001334
min	7792.000000	0.936085	0.943479	0.956943
25%	7836.750000	0.938479	0.945674	0.958012
50%	7862.000000	0.939062	0.946136	0.958738
75%	7875.250000	0.940548	0.947248	0.960318
max	7904.000000	0.941164	0.947949	0.960580

We compare these metrics with the baseline model for the `horizontal` data frame.

In [65]:
```
hor_ump_log_reg_combo_stats = pd.concat([hor_baseline_stats.describe(), hor_ump_log_re

hor_ump_log_reg_combo_stats_order = ['hor_baseline_accuracy', 'hor_ump_log_reg_accurac
hor_ump_log_reg_combo_stats_order.extend(['hor_ump_log_reg_f1', 'hor_baseline_pr_auc',
hor_ump_log_reg_combo_stats = hor_ump_log_reg_combo_stats[hor_ump_log_reg_combo_stats_

hor_ump_log_reg_combo_stats
```

Out[65]:

	hor_baseline_accuracy	hor_ump_log_reg_accuracy	hor_baseline_f1	hor_ump_log_reg_f1	hor_base
count	1000.000000	10.000000	1000.000000	10.000000	
mean	0.492673	0.939068	0.492914	0.946080	
std	0.001333	0.001768	0.001590	0.001533	
min	0.488969	0.936085	0.488043	0.943479	
25%	0.491775	0.938479	0.491841	0.945674	
50%	0.492705	0.939062	0.492906	0.946136	
75%	0.493562	0.940548	0.493968	0.947248	
max	0.496775	0.941164	0.497711	0.947949	

We see greatly increased accuracy over the `horizontal` baseline, but very little improvement over only using the `plate_x_mag` feature (with respective mean accuracies of 93.9% and 93.8%).

2.7.3 - Support Vector Machines - Horizontal with Umpires

We begin by preparing the cross validation.

In [66]:
```
kfold_splits = 10

hor_ump_svm_kfold_rand_state = 622

hor_ump_svm_kfold = StratifiedKFold(kfold_splits, shuffle=True, random_state=hor_ump_s
```

We use `GridSearchCV` to determine our parameters. We will set `C=0.0001` and a maximum of 5000 iterations.

In [67]:
```
#hor_ump_svm_feat = ['plate_x_mag', 'umpire']

#hor_ump_svm_preprocessor = ColumnTransformer(
#    transformers=[
#            ('categorical', OneHotEncoder(), ['umpire']),
#            ('continuous', StandardScaler(), ['plate_x_mag'])
#        ],
#    remainder='passthrough')

## Setting up the normalization pipeline
#hor_ump_svm_pipeline = Pipeline([
#    ('svm_preprocessor', hor_ump_svm_preprocessor),
#    ('svm', LinearSVC())
#])

## Creating the grid
#hor_ump_svm_grid = GridSearchCV(
#    hor_ump_svm_pipeline,
#    param_grid={
#        'svm__C':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
```

```
#         'svm__max_iter':[1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000]
#     },
#     scoring='accuracy',
#     cv=5
#)

## Fitting the grid
#hor_ump_svm_grid.fit(hor_ump_train[hor_ump_svm_feat], hor_ump_train.binary_bs)
```

In [68]:
```
#hor_ump_svm_grid_results = pd.DataFrame(data=hor_ump_svm_grid.cv_results_)

#hor_ump_svm_grid_results_order = ['rank_test_score', 'std_test_score', 'param_svm__C'
#hor_ump_svm_grid_results_order.extend(['mean_fit_time', 'std_fit_time', 'mean_score_t
#hor_ump_svm_grid_results_order.extend(['split1_test_score', 'split2_test_score', 'spl

#hor_ump_svm_grid_results = hor_ump_svm_grid_results[hor_ump_svm_grid_results_order].s

#hor_ump_svm_grid_results.to_csv('hor_ump_svm_grid_results.csv', index=False)
```

In [69]:
```
#hor_ump_svm_grid_results_csv = pd.read_csv('hor_ump_svm_grid_results.csv')

#hor_ump_svm_grid_results_csv
```

We now run our model with the chosen hyperparameters.

In [70]:
```
hor_ump_svm_accs = []
hor_ump_svm_f1s = []
hor_ump_svm_praucs = []
hor_ump_svm_pred_strikes = []

hor_ump_svm_feat = ['plate_x_mag', 'umpire']

hor_ump_svm_preprocessor = ColumnTransformer(
    transformers=[
        ('categorical', OneHotEncoder(), ['umpire']),
        ('continuous', StandardScaler(), ['plate_x_mag'])
        ],
    remainder='passthrough')


for train_index, test_index in hor_ump_svm_kfold.split(hor_ump_train, hor_ump_train.bi
    # Splitting up our data using the indices from our kfold split
    split_train = hor_train.iloc[train_index]
    split_test = hor_train.iloc[test_index]

    # Setting up the normalization pipeline
    hor_ump_svm_pipeline = Pipeline([
        ('svm_preprocessor', hor_ump_svm_preprocessor),
        ('svm', LinearSVC(C=0.0001, max_iter=5000))
    ])

    # Fitting the pipeline
    hor_ump_svm_pipeline.fit(split_train[hor_ump_svm_feat], split_train.binary_bs)

    # Predictions
    split_pred = hor_ump_svm_pipeline.predict(split_test[hor_ump_svm_feat])
    split_pred_dict = {1:0}
    for entry in split_pred:
```

```
        if entry==1:
            split_pred_dict[1] += 1
    hor_ump_svm_pred_strikes.append(split_pred_dict[1])

    # Evaluation metrics
    hor_ump_svm_accs.append( accuracy_score(split_test.binary_bs, split_pred) )
    hor_ump_svm_f1s.append( f1_score(split_test.binary_bs, split_pred) )
    hor_ump_svm_prec, hor_ump_svm_rec, _ = precision_recall_curve(split_test.binary_bs
    hor_ump_svm_praucs.append(auc(hor_ump_svm_rec, hor_ump_svm_prec))

hor_ump_svm_stats = pd.DataFrame(data={
    'hor_ump_svm_pred_strikes':hor_ump_svm_pred_strikes,
    'hor_ump_svm_accuracy':hor_ump_svm_accs,
    'hor_ump_svm_f1':hor_ump_svm_f1s,
    'hor_ump_svm_pr_auc':hor_ump_svm_praucs
    })

hor_ump_svm_stats.describe()
```

Out[70]:

	hor_ump_svm_pred_strikes	hor_ump_svm_accuracy	hor_ump_svm_f1	hor_ump_svm_pr_auc
count	10.000000	10.000000	10.000000	10.000000
mean	8036.300000	0.945199	0.952070	0.960525
std	61.483602	0.001847	0.001800	0.001489
min	7941.000000	0.942615	0.949109	0.957481
25%	7984.500000	0.944230	0.951066	0.959872
50%	8061.000000	0.944936	0.952049	0.960657
75%	8073.000000	0.946169	0.952885	0.961348
max	8114.000000	0.948785	0.955339	0.962731

Finally, we compare the metrics for this model with the baseline model.

In [71]:

```
hor_ump_svm_combo_stats = pd.concat([hor_baseline_stats.describe(), hor_ump_svm_stats.

hor_ump_svm_combo_stats_order = ['hor_baseline_accuracy', 'hor_ump_svm_accuracy', 'hor
hor_ump_svm_combo_stats_order.extend(['hor_ump_svm_f1', 'hor_baseline_pr_auc', 'hor_um
hor_ump_svm_combo_stats = hor_ump_svm_combo_stats[hor_ump_svm_combo_stats_order]

hor_ump_svm_combo_stats
```

Out[71]:

	hor_baseline_accuracy	hor_ump_svm_accuracy	hor_baseline_f1	hor_ump_svm_f1	hor_baseline_p
count	1000.000000	10.000000	1000.000000	10.000000	1000.00
mean	0.492673	0.945199	0.492914	0.952070	0.65
std	0.001333	0.001847	0.001590	0.001800	0.00
min	0.488969	0.942615	0.488043	0.949109	0.65
25%	0.491775	0.944230	0.491841	0.951066	0.65
50%	0.492705	0.944936	0.492906	0.952049	0.65
75%	0.493562	0.946169	0.493968	0.952885	0.65
max	0.496775	0.948785	0.497711	0.955339	0.66

Similarly to the `LogisiticRegression` case, we see large improvements over the `horizontal` baseline and minimal improvements over the `LinearSVC` model using only `plate_x_mag`.

2.8 - Modeling with Horizontal Location and Umpires: Comparing with True Balls and Strikes

We proceed similarly to `Sections 2.3, 2.6` by comparing correct prediction percentages and correct call percentages.

In [72]:

```
# Note that this requires hor_ump_log_reg_kfold, which is created and run in Section 2
# The code below requires that code to be run first anyway, so there should be no issu
# We reference the call in 2.7.2 so that the random states are always the same

hor_ump_split_data_frames = []

# This fills the list created above with the split data frames
for train_index, test_index in hor_ump_log_reg_kfold.split(hor_ump_train, hor_ump_trai
    hor_ump_split_data_frames.append(hor_ump_train.iloc[test_index])


# Next, we go through the split data frames, add the predictions, and reduce to the re
for counter in range(len(hor_ump_split_data_frames)):
    temp_df = hor_ump_split_data_frames[counter]
    temp_df = temp_df.assign(pred_bs=hor_ump_log_reg_preds[counter])
    temp_df = temp_df[['true_ball/strike', 'pred_bs', 'binary_bs', 'correct_call']]
    hor_ump_split_data_frames[counter] = temp_df



# We again go through the split data frames, check whether the predictions were correc
# and store that information
for counter in range(len(hor_ump_split_data_frames)):
    correct_preds = []
    temp_df = hor_ump_split_data_frames[counter]
    for ind in temp_df.index:
        pred = temp_df.at[ind, 'pred_bs']
```

```
        ump_call = temp_df.at[ind, 'binary_bs']
        cor_incorr = temp_df.at[ind, 'correct_call']
        if pred == ump_call:
            if cor_incorr == 'correct_call':
                correct_preds.append('correct_pred')
            else:
                correct_preds.append('incorrect_pred')
        else:
            if cor_incorr == 'correct_call':
                correct_preds.append('incorrect_pred')
            else:
                correct_preds.append('correct_pred')
    temp_df = temp_df.assign(correct_pred=correct_preds)
    hor_ump_split_data_frames[counter] = temp_df

# Finally, we make a data frame comparing the umpire calls to the predicted calls
correct_call_test_list = []
correct_pred_test_list = []

for counter in range(len(hor_ump_split_data_frames)):
    correct_call_test_list.append(hor_ump_split_data_frames[counter].correct_call.valu
    correct_pred_test_list.append(hor_ump_split_data_frames[counter].correct_pred.valu

hor_ump_log_reg_call_pred_stats = pd.DataFrame(data={
    'hor_ump_correct_call_percent':correct_call_test_list,
    'hor_ump_correct_pred_percent':correct_pred_test_list
    }
).describe()

hor_ump_log_reg_call_pred_stats
```

Out[72]:

	hor_ump_correct_call_percent	hor_ump_correct_pred_percent
count	10.000000	10.000000
mean	0.951198	0.965497
std	0.001419	0.001514
min	0.949942	0.963508
25%	0.950233	0.964325
50%	0.951032	0.965432
75%	0.951465	0.966991
max	0.954806	0.967356

Similarly to `Subsection 2.3`, we have a correct prediction percentage higher than the correct call percentage. The difference when including the `umpire` feature is 1.4%, while the difference with only the `plate_x_mag` feature is 1.2%.

2.9 - Modeling with Added Vertical Location and Umpires: Predicting Called Balls and Strikes

Similarly to `Section 2.7`, we now re-create the models of `Section 2.5` using the features `plate_x_mag, plate_z_hscw, umpire` instead of only `plate_x_mag, plate_z_hscw`.

2.9.1 - Train-Test Splits - Vertical with Umpires

As in `Section 2.7`, we create new training and test data sets stratified by umpire.

In [73]:
```
vert_ump_train, vert_ump_test = train_test_split(vertical,
                                                 test_size=0.2,
                                                 shuffle=True,
                                                 random_state=546,
                                                 stratify=vertical['umpire'])
```

We again verify that we have similar percentages of balls and strikes in both the original data set and the training data set. Note, however, that there is a much larger percentage of balls present in the `vertical` data set because of the pitches that needed to be thrown out when constructing the `horizontal` data set; see the beginning of `Section 2.2` for more details.

In [74]:
```
vertical['ball/strike'].value_counts(normalize=True)
```

Out[74]:
```
B    0.671135
S    0.328865
Name: ball/strike, dtype: float64
```

In [75]:
```
vert_ump_train['ball/strike'].value_counts(normalize=True)
```

Out[75]:
```
B    0.671164
S    0.328836
Name: ball/strike, dtype: float64
```

2.9.2 - Logistic Regression - Vertical with Umpires

We prepare for cross validation.

In [76]:
```
kfold_splits = 10

vert_ump_log_reg_kfold_rand_state = 820

vert_ump_log_reg_kfold = StratifiedKFold(kfold_splits, shuffle=True, random_state=vert
```

After recording the results of using `GridSearchCV` for hyperparameter tuning, we will use no penalty and a maximum of 1000 iterations.

In [77]:
```
#vert_ump_log_reg_feat = ['plate_x_mag', 'plate_z_hscw', 'umpire']

#vert_ump_log_reg_preprocessor = ColumnTransformer(
#    transformers=[
#            ('categorical', OneHotEncoder(), ['umpire', 'plate_z_hscw']),
#            ('continuous', StandardScaler(), ['plate_x_mag'])
#        ],
#    remainder='passthrough')
```

```
## Setting up the normalization pipeline
#vert_ump_log_reg_pipeline = Pipeline([
#    ('log_reg_preprocessor', vert_ump_log_reg_preprocessor),
#    ('log_reg', LogisticRegression())
#])

## Creating the grid
#vert_ump_log_reg_grid = GridSearchCV(
#    vert_ump_log_reg_pipeline,
#    param_grid={
#        'log_reg__penalty':['l2', None],
#        'log_reg__max_iter':[100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000
#    },
#    scoring='accuracy',
#    cv=5
#)

## Fitting the grid
#vert_ump_log_reg_grid.fit(vert_ump_train[vert_ump_log_reg_feat], vert_train.binary_bs
```

In [78]:
```
#vert_ump_log_reg_grid_results = pd.DataFrame(data=vert_ump_log_reg_grid.cv_results_)

#vert_ump_log_reg_grid_results_order = ['rank_test_score', 'std_test_score', 'param_lo
#vert_ump_log_reg_grid_results_order.extend(['param_log_reg__max_iter', 'mean_fit_time
#vert_ump_log_reg_grid_results_order.extend(['std_score_time', 'split0_test_score', 's
#vert_ump_log_reg_grid_results_order.extend(['split2_test_score', 'split3_test_score',

#vert_ump_log_reg_grid_results = vert_ump_log_reg_grid_results[vert_ump_log_reg_grid_r

#vert_ump_log_reg_grid_results.to_csv('vert_ump_log_reg_grid_results.csv', index=False
```

In [79]:
```
#vert_ump_log_reg_grid_results_csv = pd.read_csv('vert_ump_log_reg_grid_results.csv')

#vert_ump_log_reg_grid_results_csv
```

We now run the model with our chosen hyperparameters.

In [80]:
```
vert_ump_log_reg_accs = []
vert_ump_log_reg_f1s = []
vert_ump_log_reg_praucs = []

vert_ump_log_reg_preds = []
vert_ump_log_reg_pred_strikes = []

vert_ump_log_reg_feat = ['plate_x_mag', 'plate_z_hscw', 'umpire']

vert_ump_log_reg_preprocessor = ColumnTransformer(
    transformers=[
        ('categorical', OneHotEncoder(), ['umpire', 'plate_z_hscw']),
        ('continuous', StandardScaler(), ['plate_x_mag'])
    ],
    remainder='passthrough'
)


for train_index, test_index in vert_ump_log_reg_kfold.split(vert_ump_train, vert_ump_t
    # Splitting up our data using the indices from our kfold split
    split_train = vert_ump_train.iloc[train_index]
```

```
    split_test = vert_ump_train.iloc[test_index]

    # Setting up the normalization pipeline
    vert_ump_log_reg_pipeline = Pipeline([
        ('log_reg_preprocessor', vert_ump_log_reg_preprocessor),
        ('log_reg', LogisticRegression(penalty=None, max_iter=1000))
    ])

    # Fitting the pipeline
    vert_ump_log_reg_pipeline.fit(split_train[vert_ump_log_reg_feat], split_train.bina

    # Predictions
    split_pred = vert_ump_log_reg_pipeline.predict(split_test[vert_ump_log_reg_feat])
    vert_ump_log_reg_preds.append(split_pred)
    split_pred_dict = {1:0}
    for entry in split_pred:
        if entry==1:
            split_pred_dict[1] += 1
    vert_ump_log_reg_pred_strikes.append(split_pred_dict[1])

    # Evaluation metrics
    vert_ump_log_reg_accs.append( accuracy_score(split_test.binary_bs, split_pred) )
    vert_ump_log_reg_f1s.append( f1_score(split_test.binary_bs, split_pred) )
    vert_ump_log_reg_prec, vert_ump_log_reg_rec, _ = precision_recall_curve(split_test
    vert_ump_log_reg_praucs.append(auc(vert_ump_log_reg_rec, vert_ump_log_reg_prec))

vert_ump_log_reg_stats = pd.DataFrame(data={
    'vert_ump_pred_strikes':vert_ump_log_reg_pred_strikes,
    'vert_ump_log_reg_accuracy':vert_ump_log_reg_accs,
    'vert_ump_log_reg_f1':vert_ump_log_reg_f1s,
    'vert_ump_log_reg_pr_auc':vert_ump_log_reg_praucs
    })

vert_ump_log_reg_stats.describe()
```

Out[80]:

	vert_ump_pred_strikes	vert_ump_log_reg_accuracy	vert_ump_log_reg_f1	vert_ump_log_reg_pr_au
count	10.000000	10.000000	10.000000	10.00000
mean	9410.100000	0.914884	0.870439	0.89190
std	49.912034	0.002290	0.003485	0.00291
min	9316.000000	0.911364	0.864945	0.88745
25%	9397.250000	0.914046	0.869351	0.89070
50%	9404.500000	0.915007	0.870597	0.89206
75%	9412.500000	0.916164	0.871895	0.89368
max	9516.000000	0.918335	0.875953	0.89619

We now compare the metrics for this model with the baseline model for the `vertical` data frame.

In [81]:
```
vert_ump_log_reg_combo_stats = pd.concat([vert_baseline_stats.describe(), vert_ump_log

vert_ump_log_reg_combo_stats_order = ['vert_baseline_accuracy', 'vert_ump_log_reg_accu
```

```
vert_ump_log_reg_combo_stats_order.extend(['vert_ump_log_reg_f1', 'vert_baseline_pr_au
vert_ump_log_reg_combo_stats = vert_ump_log_reg_combo_stats[vert_ump_log_reg_combo_sta

vert_ump_log_reg_combo_stats
```

Out[81]:

	vert_baseline_accuracy	vert_ump_log_reg_accuracy	vert_baseline_f1	vert_ump_log_reg_f1	vert_b
count	1000.000000	10.000000	1000.000000	10.000000	
mean	0.558159	0.914884	0.329416	0.870439	
std	0.000877	0.002290	0.001326	0.003485	
min	0.555453	0.911364	0.325689	0.864945	
25%	0.557592	0.914046	0.328507	0.869351	
50%	0.558159	0.915007	0.329420	0.870597	
75%	0.558748	0.916164	0.330347	0.871895	
max	0.561238	0.918335	0.334000	0.875953	

This model's performance is comparable to what we have seen previously.

2.9.3 - Support Vector Machines - Vertical with Umpires

We begin by preparing the cross validation.

In [82]:
```
kfold_splits = 10

vert_ump_svm_kfold_rand_state = 825

vert_ump_svm_kfold = StratifiedKFold(kfold_splits, shuffle=True, random_state=vert_ump
```

After observing the following hyperparameter tuning, we take `C=0.001` with a maximum of 5000 iterations.

In [83]:
```
#vert_ump_svm_feat = ['plate_x_mag', 'plate_z_hscw', 'umpire']

#vert_ump_svm_preprocessor = ColumnTransformer(
#    transformers=[
#            ('categorical', OneHotEncoder(), ['umpire', 'plate_z_hscw']),
#            ('continuous', StandardScaler(), ['plate_x_mag'])
#        ],
#    remainder='passthrough')

## Setting up the normalization pipeline
#vert_ump_svm_pipeline = Pipeline([
#    ('svm_preprocessor', vert_ump_svm_preprocessor),
#    ('svm', LinearSVC())
#])

## Creating the grid
#vert_ump_svm_grid = GridSearchCV(
#    vert_ump_svm_pipeline,
```

```
#    param_grid={
#        'svm__C':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
#        'svm__max_iter':[1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000]
#    },
#    scoring='accuracy',
#    cv=5
#)

## Fitting the grid
#vert_ump_svm_grid.fit(vert_ump_train[vert_ump_svm_feat], vert_ump_train.binary_bs)
```

In [84]:
```
#vert_ump_svm_grid_results = pd.DataFrame(data=vert_ump_svm_grid.cv_results_)

#vert_ump_svm_grid_results_order = ['rank_test_score', 'std_test_score', 'param_svm__C
#vert_ump_svm_grid_results_order.extend(['mean_fit_time', 'std_fit_time', 'mean_score_
#vert_ump_svm_grid_results_order.extend(['split1_test_score', 'split2_test_score', 'sp

#vert_ump_svm_grid_results = vert_ump_svm_grid_results[vert_ump_svm_grid_results_order

#vert_ump_svm_grid_results.to_csv('vert_ump_svm_grid_results.csv', index=False)
```

In [85]:
```
#vert_ump_svm_grid_results_csv = pd.read_csv('vert_ump_svm_grid_results.csv')

#vert_ump_svm_grid_results_csv
```

We now run our model with the chosen hyperparameters.

In [86]:
```
vert_ump_svm_accs = []
vert_ump_svm_f1s = []
vert_ump_svm_praucs = []
vert_ump_svm_pred_strikes = []

vert_ump_svm_feat = ['plate_x_mag', 'plate_z_hscw', 'umpire']

vert_ump_svm_preprocessor = ColumnTransformer(
    transformers=[
        ('categorical', OneHotEncoder(), ['umpire', 'plate_z_hscw']),
        ('continuous', StandardScaler(), ['plate_x_mag'])
        ],
    remainder='passthrough')


for train_index, test_index in vert_ump_svm_kfold.split(vert_ump_train, vert_ump_train
    # Splitting up our data using the indices from our kfold split
    split_train = vert_train.iloc[train_index]
    split_test = vert_train.iloc[test_index]

    # Setting up the normalization pipeline
    vert_ump_svm_pipeline = Pipeline([
        ('svm_preprocessor', vert_ump_svm_preprocessor),
        ('svm', LinearSVC(C=0.01, max_iter=5000))
    ])

    # Fitting the pipeline
    vert_ump_svm_pipeline.fit(split_train[vert_ump_svm_feat], split_train.binary_bs)

    # Predictions
    split_pred = vert_ump_svm_pipeline.predict(split_test[vert_ump_svm_feat])
```

```
    split_pred_dict = {1:0}
    for entry in split_pred:
        if entry==1:
            split_pred_dict[1] += 1
    vert_ump_svm_pred_strikes.append(split_pred_dict[1])

    # Evaluation metrics
    vert_ump_svm_accs.append( accuracy_score(split_test.binary_bs, split_pred) )
    vert_ump_svm_f1s.append( f1_score(split_test.binary_bs, split_pred) )
    vert_ump_svm_prec, vert_ump_svm_rec, _ = precision_recall_curve(split_test.binary_
    vert_ump_svm_praucs.append(auc(vert_ump_svm_rec, vert_ump_svm_prec))

vert_ump_svm_stats = pd.DataFrame(data={
    'vert_ump_svm_pred_strikes':vert_ump_svm_pred_strikes,
    'vert_ump_svm_accuracy':vert_ump_svm_accs,
    'vert_ump_svm_f1':vert_ump_svm_f1s,
    'vert_ump_svm_pr_auc':vert_ump_svm_praucs
    })

vert_ump_svm_stats.describe()
```

Out[86]:

	vert_ump_svm_pred_strikes	vert_ump_svm_accuracy	vert_ump_svm_f1	vert_ump_svm_pr_auc
count	10.000000	10.000000	10.000000	10.000000
mean	9406.000000	0.915068	0.870698	0.892157
std	64.207995	0.001671	0.002423	0.002014
min	9294.000000	0.911361	0.865503	0.887838
25%	9362.500000	0.915120	0.869961	0.891353
50%	9413.500000	0.915635	0.871563	0.892687
75%	9440.750000	0.915833	0.871973	0.893583
max	9521.000000	0.916873	0.873461	0.894242

We again compare the metrics for this model with the metrics from the relevant baseline model.

In [87]:
```
vert_ump_svm_combo_stats = pd.concat([vert_baseline_stats.describe(), vert_ump_svm_sta

vert_ump_svm_combo_stats_order = ['vert_baseline_accuracy', 'vert_ump_svm_accuracy', '
vert_ump_svm_combo_stats_order.extend(['vert_ump_svm_f1', 'vert_baseline_pr_auc', 'ver
vert_ump_svm_combo_stats = vert_ump_svm_combo_stats[vert_ump_svm_combo_stats_order]

vert_ump_svm_combo_stats
```

Out[87]:

	vert_baseline_accuracy	vert_ump_svm_accuracy	vert_baseline_f1	vert_ump_svm_f1	vert_baseline
count	1000.000000	10.000000	1000.000000	10.000000	1000
mean	0.558159	0.915068	0.329416	0.870698	(
std	0.000877	0.001671	0.001326	0.002423	(
min	0.555453	0.911361	0.325689	0.865503	(
25%	0.557592	0.915120	0.328507	0.869961	(
50%	0.558159	0.915635	0.329420	0.871563	(
75%	0.558748	0.915833	0.330347	0.871973	(
max	0.561238	0.916873	0.334000	0.873461	(

We again see similar model performance despite adding the `umpire` feature.

2.10 - Modeling with Added Vertical Location and Umpires: Predicting True Balls and Strikes

We proceed similarly to `Sections 2.3, 2.6, 2.8`.

In [88]:

```
# Note that this requires vert_ump_log_reg_kfold, which is created and run in Section
# The code below requires that code to be run first anyway, so there should be no issu
# We reference the call in 2.9.2 so that the random states are always the same

vert_ump_split_data_frames = []

# This fills the list created above with the split data frames
for train_index, test_index in vert_ump_log_reg_kfold.split(vert_ump_train, vert_ump_t
    vert_ump_split_data_frames.append(vert_ump_train.iloc[test_index])


# Next, we go through the split data frames, add the predictions, and reduce to the re
for counter in range(len(vert_ump_split_data_frames)):
    temp_df = vert_ump_split_data_frames[counter]
    temp_df = temp_df.assign(pred_bs=vert_ump_log_reg_preds[counter])
    temp_df = temp_df[['true_ball/strike', 'pred_bs', 'binary_bs', 'correct_call']]
    vert_ump_split_data_frames[counter] = temp_df


# We again go through the split data frames, check whether the predictions were correc
# and store that information
for counter in range(len(vert_ump_split_data_frames)):
    correct_preds = []
    temp_df = vert_ump_split_data_frames[counter]
    for ind in temp_df.index:
        pred = temp_df.at[ind, 'pred_bs']
        ump_call = temp_df.at[ind, 'binary_bs']
        cor_incorr = temp_df.at[ind, 'correct_call']
        if pred == ump_call:
            if cor_incorr == 'correct_call':
                correct_preds.append('correct_pred')
```

```
            else:
                correct_preds.append('incorrect_pred')
        else:
            if cor_incorr == 'correct_call':
                correct_preds.append('incorrect_pred')
            else:
                correct_preds.append('correct_pred')
    temp_df = temp_df.assign(correct_pred=correct_preds)
    vert_ump_split_data_frames[counter] = temp_df

# Finally, we make a data frame comparing the umpire calls to the predicted calls
correct_call_test_list = []
correct_pred_test_list = []

for counter in range(len(vert_ump_split_data_frames)):
    correct_call_test_list.append(hor_ump_split_data_frames[counter].correct_call.valu
    correct_pred_test_list.append(hor_ump_split_data_frames[counter].correct_pred.valu

vert_ump_log_reg_call_pred_stats = pd.DataFrame(data={
    'vert_ump_correct_call_percent':correct_call_test_list,
    'vert_ump_correct_pred_percent':correct_pred_test_list
    }
).describe()

vert_ump_log_reg_call_pred_stats
```

Out[88]:

	vert_ump_correct_call_percent	vert_ump_correct_pred_percent
count	10.000000	10.000000
mean	0.951198	0.965497
std	0.001419	0.001514
min	0.949942	0.963508
25%	0.950233	0.964325
50%	0.951032	0.965432
75%	0.951465	0.966991
max	0.954806	0.967356

2.11 - Summary of Results

Let us recall the two primary questions introduced in `Section 1` that we seek to answer in this section:

Question 1: Would a pseudo-robo-umpire (i.e. a model trained on real umpire data) be more or less accurate than an actual umpire?

Question 2: How large of an overall impact does an individual umpire have on training a pseudo-robo-umpire?

We address each of these questions in turn.

2.11.1 - Addressing Question 1

To begin, we simply recall the comparisons of correct call percentage against correct prediction percentage for the `LogisticRegression` models in `Subsections 2.3, 2.6`. We do not include the correct prediction percentages from `Subsections 2.8, 2.10`, as those models include `umpire` as a feature.

In [89]:
```
# From Section 2.3
hor_no_ump_log_reg_call_pred_stats
```

Out[89]:

	hor_no_ump_correct_call_percent	hor_no_ump_correct_pred_percent
count	10.000000	10.000000
mean	0.951358	0.963241
std	0.001892	0.001976
min	0.948128	0.960102
25%	0.950635	0.962239
50%	0.951540	0.963292
75%	0.952390	0.964796
max	0.954222	0.965760

In [90]:
```
# From Section 2.6
vert_no_ump_log_reg_call_pred_stats
```

Out[90]:

	vert_no_ump_correct_call_percent	vert_no_ump_correct_pred_percent
count	10.000000	10.000000
mean	0.917635	0.947204
std	0.001877	0.001432
min	0.914990	0.945638
25%	0.916670	0.946100
50%	0.917168	0.946842
75%	0.918605	0.947880
max	0.921406	0.949578

We do see improvements in mean correct prediction percentage over mean correct call percentage, up from 95.1% to 96.3% when using the `horizontal` data frame and up from 91.7% to 94.7% when using the `vertical` data frame.

The increase in the `vertical` case is more compelling because of both

- the larger increase between correct call percentage and correct prediction percentage, and

- the fact that the `vertical` data frame contains a more varied set of pitches than the `horizontal` data frame (revisit the beginning of `Section 2.2` for more details).

According to Umpire Scorecards, the least accurate umpire in the 2024 season has been James Jean (with an accuracy of 91.8%) and the most accuracte umpire has been Austin Jones (with an accuracy of 96%), as of the afternoon of 12 Aug 2024. While these aren't exactly comparable with the data we have been using -- our data only includes pitches that were called balls or strikes, so swinging strikes, balls in play, etc. are not considered -- it does provide a good reference point regarding the 3% difference between the correct call percentage and correct prediction percentage for the `vertical` data frame. Namely, the 3% difference is smaller than the difference between the best and worst pitch callers.

In short, a pseudo-robo-umpire would be more accurate on average, but still within the range of expected outcomes by human umpires. The pseudo-robo-umpire's performance could potentially be raised further by restricting to training only on pitch data with umpires who's correct call percentage is better than average.

2.11.2 - Addressing Question 2

We constructed models in `Sections 2.2, 2.5, 2.7, 2.9`; here is how they vary:

- In `Section 2.2`, we use the `horizontal` data frame and only the `plate_x_mag` feature;
- In `Section 2.7`, we use the `horizontal` data frame with both the `plate_x_mag` and `umpire` features;
- In `Section 2.5`, we use the `vertical` data frame with the `plate_x_mag` and `plate_z_hscw` features;
- In `Section 2.9`, we use the `vertical` data frame with the `plate_x_mag, plate_z_hscw`, and `umpire` features.

Consequently, to measure the impact of individual umpires we will compare the metrics from the models in `Section 2.2` with those in `Section 2.7`, as well as the metrics from the models in `Sections 2.5` with those in `Section 2.9`.

As can be seen in the tables below, there are only negligible improvements among all metrics for the `LinearSVC` models (differences of approximately 0.0004, 0.0004, and 0.00001), only negligible improvements among accuracy and F1-score for the `LogisticRegression` models (differences of approximately 0.0008 and 0.0008), and a minor decrease in PR-AUC for the `LogisticRegression` models from `Section 2.2` to `Section 2.7`.

In the case of the models from `Section 2.5` and `Section 2.9`, the improvements across all metrics for the `LinearSVC` models are minimal (with differences of approximately 0.0016, 0.0041, and 0.0011). It is similar for the `LogisiticRegression` models, with approximate

increases of 0.000009 and 0.00014 for accuracy and F1-score, with a decrease of 0.000042 in PR-AUC.

Given that any improvements in metrics for models which included the `umpire` feature we negligible (improvements around or less than 0.08%), this provides strong evidence that individual umpires do not have a large impact on incorrect calls.

2.11.2.1 - Comparing the Effect of Umpires on Logistic Regression Models using the `Horizontal` Data Frame

We begin by creating a data frame with all of the relevant metrics for the comparison. We will then restict to certain columns to clearly compare accuracy, F1-score, and PR-AUC.

In [91]:
```
hor_log_reg_comp = pd.concat([hor_no_ump_log_reg_stats.describe(), hor_ump_log_reg_sta
```

Accuracy

In [92]:
```
hor_log_reg_acc = hor_log_reg_comp[['hor_no_ump_log_reg_accuracy', 'hor_ump_log_reg_ac

hor_log_reg_acc
```

Out[92]:

	hor_no_ump_log_reg_accuracy	hor_ump_log_reg_accuracy
count	10.000000	10.000000
mean	0.938307	0.939068
std	0.002819	0.001768
min	0.932023	0.936085
25%	0.937519	0.938479
50%	0.938445	0.939062
75%	0.939822	0.940548
max	0.942760	0.941164

F1-Score

In [93]:
```
hor_log_reg_f1 = hor_log_reg_comp[['hor_no_ump_log_reg_f1', 'hor_ump_log_reg_f1']]

hor_log_reg_f1
```

Out[93]:

	hor_no_ump_log_reg_f1	hor_ump_log_reg_f1
count	10.000000	10.000000
mean	0.945313	0.946080
std	0.002488	0.001533
min	0.939809	0.943479
25%	0.944673	0.945674
50%	0.945422	0.946136
75%	0.946617	0.947248
max	0.949277	0.947949

PR-AUC

In [94]:
```
hor_log_reg_pr_auc = hor_log_reg_comp[['hor_no_ump_log_reg_pr_auc', 'hor_ump_log_reg_p

hor_log_reg_pr_auc
```

Out[94]:

	hor_no_ump_log_reg_pr_auc	hor_ump_log_reg_pr_auc
count	10.000000	10.000000
mean	0.959020	0.958985
std	0.001884	0.001334
min	0.954768	0.956943
25%	0.958420	0.958012
50%	0.959150	0.958738
75%	0.960157	0.960318
max	0.961795	0.960580

2.11.2.2 - Comparing the Effect of Umpires on SVM Models using the `Horizontal` Data Frame

We begin by creating a data frame with all of the relevant metrics for the comparison. We will then restict to certain columns to clearly compare accuracy, F1-score, and PR-AUC.

In [95]:
```
hor_svm_comp = pd.concat([hor_no_ump_svm_stats.describe(), hor_ump_svm_stats.describe(
```

Accuracy

In [96]:
```
hor_svm_acc = hor_svm_comp[['hor_no_ump_svm_accuracy', 'hor_ump_svm_accuracy']]

hor_svm_acc
```

Out[96]:

	hor_no_ump_svm_accuracy	hor_ump_svm_accuracy
count	10.000000	10.000000
mean	0.944785	0.945199
std	0.002450	0.001847
min	0.940660	0.942615
25%	0.943506	0.944230
50%	0.945625	0.944936
75%	0.946588	0.946169
max	0.947838	0.948785

F1-Score

In [97]:
```
hor_svm_f1 = hor_svm_comp[['hor_no_ump_svm_f1', 'hor_ump_svm_f1']]

hor_svm_f1
```

Out[97]:

	hor_no_ump_svm_f1	hor_ump_svm_f1
count	10.000000	10.000000
mean	0.951680	0.952070
std	0.002124	0.001800
min	0.948050	0.949109
25%	0.950511	0.951066
50%	0.952353	0.952049
75%	0.953274	0.952885
max	0.954259	0.955339

PR-AUC

In [98]:
```
hor_svm_pr_auc = hor_svm_comp[['hor_no_ump_svm_pr_auc', 'hor_ump_svm_pr_auc']]

hor_svm_pr_auc
```

Out[98]:

	hor_no_ump_svm_pr_auc	hor_ump_svm_pr_auc
count	10.000000	10.000000
mean	0.960427	0.960525
std	0.001663	0.001489
min	0.957640	0.957481
25%	0.959694	0.959872
50%	0.960912	0.960657
75%	0.961534	0.961348
max	0.962732	0.962731

2.11.2.3 - Comparing the Effect of Umpires on Logistic Regression Models using the `Vertical` Data Frame

We begin by creating a data frame with all of the relevant metrics for the comparison. We will then restict to certain columns to clearly compare accuracy, F1-score, and PR-AUC.

In [99]:
```
vert_log_reg_comp = pd.concat([vert_no_ump_log_reg_stats.describe(), vert_ump_log_reg_
```

Accuracy

In [100…
```
vert_log_reg_acc = vert_log_reg_comp[['vert_no_ump_log_reg_accuracy', 'vert_ump_log_re

vert_log_reg_acc
```

Out[100]:

	vert_no_ump_log_reg_accuracy	vert_ump_log_reg_accuracy
count	10.000000	10.000000
mean	0.914877	0.914884
std	0.002244	0.002290
min	0.911186	0.911364
25%	0.914328	0.914046
50%	0.914917	0.915007
75%	0.916436	0.916164
max	0.917535	0.918335

F1-Score

In [101…
```
vert_log_reg_f1 = vert_log_reg_comp[['vert_no_ump_log_reg_f1', 'vert_ump_log_reg_f1']]

vert_log_reg_f1
```

Out[101]:

	vert_no_ump_log_reg_f1	vert_ump_log_reg_f1
count	10.000000	10.000000
mean	0.870294	0.870439
std	0.003510	0.003485
min	0.864618	0.864945
25%	0.868710	0.869351
50%	0.870549	0.870597
75%	0.872875	0.871895
max	0.874436	0.875953

PR-AUC

In [102…

```
vert_log_reg_pr_auc = vert_log_reg_comp[['vert_no_ump_log_reg_pr_auc', 'vert_ump_log_r

vert_log_reg_pr_auc
```

Out[102]:

	vert_no_ump_log_reg_pr_auc	vert_ump_log_reg_pr_auc
count	10.000000	10.000000
mean	0.891946	0.891902
std	0.002837	0.002916
min	0.887184	0.887457
25%	0.891466	0.890705
50%	0.891990	0.892069
75%	0.893858	0.893683
max	0.895307	0.896196

2.11.2.4 - Comparing the Effect of Umpires on SVM Models using the `Vertical` Data Frame

We begin by creating a data frame with all of the relevant metrics for the comparison. We will then restict to certain columns to clearly compare accuracy, F1-score, and PR-AUC.

In [103…

```
vert_svm_comp = pd.concat([vert_no_ump_svm_stats.describe(), vert_ump_svm_stats.descri
```

Accuracy

In [104…

```
vert_svm_acc = vert_svm_comp[['vert_no_ump_svm_accuracy', 'vert_ump_svm_accuracy']]

vert_svm_acc
```

Out[104]:

	vert_no_ump_svm_accuracy	vert_ump_svm_accuracy
count	10.000000	10.000000
mean	0.913674	0.915068
std	0.001923	0.001671
min	0.911573	0.911361
25%	0.912312	0.915120
50%	0.913052	0.915635
75%	0.914430	0.915833
max	0.917393	0.916873

F1-Score

In [105…

```
vert_svm_f1 = vert_svm_comp[['vert_no_ump_svm_f1', 'vert_ump_svm_f1']]

vert_svm_f1
```

Out[105]:

	vert_no_ump_svm_f1	vert_ump_svm_f1
count	10.000000	10.000000
mean	0.866559	0.870698
std	0.002860	0.002423
min	0.863432	0.865503
25%	0.864563	0.869961
50%	0.865482	0.871563
75%	0.867666	0.871973
max	0.871939	0.873461

PR-AUC

In [106…

```
vert_svm_pr_auc = vert_svm_comp[['vert_no_ump_svm_pr_auc', 'vert_ump_svm_pr_auc']]

vert_svm_pr_auc
```

Out[106]:

	vert_no_ump_svm_pr_auc	vert_ump_svm_pr_auc
count	10.000000	10.000000
mean	0.891092	0.892157
std	0.002555	0.002014
min	0.888144	0.887838
25%	0.889273	0.891353
50%	0.890324	0.892687
75%	0.892096	0.893583
max	0.896091	0.894242

2.11.3 - Additional Reference Tables

The following code creates tables so that the interested reader can view the relevant metrics (accuracy, F1-score, PR-AUC), as well as information on correct call/prediction percentages.

For viewing considerations, the tables are currently commented out by default in the sections below.

In [107…

```
all_desc = [
    hor_baseline_stats.describe(),
    hor_no_ump_log_reg_stats.describe(),
    hor_no_ump_svm_stats.describe(),
    hor_ump_log_reg_stats.describe(),
    hor_ump_svm_stats.describe(),
    vert_baseline_stats.describe(),
    vert_no_ump_log_reg_stats.describe(),
    vert_no_ump_svm_stats.describe(),
    vert_ump_log_reg_stats.describe(),
    vert_ump_svm_stats.describe(),
]

all_combo_stats = pd.concat(all_desc, axis=1)

# accuracy
hor_acc_cols = [
    'hor_baseline_accuracy',
    'hor_no_ump_log_reg_accuracy',
    'hor_no_ump_svm_accuracy',
    'hor_ump_log_reg_accuracy',
    'hor_ump_svm_accuracy',
]

hor_acc_stats = all_combo_stats[hor_acc_cols]

vert_acc_cols = [
    'vert_baseline_accuracy',
    'vert_no_ump_log_reg_accuracy',
    'vert_no_ump_svm_accuracy',
    'vert_ump_log_reg_accuracy',
```

```
    'vert_ump_svm_accuracy',
]

vert_acc_stats = all_combo_stats[vert_acc_cols]


# F1-scores
hor_f1_cols = [
    'hor_baseline_f1',
    'hor_no_ump_log_reg_f1',
    'hor_no_ump_svm_f1',
    'hor_ump_log_reg_f1',
    'hor_ump_svm_f1',
]

hor_f1_stats = all_combo_stats[hor_f1_cols]

vert_f1_cols = [
    'vert_baseline_f1',
    'vert_no_ump_log_reg_f1',
    'vert_no_ump_svm_f1',
    'vert_ump_log_reg_f1',
    'vert_ump_svm_f1',
]

vert_f1_stats = all_combo_stats[vert_f1_cols]


# PR-AUCs
hor_pr_auc_cols = [
    'hor_baseline_pr_auc',
    'hor_no_ump_log_reg_pr_auc',
    'hor_no_ump_svm_pr_auc',
    'hor_ump_log_reg_pr_auc',
    'hor_ump_svm_pr_auc',
]

hor_pr_auc_stats = all_combo_stats[hor_pr_auc_cols]

vert_pr_auc_cols = [
    'vert_baseline_pr_auc',
    'vert_no_ump_log_reg_pr_auc',
    'vert_no_ump_svm_pr_auc',
    'vert_ump_log_reg_pr_auc',
    'vert_ump_svm_pr_auc',
]

vert_pr_auc_stats = all_combo_stats[vert_pr_auc_cols]

# Predictions vs. Umpires
all_pred_list = [
     hor_no_ump_log_reg_call_pred_stats,
    vert_no_ump_log_reg_call_pred_stats,
    hor_ump_log_reg_call_pred_stats,
    vert_ump_log_reg_call_pred_stats
]

all_preds = pd.concat(all_pred_list, axis=1)

hor_no_ump_pred_minus_call = all_preds.at['mean', 'hor_no_ump_correct_pred_percent'] -
```

```
vert_no_ump_pred_minus_call = all_preds.at['mean', 'vert_no_ump_correct_pred_percent']
hor_ump_pred_minus_call = all_preds.at['mean', 'hor_ump_correct_pred_percent'] - all_p
vert_ump_pred_minus_call = all_preds.at['mean', 'vert_ump_correct_pred_percent'] - all

preds_minus_calls_vals = [
    hor_no_ump_pred_minus_call,
    vert_no_ump_pred_minus_call,
    hor_ump_pred_minus_call,
    vert_ump_pred_minus_call
]

preds_minus_calls = pd.DataFrame(
    {'Correct Prediction Rate - Correct Call Rate':preds_minus_calls_vals},
    index = ['hor_no_ump', 'vert_no_ump', 'hor_ump', 'vert_ump']
)
```

2.11.3.1 - Accuracy Tables

In [108…
```
#hor_acc_stats
```

In [109…
```
#vert_acc_stats
```

2.11.3.2 - F1-Score Tables

In [110…
```
#hor_f1_stats
```

In [111…
```
#vert_f1_stats
```

2.11.3.3 - PR-AUC Tables

In [112…
```
#hor_pr_auc_stats
```

In [113…
```
#vert_pr_auc_stats
```

2.11.3.4 - Correct Calls and Correct Predictions Table

In [114…
```
#all_preds
```

In [115…
```
#preds_minus_calls
```

---

3 - Non-Location Factors: Visualizations and Feature Selection

Recall that in this section, we are focusing on the following questions:

Question 3: What factors (beyond pitch location) most impact an umpire making an incorrect ball/strike call?

We begin by reading in the data for this section.

In [116…
```
all_features_df = pd.read_csv("large_model_data.csv")

pd.set_option('display.max_columns', None)

all_features_df
```

Out[116]:

	ball/strike	binary_bs	true_ball/strike	correct_call	zone	hscw	plate_x	plate_x_mag	plate
0	S	1	true_strike	correct_call	7.0	heart	-0.53	0.53	
1	B	0	true_ball	correct_call	14.0	waste	1.89	1.89	
2	B	0	true_ball	correct_call	13.0	shadow	-0.84	0.84	
3	S	1	true_strike	correct_call	8.0	heart	-0.21	0.21	
4	S	1	true_strike	correct_call	5.0	heart	0.00	0.00	
...	...	...	...	...	...	...	...	...	
358476	S	1	true_strike	correct_call	4.0	heart	-0.31	0.31	
358477	S	1	true_strike	correct_call	7.0	shadow	-0.77	0.77	
358478	B	0	true_ball	correct_call	11.0	shadow	-0.16	0.16	
358479	B	0	true_ball	correct_call	3.0	shadow	0.57	0.57	
358480	S	1	true_strike	correct_call	8.0	heart	-0.23	0.23	

358481 rows × 36 columns

The columns of this data set are:

- `ball/strike` - the umpire's call on the field of if that pitch was a ball or strike ^
- `binary_bs` - a binary version of `ball/strike`
- `true_ball/strike` - an assessment of if the pitch was a ball or called strike based on the regulation strike zone
- `correct_call` - checks if the umpire's call matches the correct call according to the regulation strike zone
- `zone` - the Gameday Zone of the pitch ^

- `hscw` - records if the pitch was in the heart, shadow, chase, or waste region of the strike zone
- `plate_x` - the horizontal location of the pitch (relative to the middle of home plate) as the pitch crossed home plate ^
- `plate_x_mag` - absolute value of `plate_x`
- `plate_x_dir` - sign of `plate_x`
- `plate_z` - the height of the pitch ^
- `sz_top` - the top height of the regulation strike zone (changes per batter) ^
- `sz_bot` - the bottom height of the regulation strike zone (changes per batter) ^
- `release_speed` - speed of pitch out-of-hand, as measured by Statcast (since data is from 2023) ^
- `release_pos_x` - horizontal release position of the ball (relative to the middle of the plate) ^
- `release_pos_y` - vertical release position of the ball (relative to the middle of the plate) ^
- `balls` - number of balls in count before the pitch ^
- `strikes` - number of strikes in count before the pitch ^
- `pfx_x` - horizontal movement of the pitch ^
- `pfx_z` - vertical movement of the pitch ^
- `outs_when_up` - number of out before the pitch ^
- `inning` - inning number before the pitch ^
- `vx0` - horizontal velocity of the pitch (measured at y=50) ^
- `vy0` - depth velocity of the pitch (measured at y=50) ^
- `vz0` - vertical velocity of the pitch (measured at y=50) ^
- `ax` - horizontal acceleration of the pitch (measured at y=50) ^
- `ay` - depth acceleration of the pitch (measured at y=50) ^
- `az` - vertical acceleration of the pitch (measured at y=50) ^
- `effective_speed` - adjusted speed based on pitcher's extension at release ^
- `release_spin_rate` - spin rate of pitch, as measured by Statcast ^
- `release_extension` - pitcher's extension at release, as measured by Statcast ^
- `release_pos_y` - depth of release position, as measured from catcher's perspective ^
- `at_bat_number` - plate appearance number for the game ^
- `pitch_number` - number of pitch in the given plate appearance ^
- `home_score` - score of the home team before the pitch ^
- `away_score` - score of the away team before the pitch ^
- `umpire` - identity of home plate umpire, as scraped from Baseball Reference

^ as provided by Baseball Savant, see the documentation for more details

3.1 - Visualizations

We will soon visually examine how most of our non-location features impact the rate at which umpires correctly call pitches. However, we first categorize how and why the features present

will or will not be used.

- Firstly, note that we are tracking correct call percentage across features, so we will not be using `correct_call` as its own feature.
- Secondly, `ball/strike`, `binary_bs`, `true_ball/strike`, `plate_x`, `plate_x_mag`, `plate_x_dir`, and `plate_z` are all directly tied to pitch location, so we will not be using them as independent features in this section.
- Thirdly, we will include the `zone` feature at times for completeness, but it is also based on pitch location and thus will not be a focus of this section. Additionally, we have already considered the `umpire` feature in `Section 2`, so we will not consider it here.
- Finally, we note that `effective_speed` is an adjusted version of `release_speed` based on `release_extension`; as a result, we only use `effective_speed` in this section.

For continuous features, we will indicate quartiles using red and black lines. For certain linear categorical features, we will also include relevant information about the distribution to indicate outliers (such as `pitch_number`). We will also include a purple line in our visualizations to represent the mean correct call percentage from across this dataset (91.27%) as a reference point.

Note that certain data is collected at a level of granularity that would require a tremendous amount of data to have sufficiently many samples; for example, `sz_top` data is collected at a precision of 2 decimal places (e.g. 3.33). In cases like this, we will collect data points into large "buckets" for increase stability. In the case of `sz_top`, every pitch with an `sz_top` value between 3.3 and 3.4 will be collected into one group labeled "3.35". We then plot the correct call percentage for all of these groups instead of the original features. Whenever we use a bucket version of a feature, we will make explicit note of this process.

3.1.1 - Regions

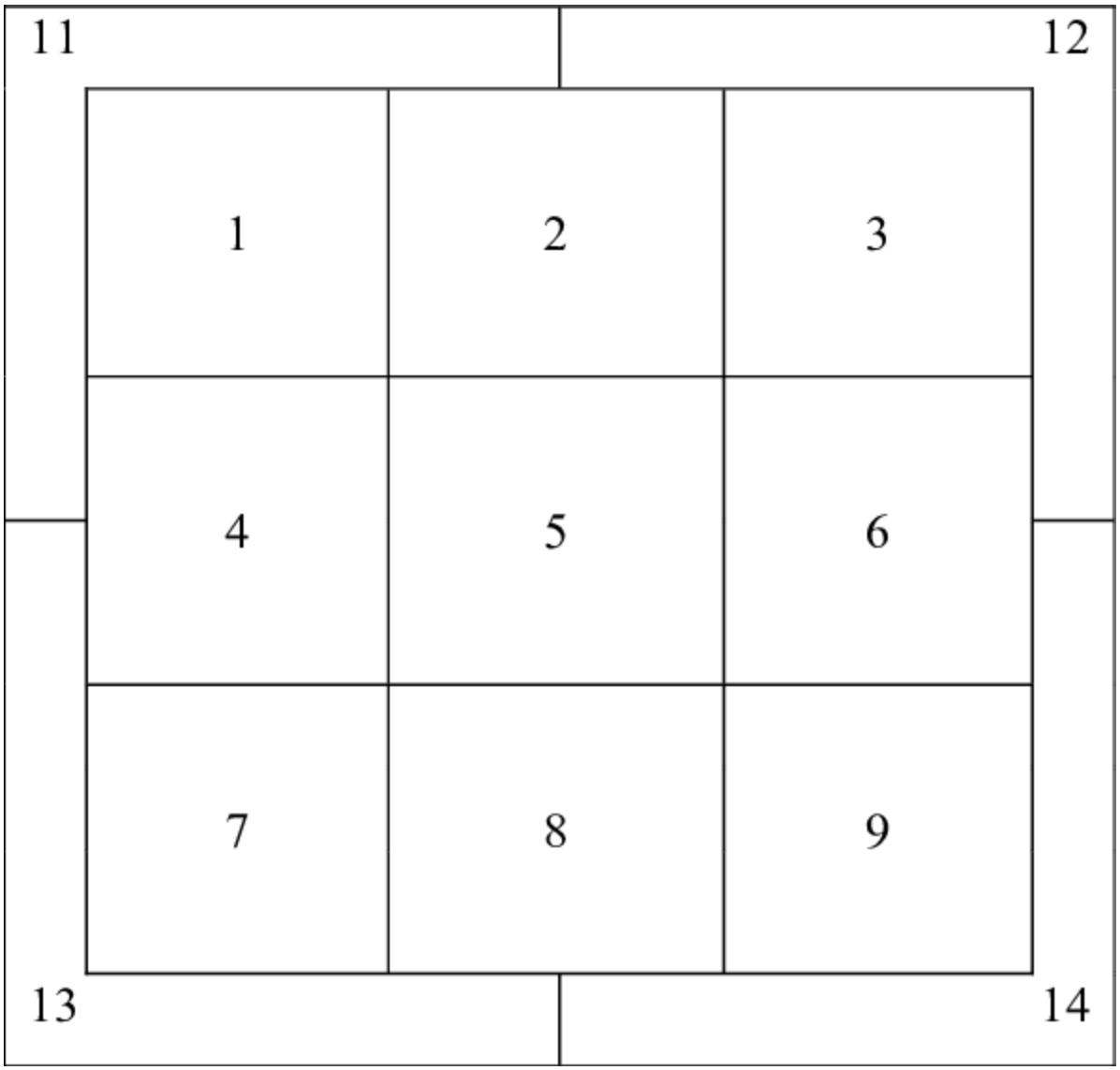
3.1.1.1 - `zone`

We begin by examining how correct call percentage varies by Statcast Gameday Zone. For reference, we include the following visual and note that zones 1-9 are within the regulation strike zone and that zones 11-14 are outside of the regulation strike zone.

In [117…

```
Image("statcast-gameday-zones.png")
```

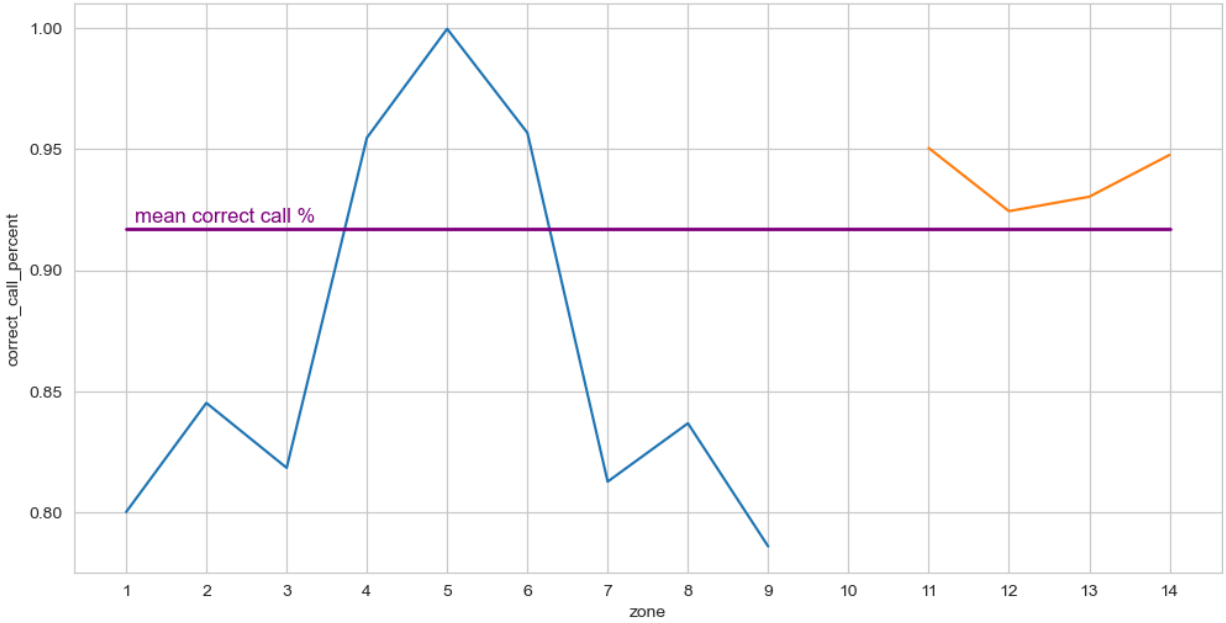
Out[117]:



In [118…

```
zone_percents_1 = []
zone_index_1 = []

zone_percents_2 = []
zone_index_2 = []

for index in range(1,10):
    temp_list = all_features_df[all_features_df['zone'] == index].correct_call.value_c
    zone_percents_1.append(temp_list[0])
    zone_index_1.append(index)

for index in range(11,15):
    temp_list = all_features_df[all_features_df['zone'] == index].correct_call.value_c
    zone_percents_2.append(temp_list[0])
    zone_index_2.append(index)

zone_stats_1 = pd.DataFrame({'zone':zone_index_1, 'correct_call_percent':zone_percents
zone_stats_2 = pd.DataFrame({'zone':zone_index_2, 'correct_call_percent':zone_percents

plt.figure(figsize=(12,6))
sns.lineplot(data=zone_stats_1, x='zone', y='correct_call_percent').set_xticks(range(1
sns.lineplot(data=zone_stats_2, x='zone', y='correct_call_percent').set_xticks(range(1
plt.plot([1, 14], [0.9172, 0.9172], color='purple', linewidth=2)
```

```
plt.text(1.1, 0.92, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



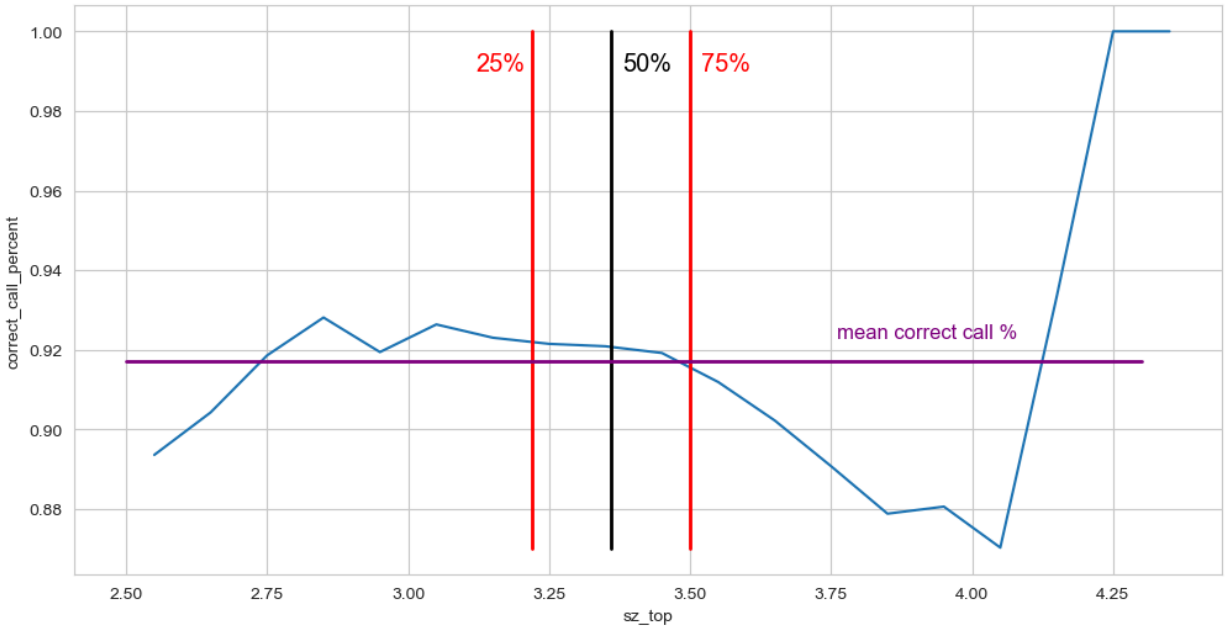
3.1.2 - Boundaries (top and bottom of the strike zone)

3.1.2.1 - `sz_top`

For `sz_top`, we split our data into buckets with a constant width of 0.1.

In [119…

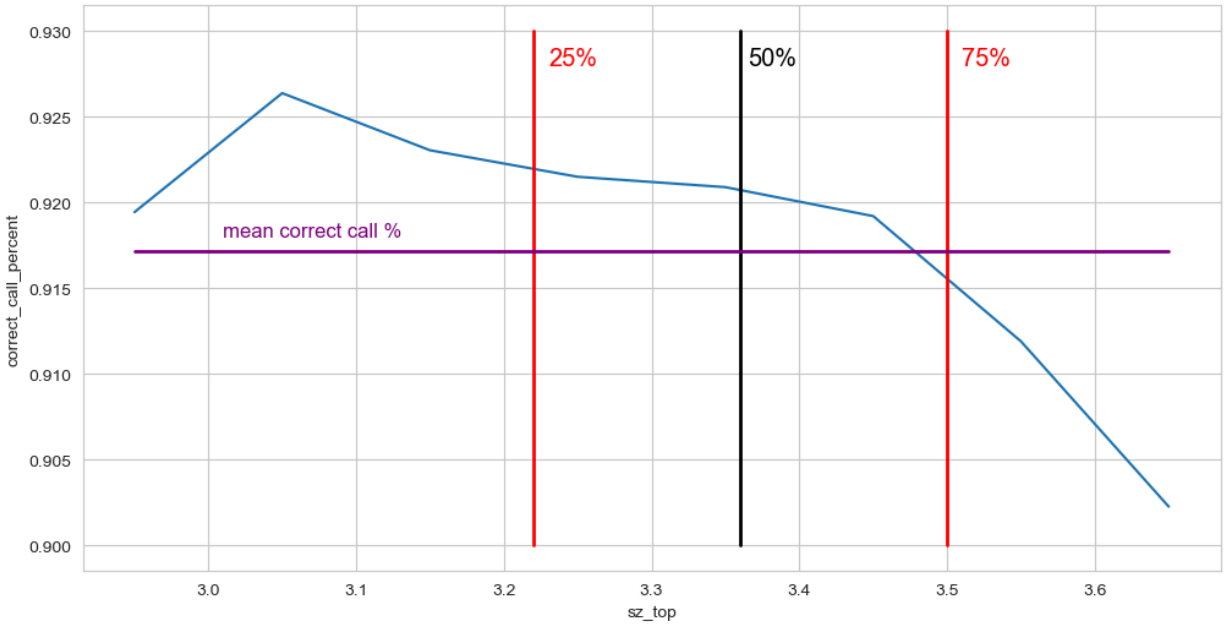
```
sz_top_percents = []
sz_top_index = []

for index in range(25,44):
    temp_list = all_features_df[
        (index*0.1 <= all_features_df['sz_top']) &
        (all_features_df['sz_top'] < (index*0.1)+0.1)].correct_call.value_counts(norma
    sz_top_percents.append(temp_list[0])
    sz_top_index.append(index*0.1+0.05)

sz_top_stats = pd.DataFrame({'sz_top':sz_top_index, 'correct_call_percent':sz_top_perc

plt.figure(figsize=(12,6))
sns.lineplot(data=sz_top_stats, x='sz_top', y='correct_call_percent')
plt.plot([3.22, 3.22], [0.87, 1], color='red', linewidth=2)
plt.plot([3.36, 3.36], [0.87, 1], color='black', linewidth=2)
plt.plot([3.5, 3.5], [0.87, 1], color='red', linewidth=2)
plt.text(3.12, 0.99, '25%', color='red', fontsize='x-large')
plt.text(3.38, 0.99, '50%', color='black', fontsize='x-large')
plt.text(3.52, 0.99, '75%', color='red', fontsize='x-large')
plt.plot([2.5, 4.3], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(3.76, 0.923, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a top height of the strike zone between 3.22 and 3.5, so we zoom in on that region:

In [120…

```
plt.figure(figsize=(12,6))
sns.lineplot(
    data=sz_top_stats[(2.95 <= sz_top_stats['sz_top']) & (sz_top_stats['sz_top'] < 3.7
    x='sz_top',
    y='correct_call_percent')
plt.plot([3.22, 3.22], [0.9, 0.93], color='red', linewidth=2)
plt.plot([3.36, 3.36], [0.9, 0.93], color='black', linewidth=2)
plt.plot([3.5, 3.5], [0.9, 0.93], color='red', linewidth=2)
plt.text(3.23, 0.928, '25%', color='red', fontsize='x-large')
plt.text(3.365, 0.928, '50%', color='black', fontsize='x-large')
plt.text(3.51, 0.928, '75%', color='red', fontsize='x-large')
plt.plot([2.95, 3.65], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(3.01, 0.918, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```

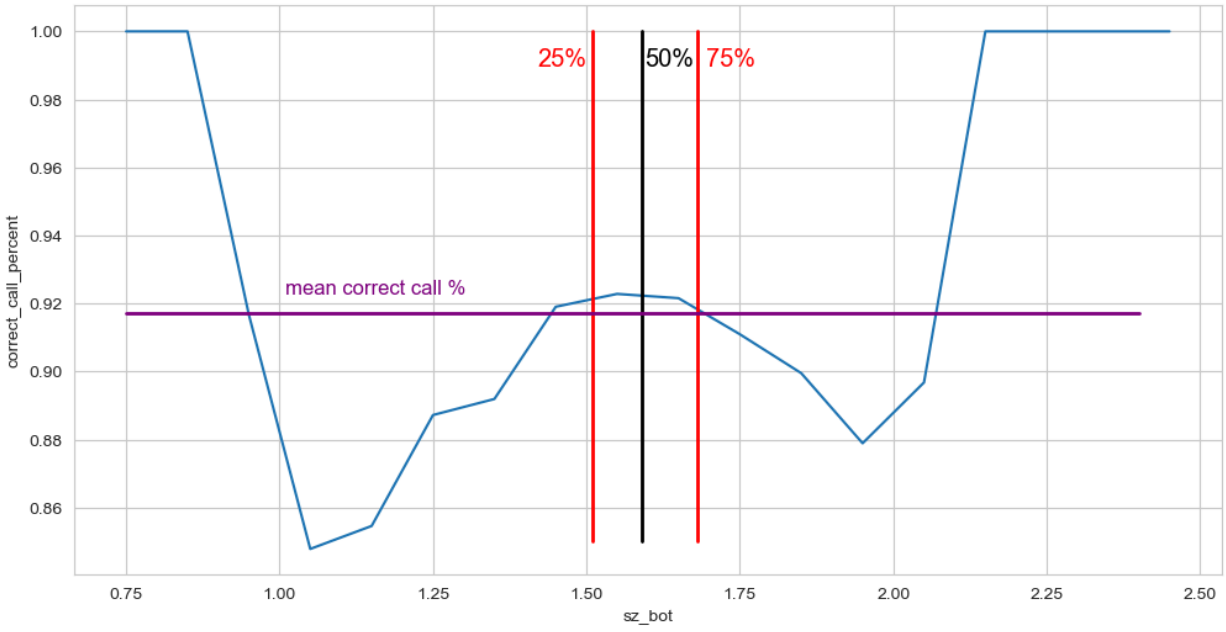


In [121…
```
# For summary statistics on sz_top, run the following:
# all_features_df.sz_top.describe()
```

3.1.2.2 - `sz_bot`

For `sz_bot`, we split our data into buckets with a constant width of 0.1.

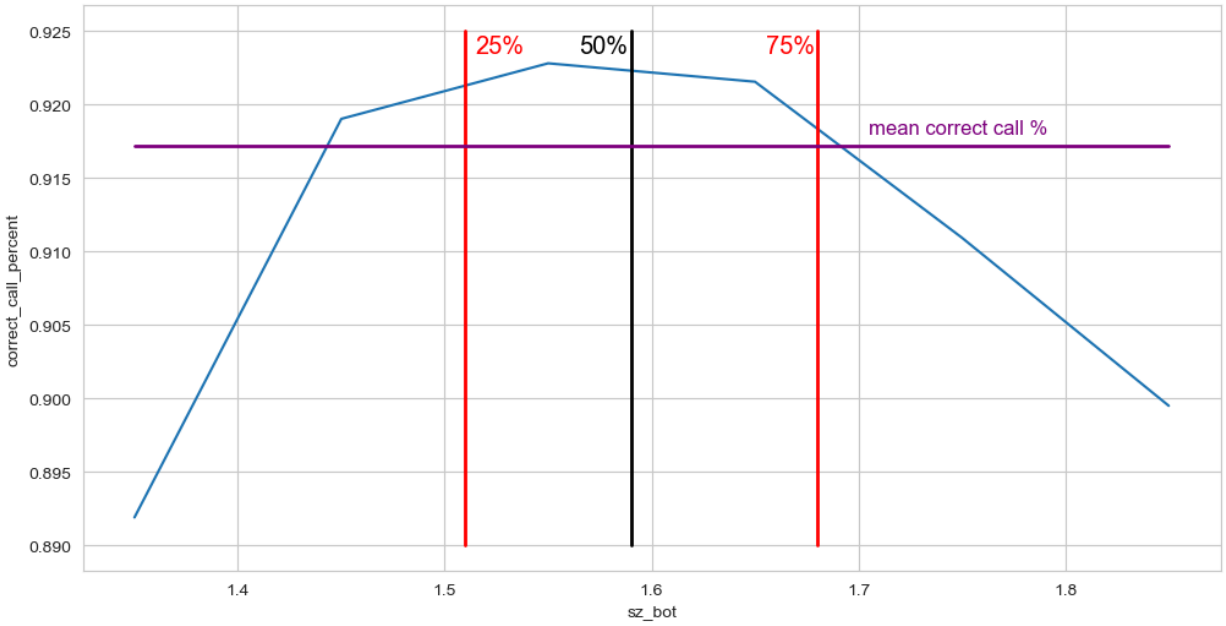
In [122…
```
sz_bot_percents = []
sz_bot_index = []

for index in range(7,25):
    temp_list = all_features_df[
        (index*0.1 <= all_features_df['sz_bot']) &
        (all_features_df['sz_bot'] < (index*0.1)+0.1)].correct_call.value_counts(norma
    sz_bot_percents.append(temp_list[0])
    sz_bot_index.append(index*0.1+0.05)

sz_bot_stats = pd.DataFrame({'sz_bot':sz_bot_index, 'correct_call_percent':sz_bot_perc

plt.figure(figsize=(12,6))
sns.lineplot(data=sz_bot_stats, x='sz_bot', y='correct_call_percent')
plt.plot([1.51, 1.51], [0.85, 1], color='red', linewidth=2)
plt.plot([1.59, 1.59], [0.85, 1], color='black', linewidth=2)
plt.plot([1.68, 1.68], [0.85, 1], color='red', linewidth=2)
plt.text(1.42, 0.99, '25%', color='red', fontsize='x-large')
plt.text(1.595, 0.99, '50%', color='black', fontsize='x-large')
plt.text(1.695, 0.99, '75%', color='red', fontsize='x-large')
plt.plot([0.75, 2.4], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(1.01, 0.923, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a bottom height of the strike zone between 1.51 and 1.68, so we zoom in on that region:

In [123…

```
plt.figure(figsize=(12,6))
sns.lineplot(
    data=sz_bot_stats[(1.3 < sz_bot_stats['sz_bot']) & (sz_bot_stats['sz_bot'] < 1.9)]
    x='sz_bot',
    y='correct_call_percent')
plt.plot([1.51, 1.51], [0.89, 0.925], color='red', linewidth=2)
plt.plot([1.59, 1.59], [0.89, 0.925], color='black', linewidth=2)
plt.plot([1.68, 1.68], [0.89, 0.925], color='red', linewidth=2)
plt.text(1.515, 0.9235, '25%', color='red', fontsize='x-large')
plt.text(1.565, 0.9235, '50%', color='black', fontsize='x-large')
plt.text(1.655, 0.9235, '75%', color='red', fontsize='x-large')
plt.plot([1.35, 1.85], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(1.705, 0.918, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [124…

```
# For summary statistics, run the following:
# all_features_df.sz_bot.describe()
```

3.1.3 - Release

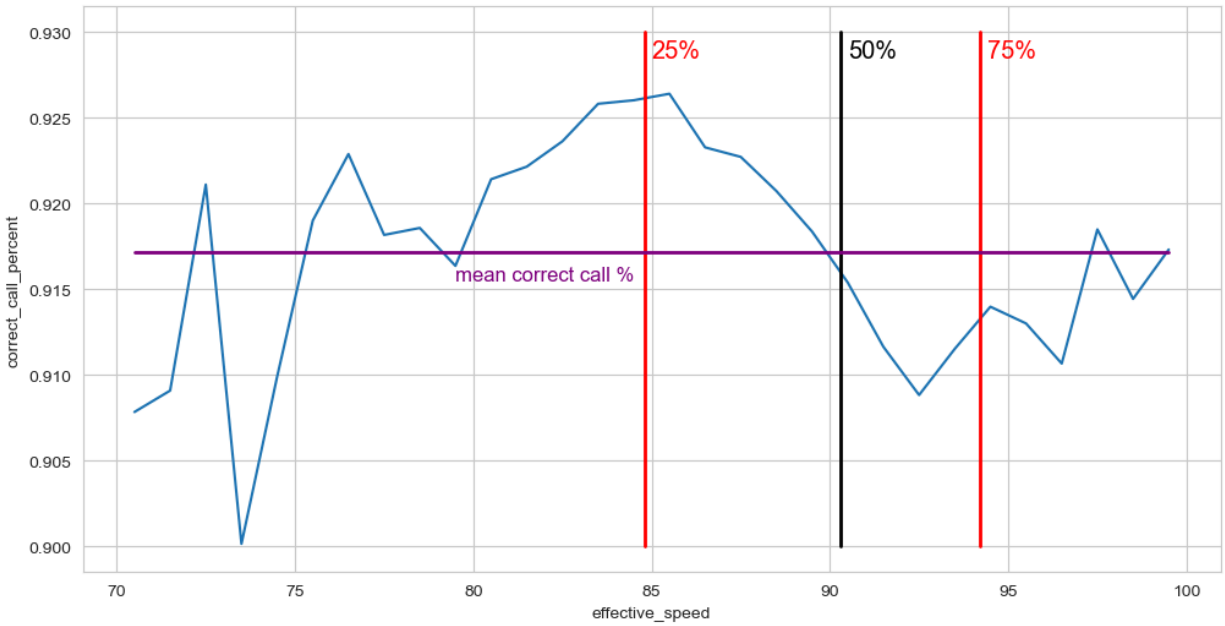
3.1.3.1 - `effective_speed`

For `sz_bot`, we split our data into buckets with a constant width of 1.

In [125…

```
effective_speed_percents = []
effective_speed_index = []

for index in range(70,100):
    temp_list = all_features_df[
        (index <= all_features_df['effective_speed']) &
        (all_features_df['effective_speed'] < index+1)].correct_call.value_counts(norm
    effective_speed_percents.append(temp_list[0])
    effective_speed_index.append(index+0.5)

effective_speed_stats = pd.DataFrame({'effective_speed':effective_speed_index,
```

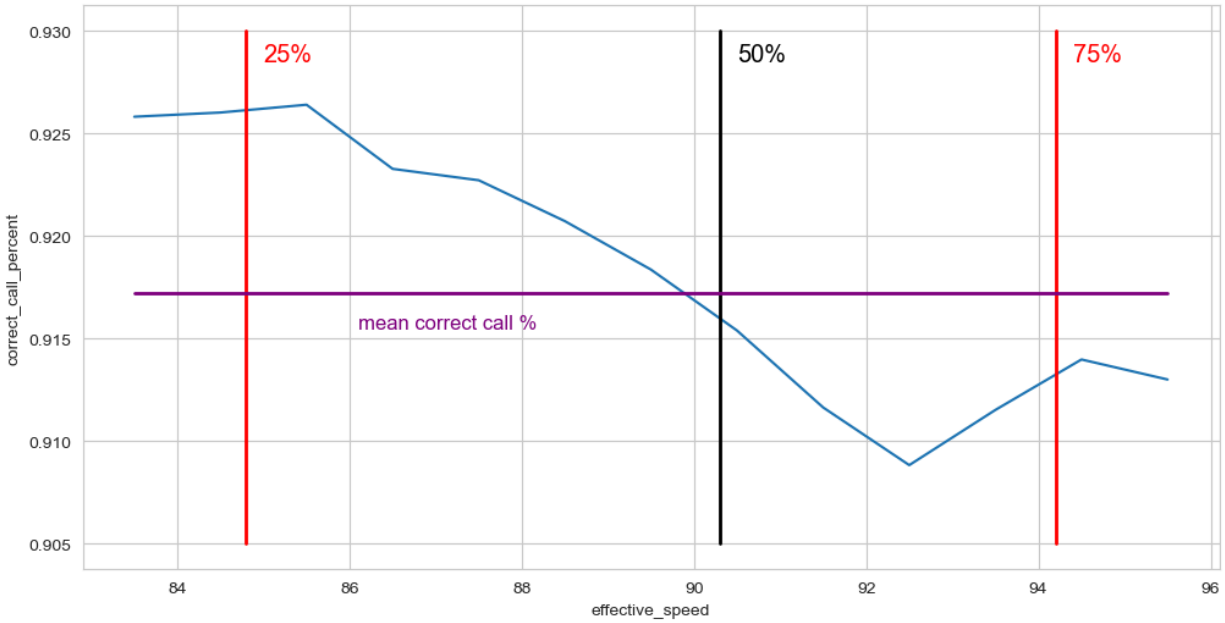
```
                                        'correct_call_percent':effective_speed_percents}

plt.figure(figsize=(12,6))
sns.lineplot(data=effective_speed_stats, x='effective_speed', y='correct_call_percent'
plt.plot([84.8, 84.8], [0.9, 0.93], color='red', linewidth=2)
plt.plot([90.3, 90.3], [0.9, 0.93], color='black', linewidth=2)
plt.plot([94.2, 94.2], [0.9, 0.93], color='red', linewidth=2)
plt.text(85.0, 0.9285, '25%', color='red', fontsize='x-large')
plt.text(90.5, 0.9285, '50%', color='black', fontsize='x-large')
plt.text(94.4, 0.9285, '75%', color='red', fontsize='x-large')
plt.plot([70.5, 99.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(79.5, 0.9155, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with an effective speed between 84.8 and 94.2, so we zoom in on that region:

In [126…

```
plt.figure(figsize=(12,6))
sns.lineplot(
    data=effective_speed_stats[
        (83 < effective_speed_stats['effective_speed']) &
        (effective_speed_stats['effective_speed'] < 96)],
    x='effective_speed',
    y='correct_call_percent')
plt.plot([84.8, 84.8], [0.905, 0.93], color='red', linewidth=2)
plt.plot([90.3, 90.3], [0.905, 0.93], color='black', linewidth=2)
plt.plot([94.2, 94.2], [0.905, 0.93], color='red', linewidth=2)
plt.text(85.0, 0.9285, '25%', color='red', fontsize='x-large')
plt.text(90.5, 0.9285, '50%', color='black', fontsize='x-large')
plt.text(94.4, 0.9285, '75%', color='red', fontsize='x-large')
plt.plot([83.5, 95.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(86.1, 0.9155, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```

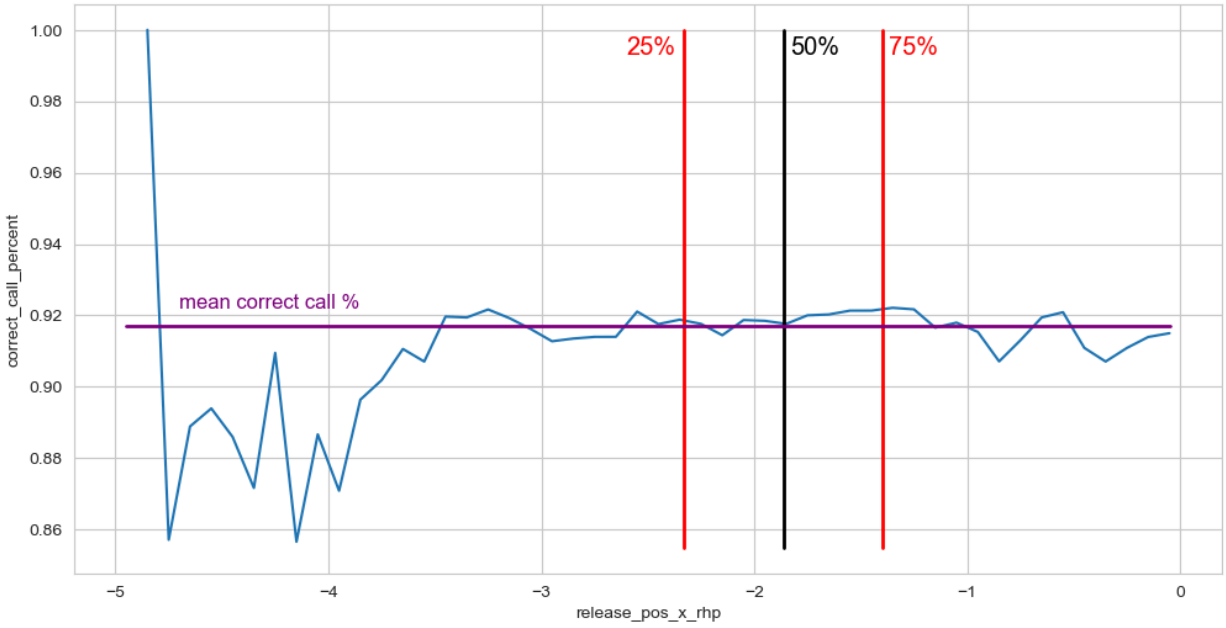


In [127…
```
# For summary statistics, run the following:
# all_features_df.effective_speed.describe()
```

3.1.3.2 - `release_pos_x` (RHP)

We split our data into buckets with a constant width of 0.1.

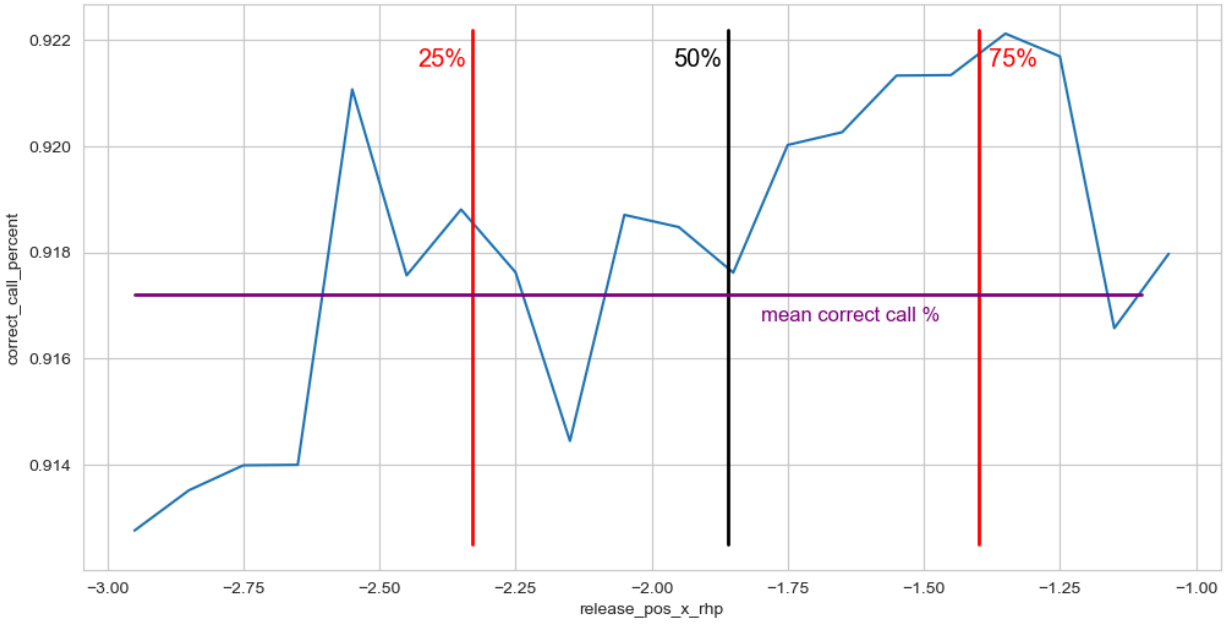
In [128…
```
release_pos_x_rhp_percents = []
release_pos_x_rhp_index = []

for index in range(-49,0):
    temp_list = all_features_df[
        (index*0.1 <= all_features_df['release_pos_x']) &
        (all_features_df['release_pos_x'] < (index*0.1)+0.1)].correct_call.value_count
    release_pos_x_rhp_percents.append(temp_list[0])
    release_pos_x_rhp_index.append(index*0.1+0.05)

release_pos_x_rhp_stats = pd.DataFrame({
    'release_pos_x_rhp':release_pos_x_rhp_index,
    'correct_call_percent':release_pos_x_rhp_percents
    })

plt.figure(figsize=(12,6))
sns.lineplot(data=release_pos_x_rhp_stats, x='release_pos_x_rhp', y='correct_call_perc
plt.plot([-2.33, -2.33], [0.855, 1], color='red', linewidth=2)
plt.plot([-1.86, -1.86], [0.855, 1], color='black', linewidth=2)
plt.plot([-1.40, -1.40], [0.855, 1], color='red', linewidth=2)
plt.text(-2.6, 0.993, '25%', color='red', fontsize='x-large')
plt.text(-1.83, 0.993, '50%', color='black', fontsize='x-large')
plt.text(-1.37, 0.993, '75%', color='red', fontsize='x-large')
plt.plot([-4.95, -0.05], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-4.7, 0.922, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches (from right-handed pitchers) occur with a horizontal release position between -2.33 and -1.40, so we zoom in on that region:

In [129…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=release_pos_x_rhp_stats[
    (-3 < release_pos_x_rhp_stats['release_pos_x_rhp']) &
    (release_pos_x_rhp_stats['release_pos_x_rhp'] < -1)
    ],
    x='release_pos_x_rhp',
    y='correct_call_percent')
plt.plot([-2.33, -2.33], [0.9125, 0.9222], color='red', linewidth=2)
plt.plot([-1.86, -1.86], [0.9125, 0.9222], color='black', linewidth=2)
plt.plot([-1.40, -1.40], [0.9125, 0.9222], color='red', linewidth=2)
plt.text(-2.43, 0.9215, '25%', color='red', fontsize='x-large')
plt.text(-1.96, 0.9215, '50%', color='black', fontsize='x-large')
plt.text(-1.38, 0.9215, '75%', color='red', fontsize='x-large')
plt.plot([-2.95, -1.1], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-1.8, 0.9167, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```

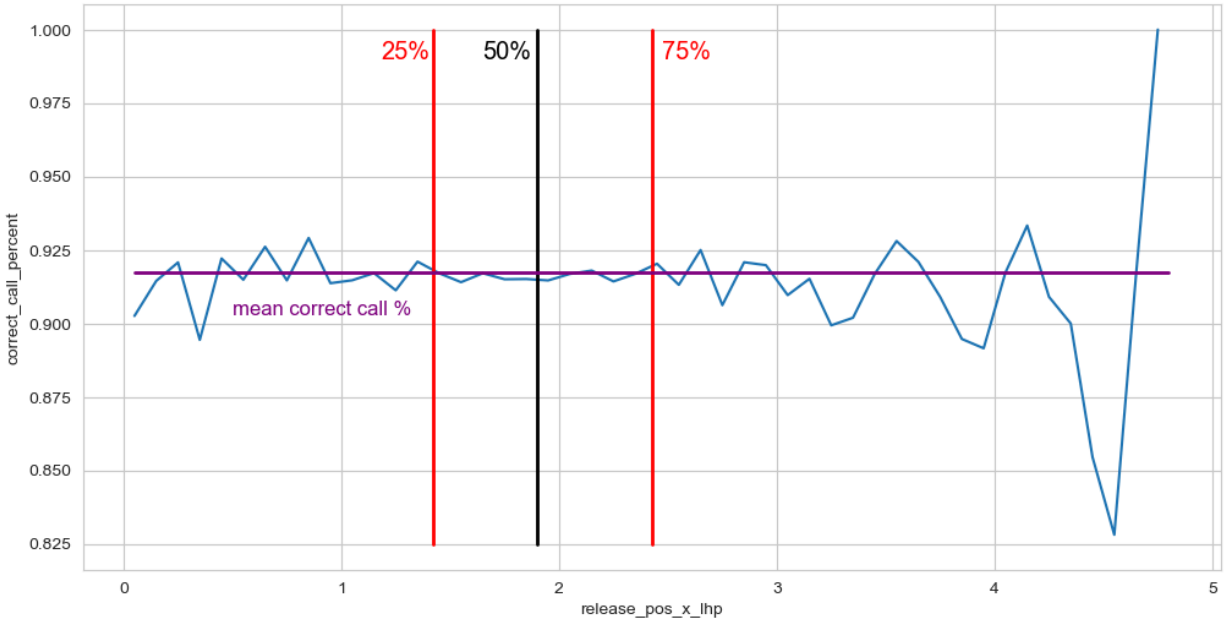


In [130…
```
# For summary statistics, run the following:
# all_features_df[all_features_df['release_pos_x'] < 0].release_pos_x.describe()
```

3.1.3.3 - `release_pos_x` (LHP)

We split our data into buckets with a constant width of 0.1.

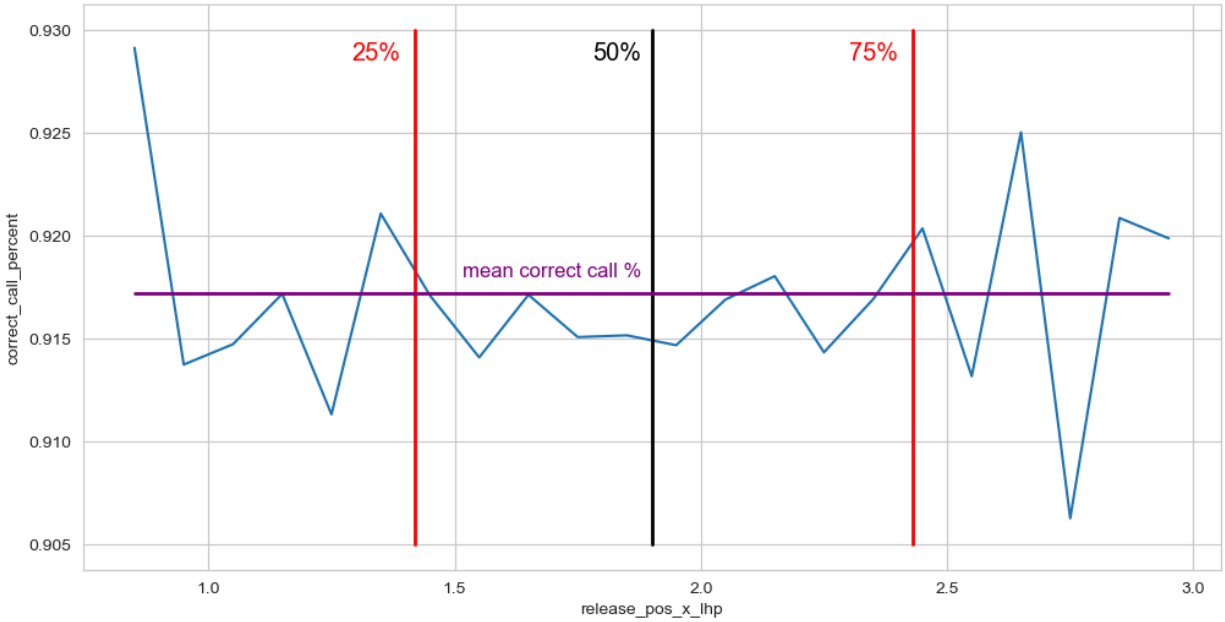
In [131…
```
release_pos_x_lhp_percents = []
release_pos_x_lhp_index = []

for index in range(0,48):
    temp_list = all_features_df[
        (index*0.1 <= all_features_df['release_pos_x']) &
        (all_features_df['release_pos_x'] < (index*0.1)+0.1)].correct_call.value_count
    release_pos_x_lhp_percents.append(temp_list[0])
    release_pos_x_lhp_index.append(index*0.1+0.05)

release_pos_x_lhp_stats = pd.DataFrame({
    'release_pos_x_lhp':release_pos_x_lhp_index,
    'correct_call_percent':release_pos_x_lhp_percents
    })

plt.figure(figsize=(12,6))
sns.lineplot(data=release_pos_x_lhp_stats, x='release_pos_x_lhp', y='correct_call_perc
plt.plot([1.42, 1.42], [0.825, 1], color='red', linewidth=2)
plt.plot([1.90, 1.90], [0.825, 1], color='black', linewidth=2)
plt.plot([2.43, 2.43], [0.825, 1], color='red', linewidth=2)
plt.text(1.18, 0.99, '25%', color='red', fontsize='x-large')
plt.text(1.65, 0.99, '50%', color='black', fontsize='x-large')
plt.text(2.47, 0.99, '75%', color='red', fontsize='x-large')
plt.plot([0.05, 4.8], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(0.5, 0.903, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches (from left-handed pitchers) occur with a horizontal release position between 1.42 and 2.43, so we zoom in on that region:

In [132…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=release_pos_x_lhp_stats[
    (0.8 < release_pos_x_lhp_stats['release_pos_x_lhp']) &
    (release_pos_x_lhp_stats['release_pos_x_lhp'] < 3)
    ],
    x='release_pos_x_lhp',
    y='correct_call_percent')
plt.plot([1.42, 1.42], [0.905, 0.93], color='red', linewidth=2)
plt.plot([1.90, 1.90], [0.905, 0.93], color='black', linewidth=2)
plt.plot([2.43, 2.43], [0.905, 0.93], color='red', linewidth=2)
plt.text(1.29, 0.9285, '25%', color='red', fontsize='x-large')
plt.text(1.78, 0.9285, '50%', color='black', fontsize='x-large')
plt.text(2.3, 0.9285, '75%', color='red', fontsize='x-large')
plt.plot([0.85, 2.95], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(1.515, 0.918, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```

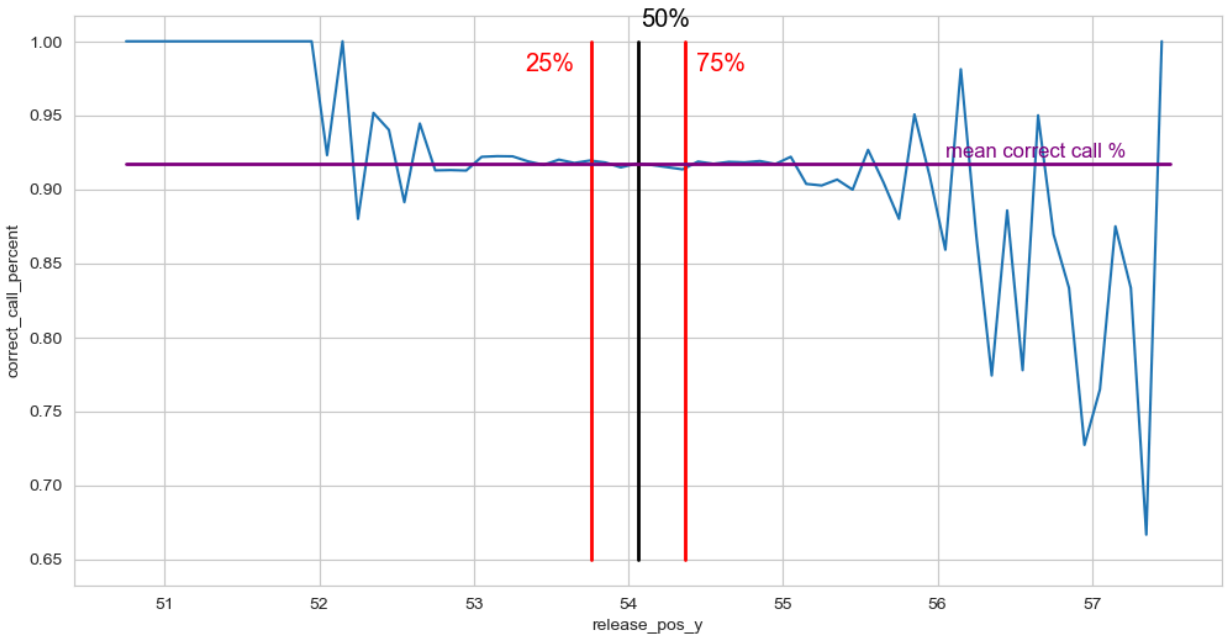


In [133…
```
# For summary statistics, run the following:
# all_features_df[all_features_df['release_pos_x'] >= 0].release_pos_x.describe()
```

3.1.3.4 - `release_pos_y`

We split our data into buckets with a constant width of 1.

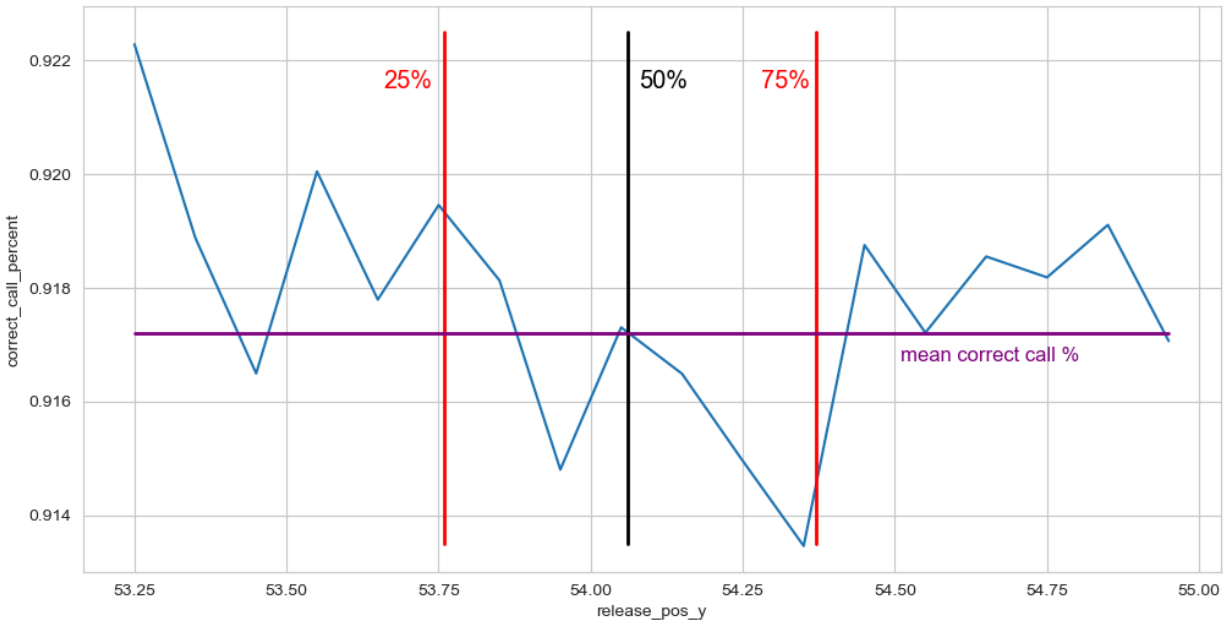
In [134…
```
release_pos_y_percents = []
release_pos_y_index = []

for index in range(500,575):
    temp_list = all_features_df[
        (index*0.1 <= all_features_df['release_pos_y']) &
        (all_features_df['release_pos_y'] < (index*0.1)+0.1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        release_pos_y_percents.append(temp_list[0])
        release_pos_y_index.append(index*0.1+0.05)

release_pos_y_stats = pd.DataFrame({
    'release_pos_y':release_pos_y_index,
    'correct_call_percent':release_pos_y_percents
    })

plt.figure(figsize=(12,6))
sns.lineplot(data=release_pos_y_stats, x='release_pos_y', y='correct_call_percent')
plt.plot([53.76, 53.76], [0.65, 1], color='red', linewidth=2)
plt.plot([54.06, 54.06], [0.65, 1], color='black', linewidth=2)
plt.plot([54.37, 54.37], [0.65, 1], color='red', linewidth=2)
plt.text(53.33, 0.98, '25%', color='red', fontsize='x-large')
plt.text(54.08, 1.01, '50%', color='black', fontsize='x-large')
plt.text(54.44, 0.98, '75%', color='red', fontsize='x-large')
plt.plot([50.75, 57.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(56.05, 0.922, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a vertical release position between 53.76 and 54.37, so we zoom in on that region:

In [135…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=release_pos_y_stats[
    (53.2 < release_pos_y_stats['release_pos_y']) &
    (release_pos_y_stats['release_pos_y'] < 55)
    ],
    x='release_pos_y',
    y='correct_call_percent')
plt.plot([53.76, 53.76], [0.9135, 0.9225], color='red', linewidth=2)
plt.plot([54.06, 54.06], [0.9135, 0.9225], color='black', linewidth=2)
plt.plot([54.37, 54.37], [0.9135, 0.9225], color='red', linewidth=2)
plt.text(53.66, 0.9215, '25%', color='red', fontsize='x-large')
plt.text(54.08, 0.9215, '50%', color='black', fontsize='x-large')
plt.text(54.28, 0.9215, '75%', color='red', fontsize='x-large')
plt.plot([53.25, 54.95], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(54.51, 0.9167, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```

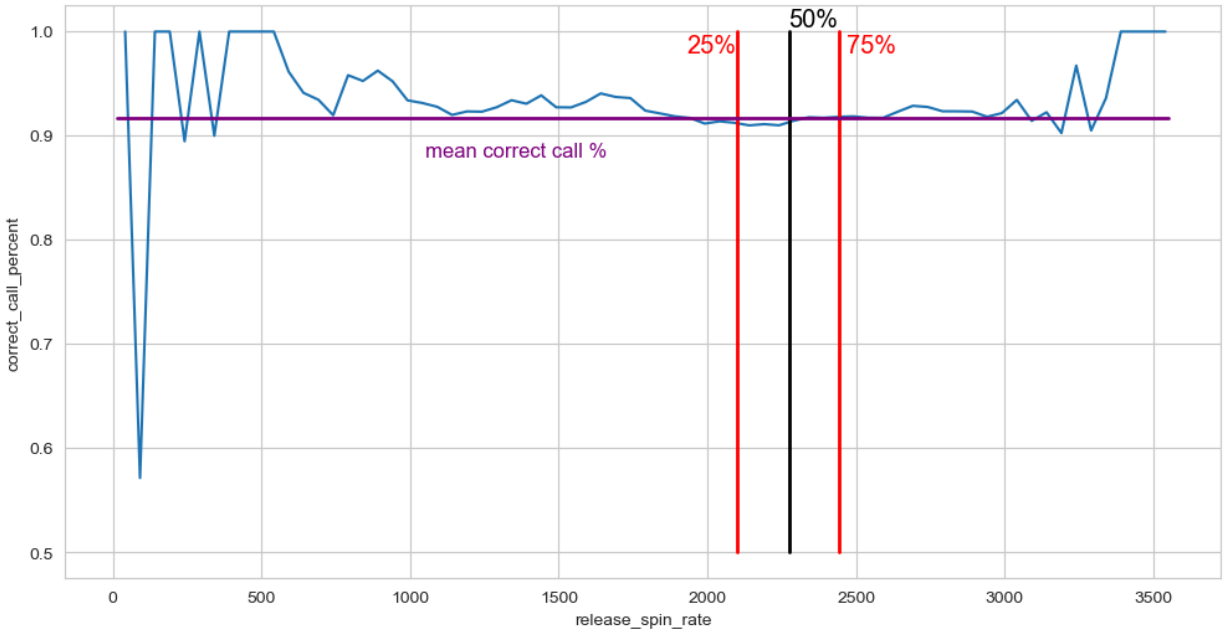


In [136…
```
# For summary statistics, run the following:
# all_features_df.release_pos_y.describe()
```

3.1.3.5 - `release_spin_rate`

We split our data into buckets with a constant width of 50.

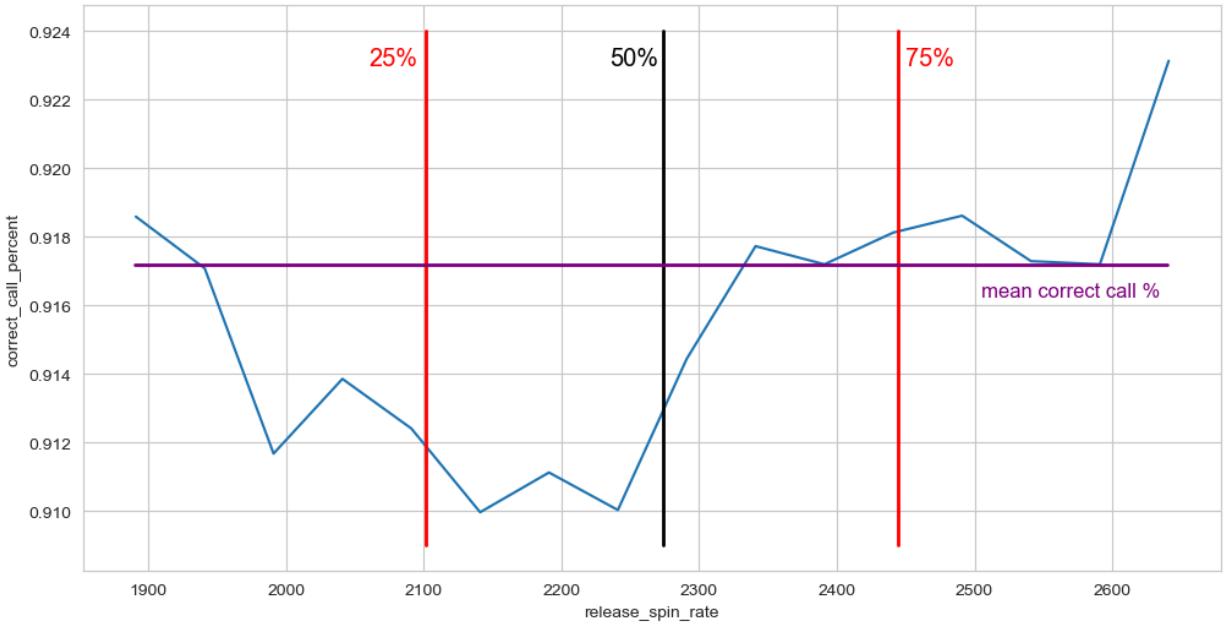
In [137…
```
release_spin_rate_percents = []
release_spin_rate_index = []

for index in range(16,3531,50):
    temp_list = all_features_df[
        (index <= all_features_df['release_spin_rate']) &
        (all_features_df['release_spin_rate'] < index+50)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        release_spin_rate_percents.append(temp_list[0])
        release_spin_rate_index.append(index+25)

release_spin_rate_stats = pd.DataFrame({
    'release_spin_rate':release_spin_rate_index,
    'correct_call_percent':release_spin_rate_percents
    })

plt.figure(figsize=(12,6))
sns.lineplot(data=release_spin_rate_stats, x='release_spin_rate', y='correct_call_perc
plt.plot([2102, 2102], [0.5, 1], color='red', linewidth=2)
plt.plot([2274, 2274], [0.5, 1], color='black', linewidth=2)
plt.plot([2444, 2444], [0.5, 1], color='red', linewidth=2)
plt.text(1930, 0.98, '25%', color='red', fontsize='x-large')
plt.text(2274, 1.005, '50%', color='black', fontsize='x-large')
plt.text(2470, 0.98, '75%', color='red', fontsize='x-large')
plt.plot([15, 3550], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(1050, 0.88, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a release spin rate between 2102 and 2444, so we zoom in on that region:

In [138…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=release_spin_rate_stats[
    (1850 < release_spin_rate_stats['release_spin_rate']) &
    (release_spin_rate_stats['release_spin_rate'] < 2650)
    ],
    x='release_spin_rate',
    y='correct_call_percent')
plt.plot([2102, 2102], [0.909, 0.924], color='red', linewidth=2)
plt.plot([2274, 2274], [0.909, 0.924], color='black', linewidth=2)
plt.plot([2444, 2444], [0.909, 0.924], color='red', linewidth=2)
plt.text(2060, 0.923, '25%', color='red', fontsize='x-large')
plt.text(2235, 0.923, '50%', color='black', fontsize='x-large')
plt.text(2450, 0.923, '75%', color='red', fontsize='x-large')
plt.plot([1890, 2640], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(2505, 0.91625, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [139…

```
# For summary statistics, run the following:
# all_features_df.release_spin_rate.describe()
```

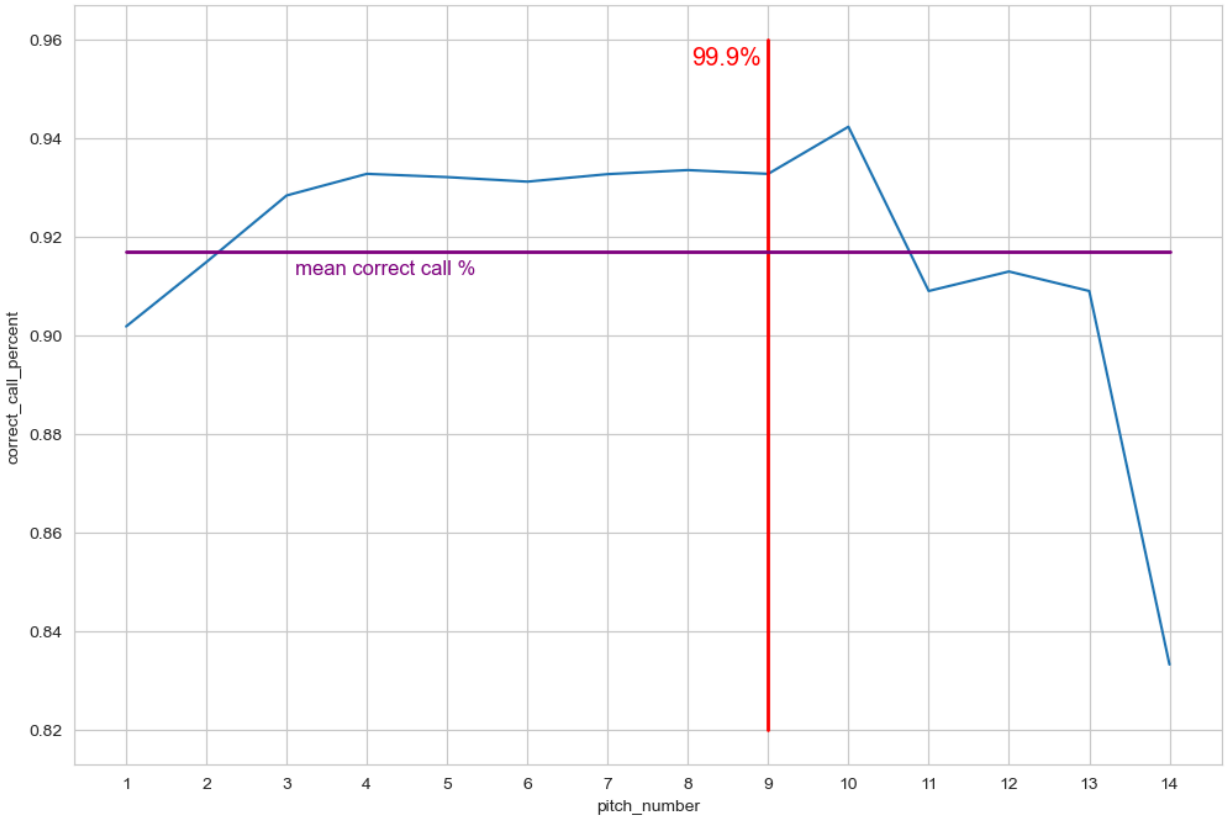
3.1.4 - Plate Appearance

3.1.4.1 - `pitch_number`

We note that 99.9% of all pitches in this data frame occur as a 9th pitch in an at-bat or earlier.

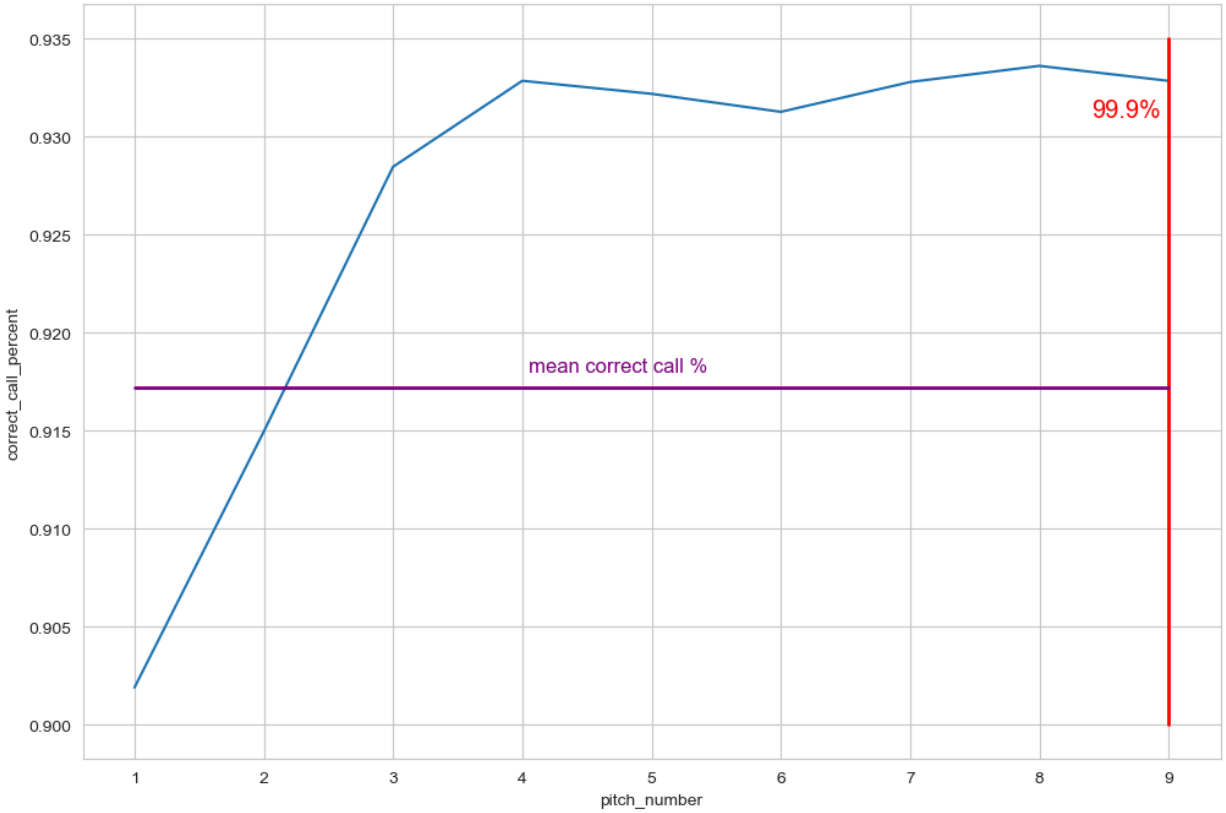
In [140…

```
pitch_number_percents = []
pitch_number_index = []

for index in range(1,15):
    temp_list = all_features_df[all_features_df['pitch_number']==index].correct_call.v
    pitch_number_percents.append(temp_list[0])
    pitch_number_index.append(index)

pitch_number_stats = pd.DataFrame({'pitch_number':pitch_number_index, 'correct_call_pe

plt.figure(figsize=(12,8))
sns.lineplot(data=pitch_number_stats, x='pitch_number', y='correct_call_percent').set_
plt.plot([9, 9], [0.82, 0.96], color='red', linewidth=2)
plt.text(8.05, 0.955, '99.9%', color='red', fontsize='x-large')
plt.plot([1, 14], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(3.1, 0.9125, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [141…

```
plt.figure(figsize=(12,8))
sns.lineplot(data=pitch_number_stats[
    pitch_number_stats['pitch_number'] <= 9
    ],
    x='pitch_number',
    y='correct_call_percent'
    ).set_xticks(range(1,15))
plt.plot([9, 9], [0.9, 0.935], color='red', linewidth=2)
plt.text(8.4, 0.931, '99.9%', color='red', fontsize='x-large')
plt.plot([1, 9], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(4.05, 0.918, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [142…

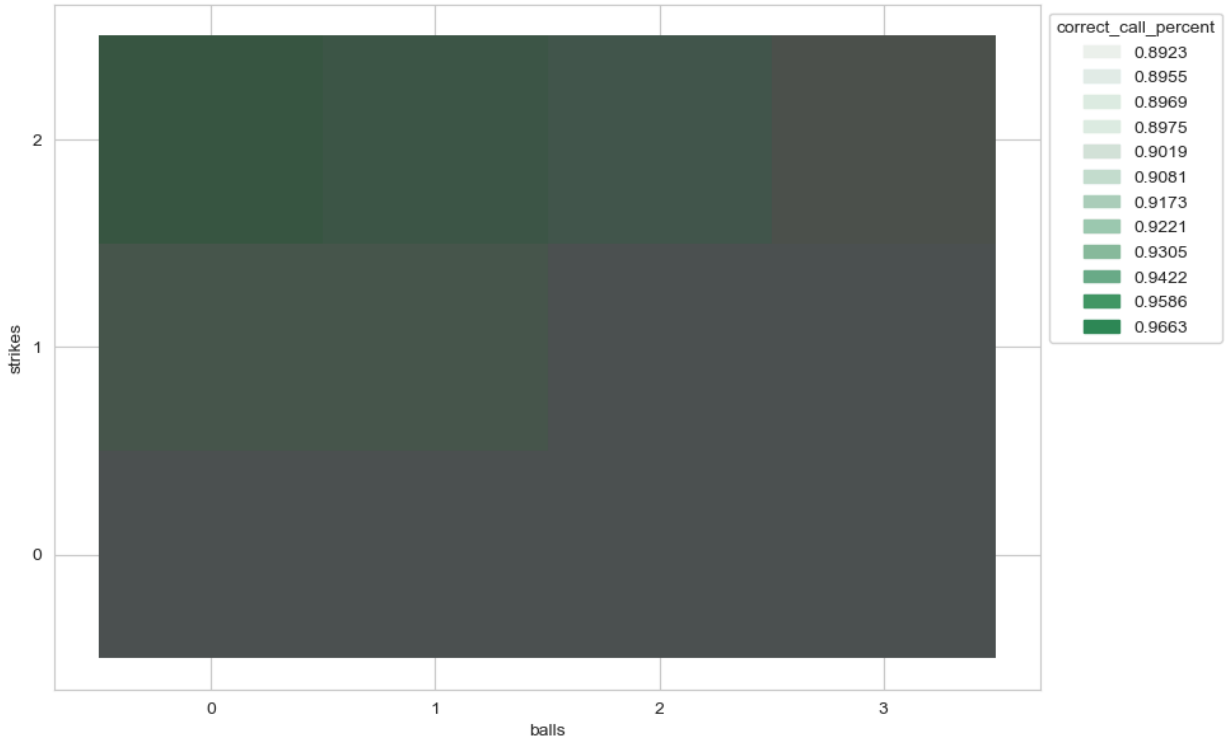
```
# The following code produces a visualization of number of pitches with given pitch nu

#pitch_number_count = []
# uses pitch_number_index from previous cell

#for index in range(1,15):
#    pitch_number_count.append(all_features_df[all_features_df['pitch_number']==index]

#pitch_number_count_stats = pd.DataFrame({
#    'pitch_number':pitch_number_index,
#    'occurrences':pitch_number_count
#    })

#up_to_9_pitch_number = all_features_df[all_features_df['pitch_number'] <= 9].correct_
#over_9_pitch_number = all_features_df[all_features_df['pitch_number'] > 9].correct_ca

#percent_9_pitch_percent = up_to_9_pitch_number / (up_to_9_pitch_number + over_9_pitch

#plt.figure(figsize=(12,6))
#sns.lineplot(data=pitch_number_count_stats, x='pitch_number', y='occurrences').set_xt
#plt.plot([9, 9], [0, 125000], color='red', linewidth=2)
#plt.text(7.9, 121000, str(round(percent_9_pitch_percent, 3))+'%', color='red', fontsi
#plt.show()
```

3.1.4.2 - Count (`balls` and `strikes`)

Instead of investigating `balls` and `strikes` independently, we instead combine them into the count for that pitch.

In [143…

```
count_percents = []
balls_index = []
strikes_index = []

for ball_index in range(0,4):
    for strike_index in range(0,3):
        temp_list = all_features_df[(all_features_df['balls'] == ball_index)
                                    & (all_features_df['strikes'] == strike_index)
                                    ].correct_call.value_counts(normalize=True).to_list
        count_percents.append(round(temp_list[0],4))
        balls_index.append(str(ball_index))
        strikes_index.append(str(strike_index))

count_stats = pd.DataFrame({'balls':balls_index, 'strikes':strikes_index, 'correct_cal

count_stats['strikes'] = pd.Categorical(count_stats['strikes'], ['2', '1', '0'])

plt.figure(figsize=(10,7))
ax = sns.histplot(data=count_stats, x='balls', y='strikes', hue='correct_call_percent'
                  palette=sns.light_palette('seagreen', as_cmap=True))
sns.move_legend(ax, "upper left", bbox_to_anchor=(1, 1))
plt.show()
```

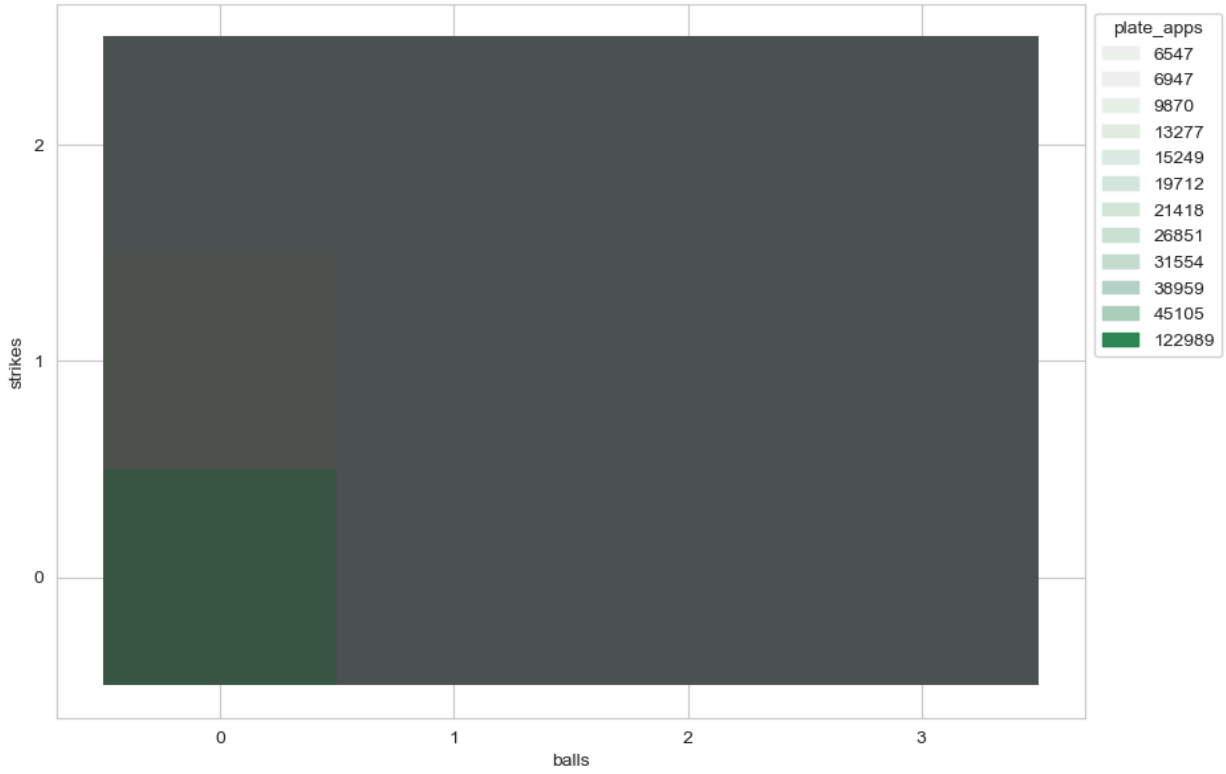


We also provide a visualization for how often a given count occurs. Understandably, the most common count by far is 0-0.

In [144…

```
count_pas = []
balls_index = []
strikes_index = []

for ball_index in range(0,4):
    for strike_index in range(0,3):
        count_pas.append(all_features_df[(all_features_df['balls'] == ball_index)
                                    & (all_features_df['strikes'] == strike_index)
```

```
                                    ].correct_call.count())
        balls_index.append(str(ball_index))
        strikes_index.append(str(strike_index))

count_pa_stats = pd.DataFrame({'balls':balls_index, 'strikes':strikes_index, 'plate_ap

count_pa_stats['strikes'] = pd.Categorical(count_stats['strikes'], ['2', '1', '0'])


plt.figure(figsize=(10,7))
ax = sns.histplot(data=count_pa_stats, x='balls', y='strikes', hue='plate_apps',
                  palette=sns.light_palette('seagreen', as_cmap=True))
sns.move_legend(ax, "upper left", bbox_to_anchor=(1, 1))
plt.show()
```



3.1.5 - Game State

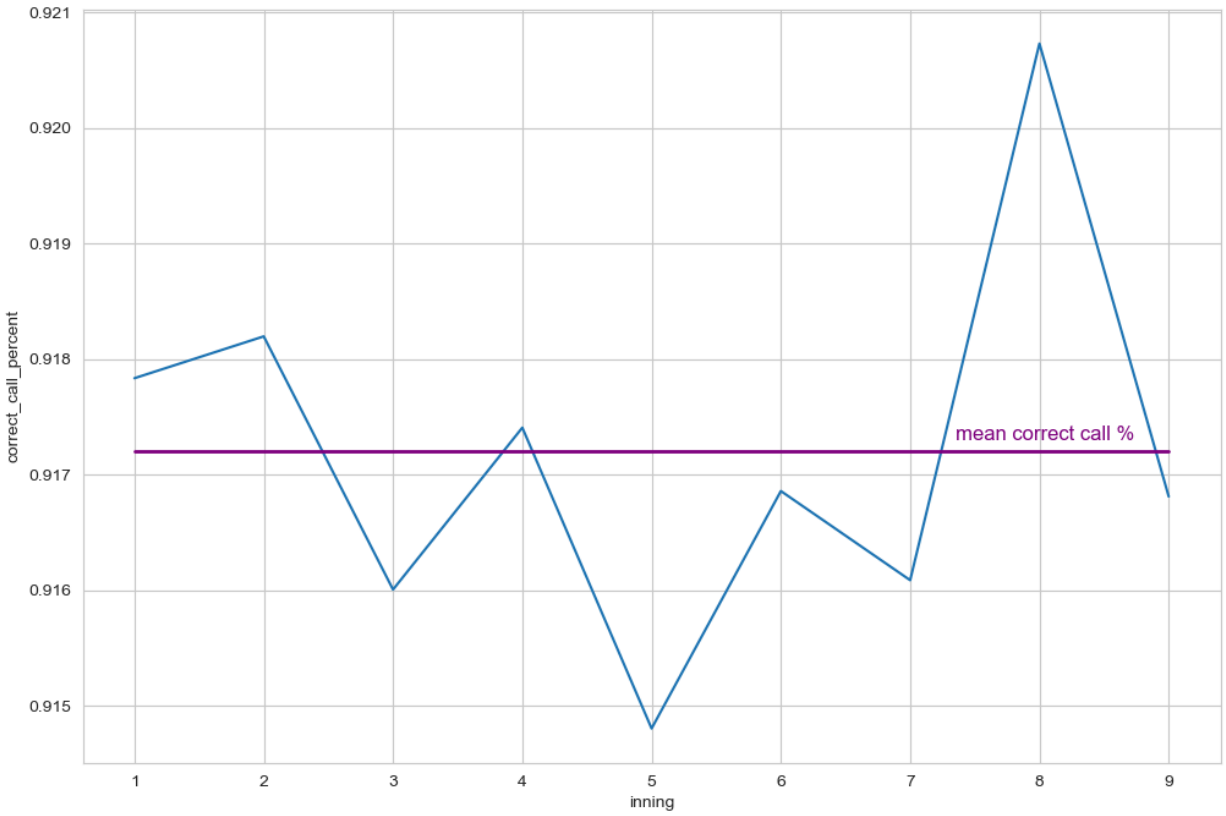
3.1.5.1 - Place in Game - `inning` and `outs_when_up`

We begin by examining pitches by `inning` and `outs_when_up` independently and then jointly.

In [145…

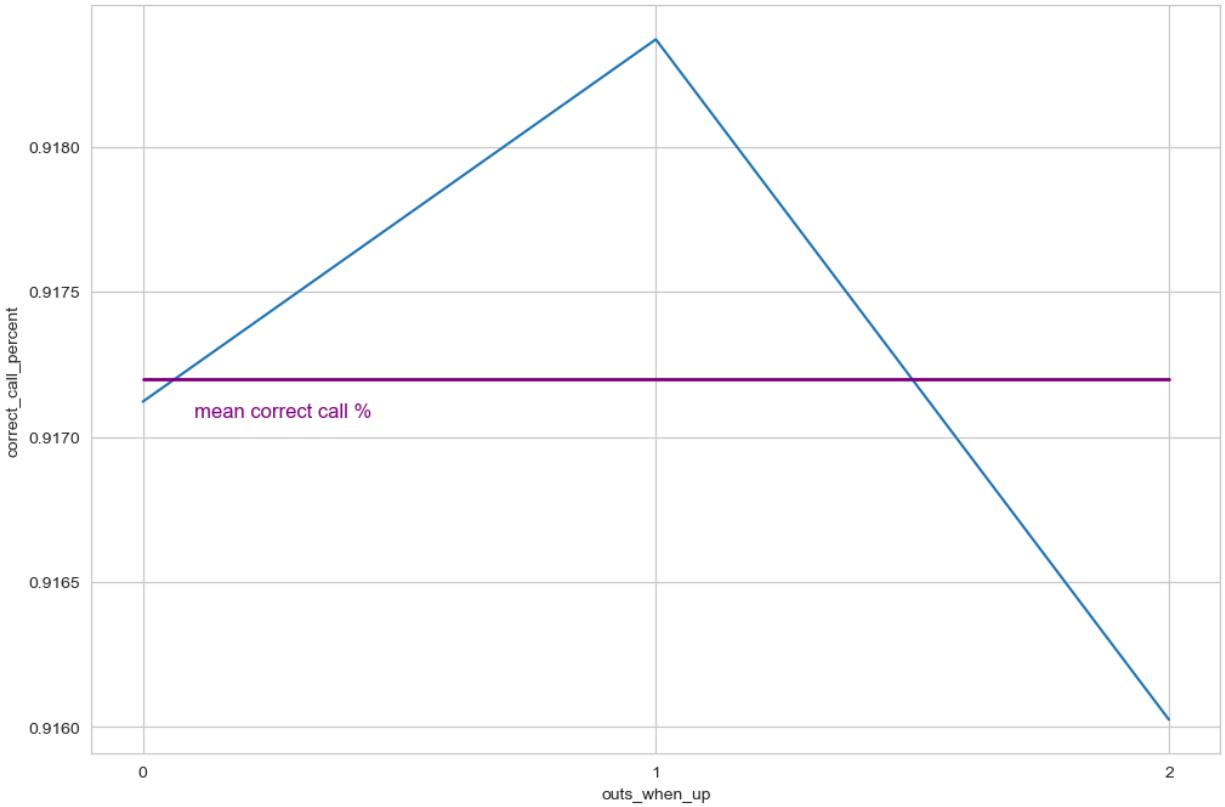
```
inning_percents = []
inning_index = []

for index in range(1,10):
    temp_list = all_features_df[all_features_df['inning']==index].correct_call.value_c
    inning_percents.append(temp_list[0])
    inning_index.append(index)

inning_stats = pd.DataFrame({'inning':inning_index, 'correct_call_percent':inning_perc
```

```
plt.figure(figsize=(12,8))
sns.lineplot(data=inning_stats, x='inning', y='correct_call_percent')
plt.plot([1, 9], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(7.35, 0.9173, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [146…

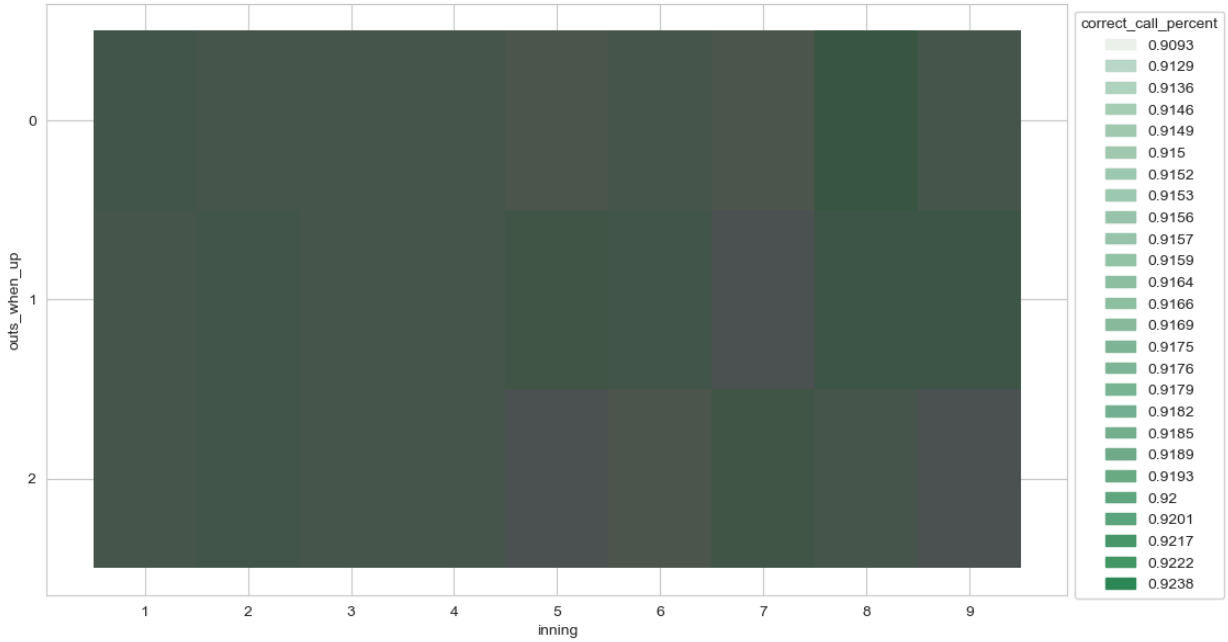
```
outs_when_up_percents = []
outs_when_up_index = []

for index in range(0,3):
    temp_list = all_features_df[all_features_df['outs_when_up']==index].correct_call.v
    outs_when_up_percents.append(temp_list[0])
    outs_when_up_index.append(index)

outs_when_up_stats = pd.DataFrame({'outs_when_up':outs_when_up_index, 'correct_call_pe

plt.figure(figsize=(12,8))
sns.lineplot(data=outs_when_up_stats, x='outs_when_up', y='correct_call_percent').set_
plt.plot([0, 2], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(0.1, 0.91707, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [147…

```
# note that inning_index and outs_when_up_index replace lists from above two cells, bu
# additionally, lists are not needed outside of any individual cell

game_state_percents = []
inning_index = []
outs_when_up_index = []

for inning_counter in range(1,10):
    for outs_when_up_counter in range(0,3):
        temp_list = all_features_df[
            (all_features_df['inning'] == inning_counter) &
            (all_features_df['outs_when_up'] == outs_when_up_counter)
            ].correct_call.value_counts(normalize=True).to_list()
        game_state_percents.append(round(temp_list[0],4))
        inning_index.append(str(inning_counter))
        outs_when_up_index.append(str(outs_when_up_counter))

game_state_stats = pd.DataFrame({
    'inning':inning_index,
    'outs_when_up':outs_when_up_index,
    'correct_call_percent':game_state_percents
    })

#game_state_stats['outs_when_up'] = pd.Categorical(count_stats['outs_when_up'], ['2',

plt.figure(figsize=(12,7))
ax = sns.histplot(
    data=game_state_stats,
    x='inning',
    y='outs_when_up',
    hue='correct_call_percent',
    palette=sns.light_palette('seagreen', as_cmap=True)
    )
```

```
sns.move_legend(ax, "upper left", bbox_to_anchor=(1, 1))
plt.show()
```

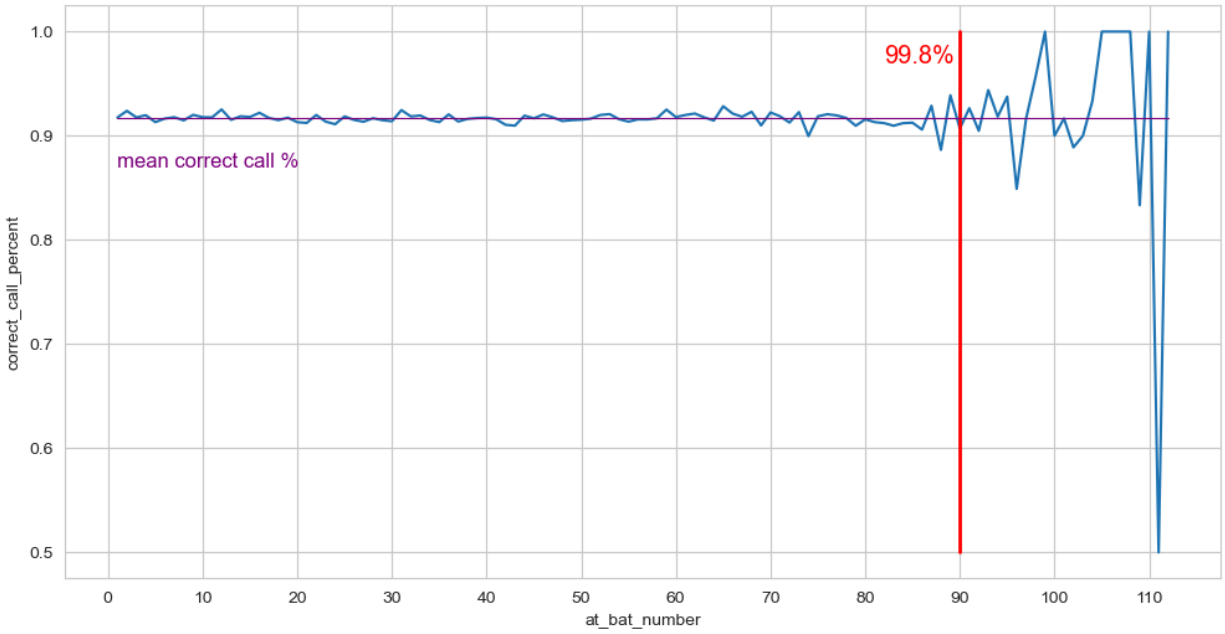


3.1.5.2 - `at_bat_number`

Note that 99.8% of pitches occur within the first 90 plate appearances of a game.

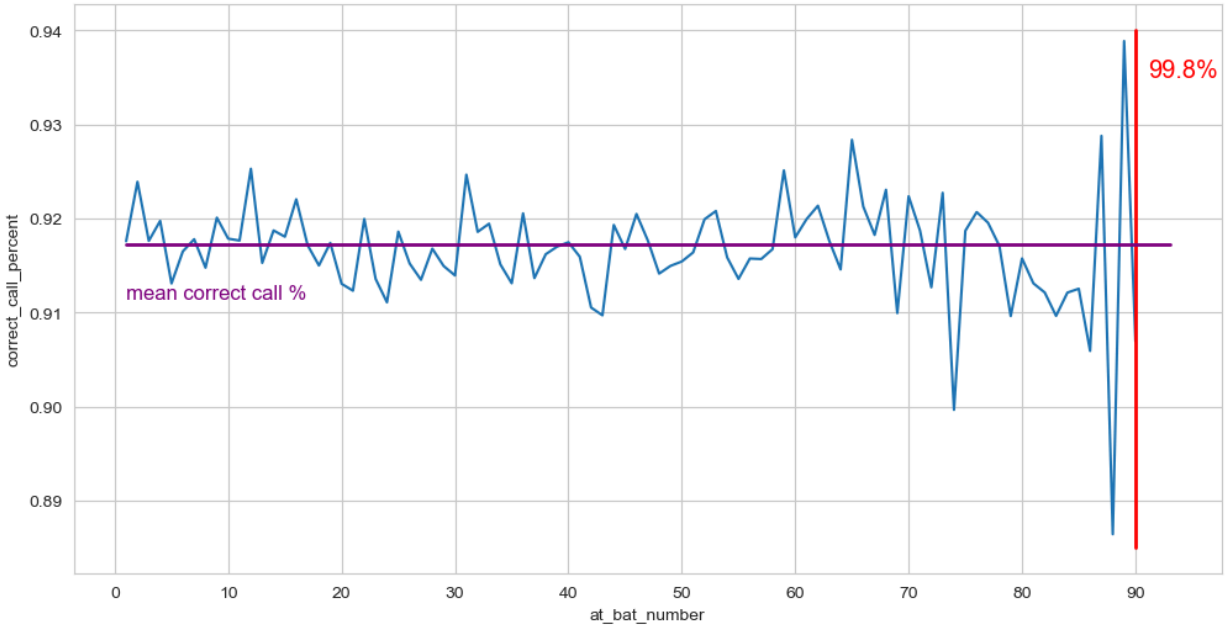
In [148…

```
at_bat_number_percents = []
at_bat_number_index = []

for index in range(1,113):
    temp_list = all_features_df[all_features_df['at_bat_number']==index].correct_call.
    at_bat_number_percents.append(temp_list[0])
    at_bat_number_index.append(index)

at_bat_number_stats = pd.DataFrame({'at_bat_number':at_bat_number_index, 'correct_call

plt.figure(figsize=(12,6))
sns.lineplot(data=at_bat_number_stats, x='at_bat_number', y='correct_call_percent').se
plt.plot([90, 90], [0.5, 1], color='red', linewidth=2)
plt.text(82, 0.97, '99.8%', color='red', fontsize='x-large')
plt.plot([1, 112], [0.9172, 0.9172], color='purple', linewidth=0.8)
plt.text(1, 0.87, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [149…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=at_bat_number_stats[at_bat_number_stats['at_bat_number'] < 91],
             x='at_bat_number',
             y='correct_call_percent').set_xticks(range(0,120,10))
plt.plot([90, 90], [0.885, 0.94], color='red', linewidth=2)
plt.text(91.1, 0.935, '99.8%', color='red', fontsize='x-large')
plt.plot([1, 93], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(1, 0.9115, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [150…

```
#at_bat_number_count = []
## uses at_bat_number_index from previous cell

#for index in range(1,113):
#    at_bat_number_count.append(all_features_df[all_features_df['at_bat_number']==inde

#at_bat_number_count_stats = pd.DataFrame({
#    'at_bat_number':at_bat_number_index,
```

```
#    'occurrences':at_bat_number_count
#    })

#up_to_90_at_bats = all_features_df[all_features_df['at_bat_number'] <= 90].correct_ca
#over_90_at_bats = all_features_df[all_features_df['at_bat_number'] > 90].correct_call

#percent_90_at_bats = up_to_90_at_bats / (up_to_90_at_bats + over_90_at_bats)

#plt.figure(figsize=(12,6))
#sns.lineplot(data=at_bat_number_count_stats, x='at_bat_number', y='occurrences').set_
#plt.plot([90, 90], [0, 5000], color='red', linewidth=2)
#plt.text(80.5, 4700, str(round(percent_90_at_bats, 3))+'%', color='red', fontsize='x-
#plt.show()
```

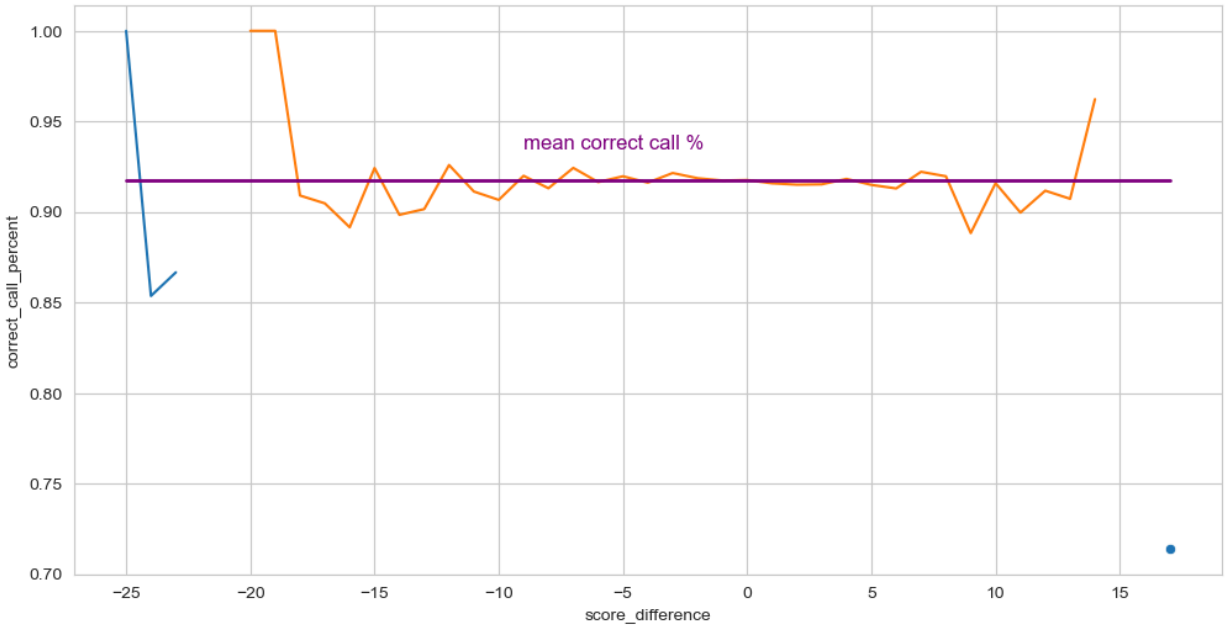
3.1.5.3 - Score Difference (`home_score` - `away_score`)

Instead of considering each team's score individually, we instead consider the relative game score by looking at `home_score` - `away_score`

In [151…

```
score_difference_percents1 = []
score_difference_index1 = []

score_difference_percents2 = []
score_difference_index2 = []

score_difference_percents3 = []
score_difference_index3 = []

for index in range(-25,-22):
    temp_list = all_features_df[
        all_features_df['home_score']-all_features_df['away_score'] == index
        ].correct_call.value_counts(normalize=True).to_list()
    score_difference_percents1.append(temp_list[0])
    score_difference_index1.append(index)

for index in range(-20,15):
    temp_list = all_features_df[
        all_features_df['home_score']-all_features_df['away_score'] == index
        ].correct_call.value_counts(normalize=True).to_list()
    score_difference_percents2.append(temp_list[0])
    score_difference_index2.append(index)


for index in range(17,18):
    temp_list = all_features_df[
        all_features_df['home_score']-all_features_df['away_score'] == index
        ].correct_call.value_counts(normalize=True).to_list()
    score_difference_percents3.append(temp_list[0])
    score_difference_index3.append(index)

score_difference_stats1 = pd.DataFrame({
    'score_difference':score_difference_index1,
    'correct_call_percent':score_difference_percents1
    })

score_difference_stats2 = pd.DataFrame({
    'score_difference':score_difference_index2,
```

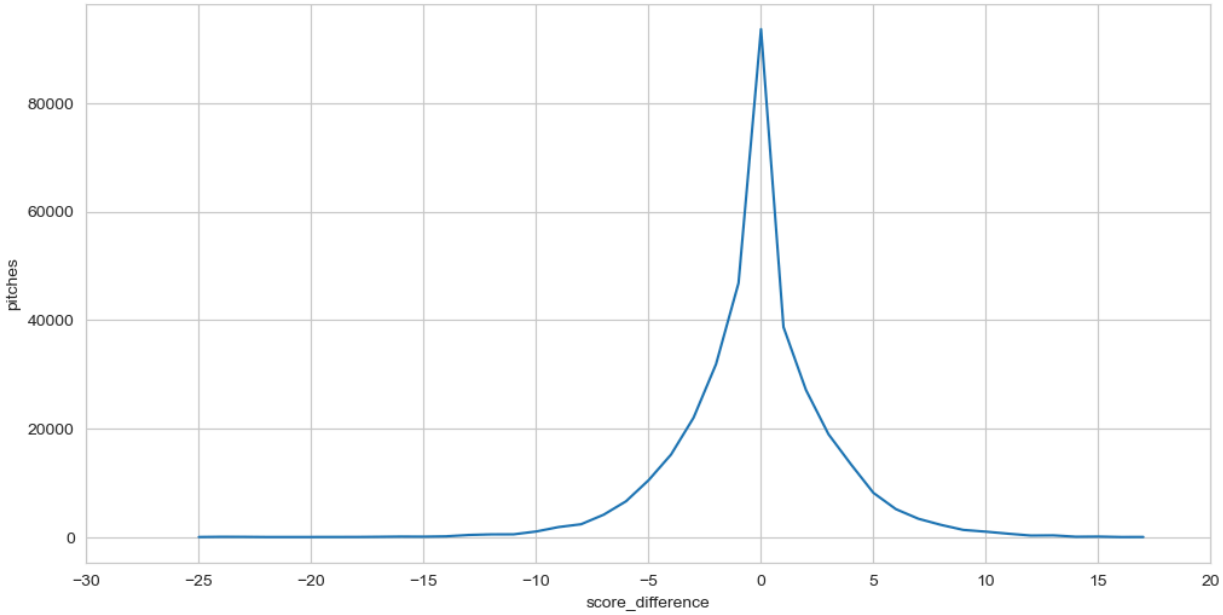
```
    'correct_call_percent':score_difference_percents2
    })

score_difference_stats3 = pd.DataFrame({
    'score_difference':score_difference_index3,
    'correct_call_percent':score_difference_percents3
    })


plt.figure(figsize=(12,6))
sns.lineplot(data=score_difference_stats1, x='score_difference', y='correct_call_perce
sns.lineplot(data=score_difference_stats2, x='score_difference', y='correct_call_perce
sns.scatterplot(data=score_difference_stats3, x='score_difference', y='correct_call_pe
plt.plot([-25, 17], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-9, 0.935, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



We now look at how many pitches occur with a given score difference:

In [152…

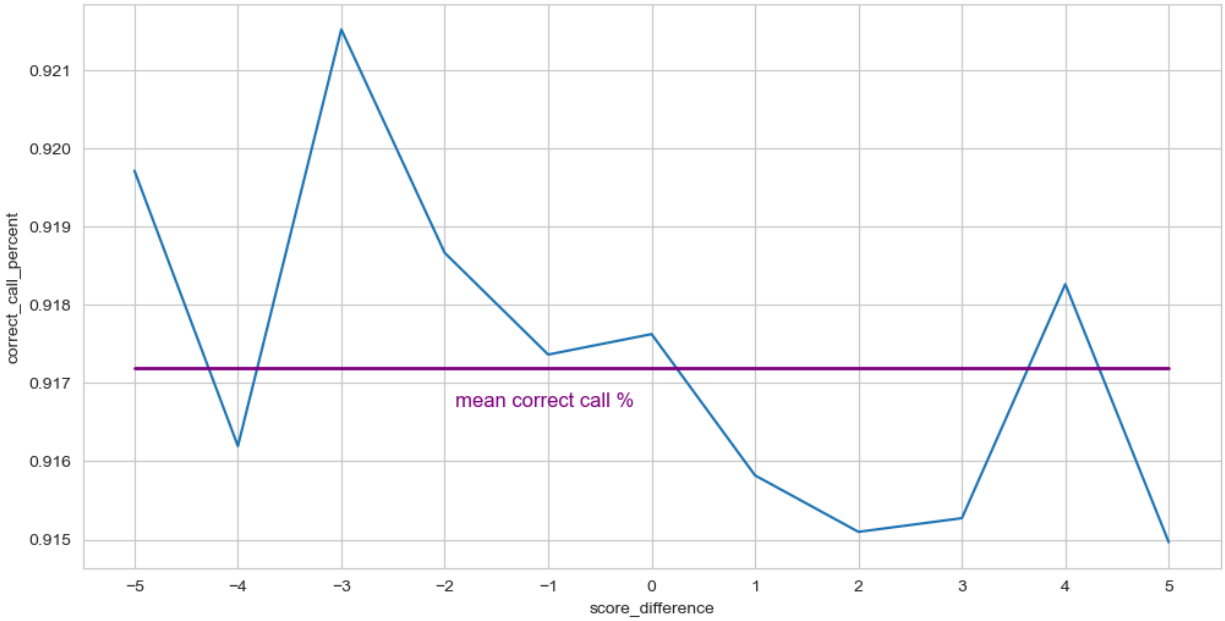
```
score_difference_counts_index = []
score_difference_counts_pitches = []

for index in range(-25,18):
    score_difference_counts_pitches.append(all_features_df[
        all_features_df['home_score']-all_features_df['away_score'] == index
        ].binary_bs.count()
        )
    score_difference_counts_index.append(index)

score_difference_counts_stats = pd.DataFrame({
    'score_difference':score_difference_counts_index,
    'pitches':score_difference_counts_pitches
    })

plt.figure(figsize=(12,6))
sns.lineplot(data=score_difference_counts_stats, x='score_difference', y='pitches').se
plt.show()
```



Consequently, we focus on pitches with a score difference between -5 and 5:

In [153…

```
plt.figure(figsize=(12,6))
sns.lineplot(
    data=score_difference_stats2[
        (-6 < score_difference_stats2['score_difference']) &
        (score_difference_stats2['score_difference'] < 6)
    ],
    x='score_difference',
    y='correct_call_percent'
    ).set_xticks((range(-5,6,1)))
plt.plot([-5, 5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-1.9, 0.9167, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



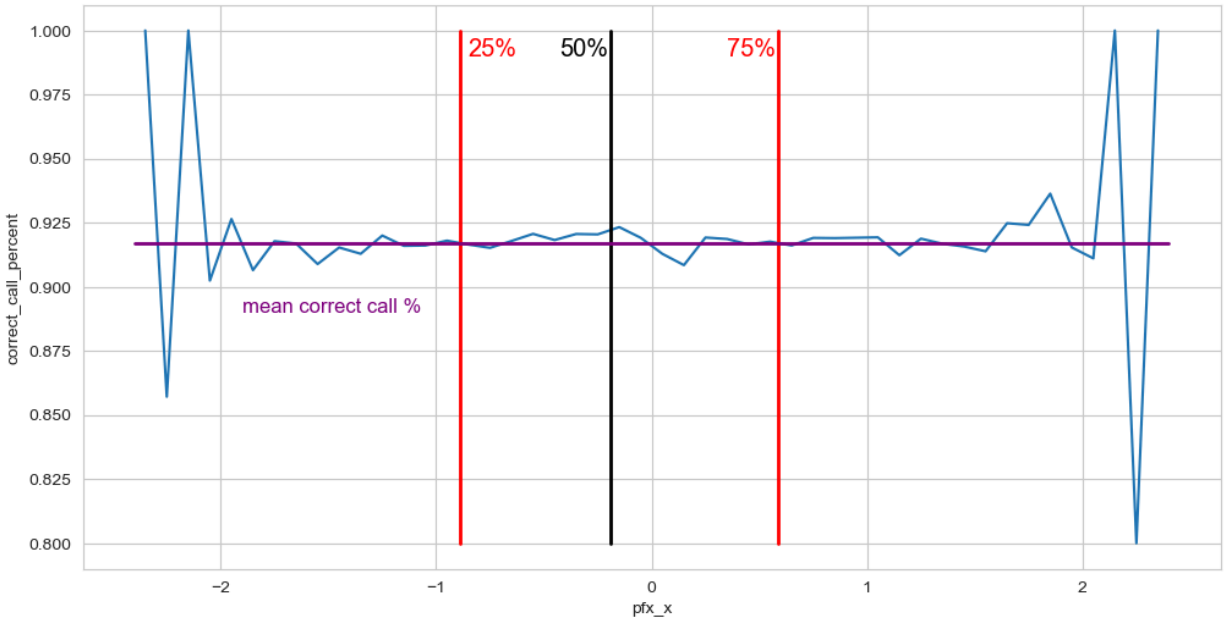
3.1.6 - Movement

3.1.6.1 - pfx_x

We split our data into buckets with a constant width of 0.1.

In [154…

```
pfx_x_percents = []
pfx_x_index = []

for index in range(-240,237,10):
    temp_list = all_features_df[
        (index*0.01 <= all_features_df['pfx_x']) &
        (all_features_df['pfx_x'] < (index*0.01)+0.1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        pfx_x_percents.append(temp_list[0])
        pfx_x_index.append(index*0.01+0.05)

pfx_x_stats = pd.DataFrame({'pfx_x':pfx_x_index, 'correct_call_percent':pfx_x_percents

plt.figure(figsize=(12,6))
sns.lineplot(data=pfx_x_stats, x='pfx_x', y='correct_call_percent')
plt.plot([-0.89, -0.89], [0.8, 1], color='red', linewidth=2)
plt.plot([-0.19, -0.19], [0.8, 1], color='black', linewidth=2)
plt.plot([0.59, 0.59], [0.8, 1], color='red', linewidth=2)
plt.text(-0.85, 0.99, '25%', color='red', fontsize='x-large')
plt.text(-0.43, 0.99, '50%', color='black', fontsize='x-large')
plt.text(0.35, 0.99, '75%', color='red', fontsize='x-large')
plt.plot([-2.4, 2.4], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-1.9, 0.89, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```

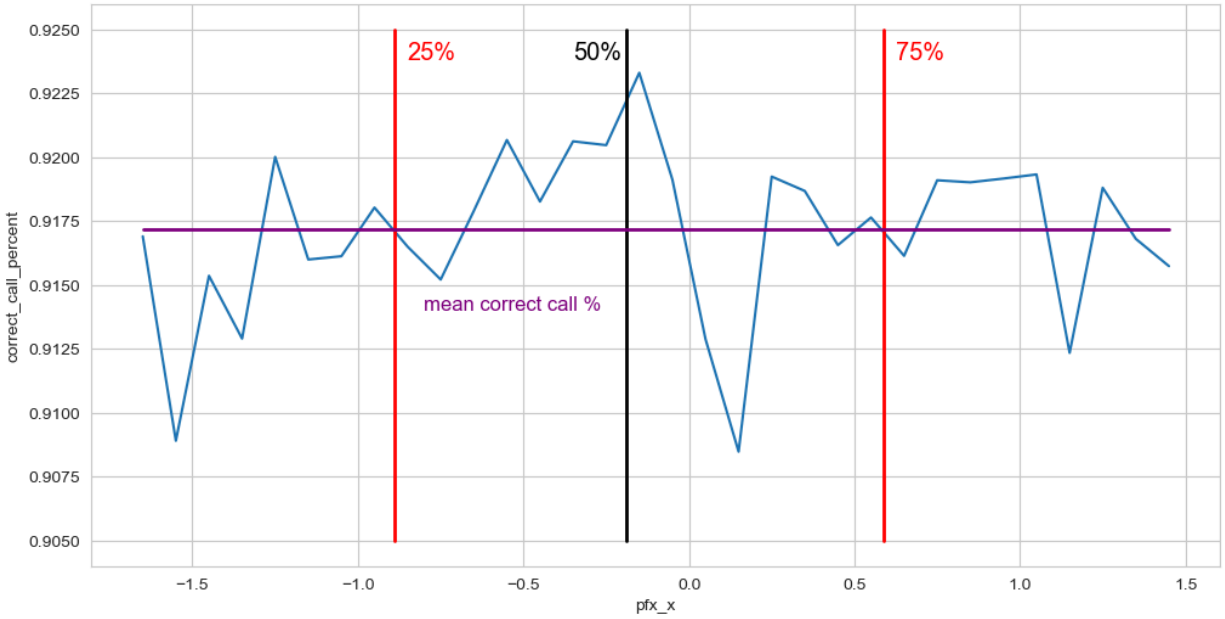


Note that 50% of all pitches occur with a horizontal movement between -0.89 and 0.59, so we zoom in on that region:

In [155…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=pfx_x_stats[
    (-1.7 < pfx_x_stats['pfx_x']) &
    (pfx_x_stats['pfx_x'] < 1.5)
```

```
    ],
    x='pfx_x',
    y='correct_call_percent'
    )
plt.plot([-0.89, -0.89], [0.905, 0.925], color='red', linewidth=2)
plt.plot([-0.19, -0.19], [0.905, 0.925], color='black', linewidth=2)
plt.plot([0.59, 0.59], [0.905, 0.925], color='red', linewidth=2)
plt.text(-0.85, 0.9238, '25%', color='red', fontsize='x-large')
plt.text(-0.35, 0.9238, '50%', color='black', fontsize='x-large')
plt.text(0.625, 0.9238, '75%', color='red', fontsize='x-large')
plt.plot([-1.65, 1.45], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-0.8, 0.914, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [156…

```
# For summary statistics, run the following:
# all_features_df.pfx_x.describe()
```

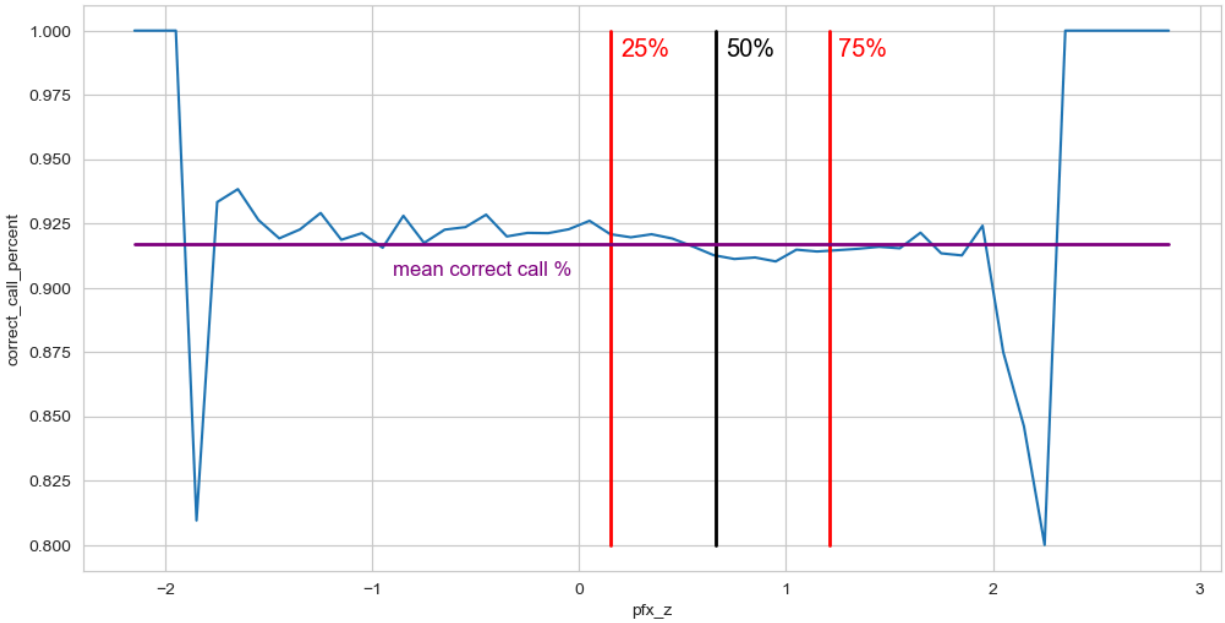
3.1.6.2 - pfx_z

We split our data into buckets with a constant width of 0.1.

In [157…

```
pfx_z_percents = []
pfx_z_index = []

for index in range(-220,284,10):
    temp_list = all_features_df[
        (index*0.01 <= all_features_df['pfx_z']) &
        (all_features_df['pfx_z'] < (index*0.01)+0.1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        pfx_z_percents.append(temp_list[0])
        pfx_z_index.append(index*0.01+0.05)

pfx_z_stats = pd.DataFrame({'pfx_z':pfx_z_index, 'correct_call_percent':pfx_z_percents

plt.figure(figsize=(12,6))
sns.lineplot(data=pfx_z_stats, x='pfx_z', y='correct_call_percent')
```

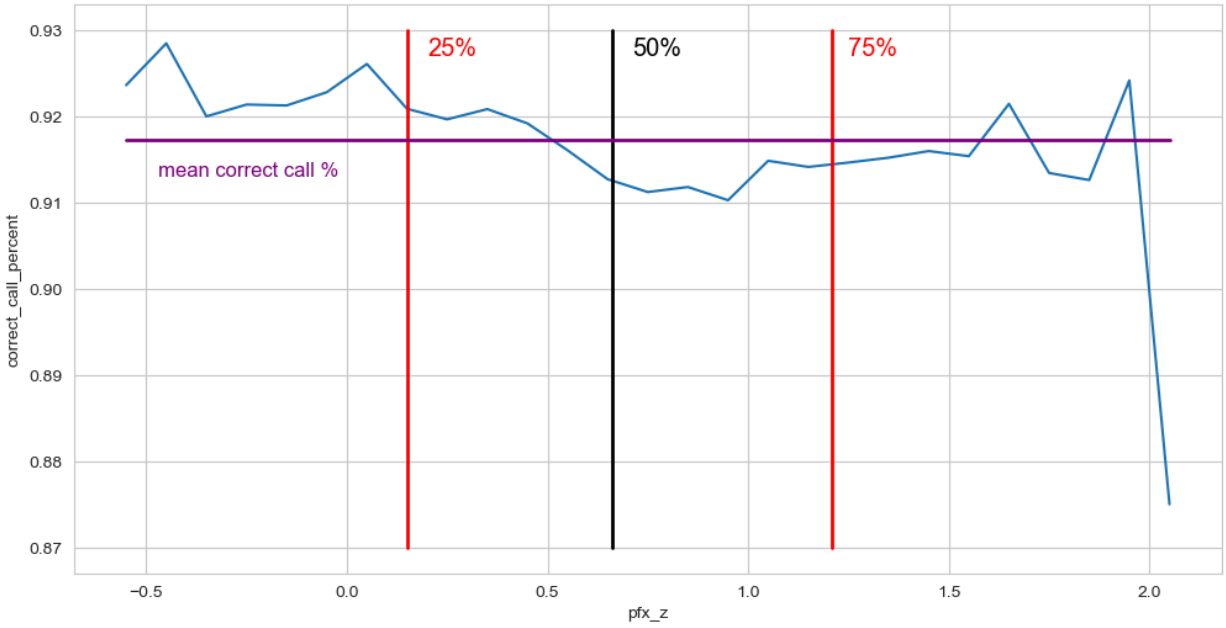
```
plt.plot([0.15, 0.15], [0.8, 1], color='red', linewidth=2)
plt.plot([0.66, 0.66], [0.8, 1], color='black', linewidth=2)
plt.plot([1.21, 1.21], [0.8, 1], color='red', linewidth=2)
plt.text(0.2, 0.99, '25%', color='red', fontsize='x-large')
plt.text(0.71, 0.99, '50%', color='black', fontsize='x-large')
plt.text(1.25, 0.99, '75%', color='red', fontsize='x-large')
plt.plot([-2.15, 2.85], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-0.9, 0.905, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a vertical movement between 0.15 and 1.21, so we zoom in on that region:

In [158…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=pfx_z_stats[
    (-0.6 < pfx_z_stats['pfx_z']) &
    (pfx_z_stats['pfx_z'] < 2.1)
    ],
    x='pfx_z',
    y='correct_call_percent'
    )
plt.plot([0.15, 0.15], [0.87, 0.93], color='red', linewidth=2)
plt.plot([0.66, 0.66], [0.87, 0.93], color='black', linewidth=2)
plt.plot([1.21, 1.21], [0.87, 0.93], color='red', linewidth=2)
plt.text(0.2, 0.927, '25%', color='red', fontsize='x-large')
plt.text(0.71, 0.927, '50%', color='black', fontsize='x-large')
plt.text(1.25, 0.927, '75%', color='red', fontsize='x-large')
plt.plot([-0.55, 2.05], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-0.47, 0.913, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [159…

```
# For summary statistics, run the following:
# all_features_df.pfx_z.describe()
```

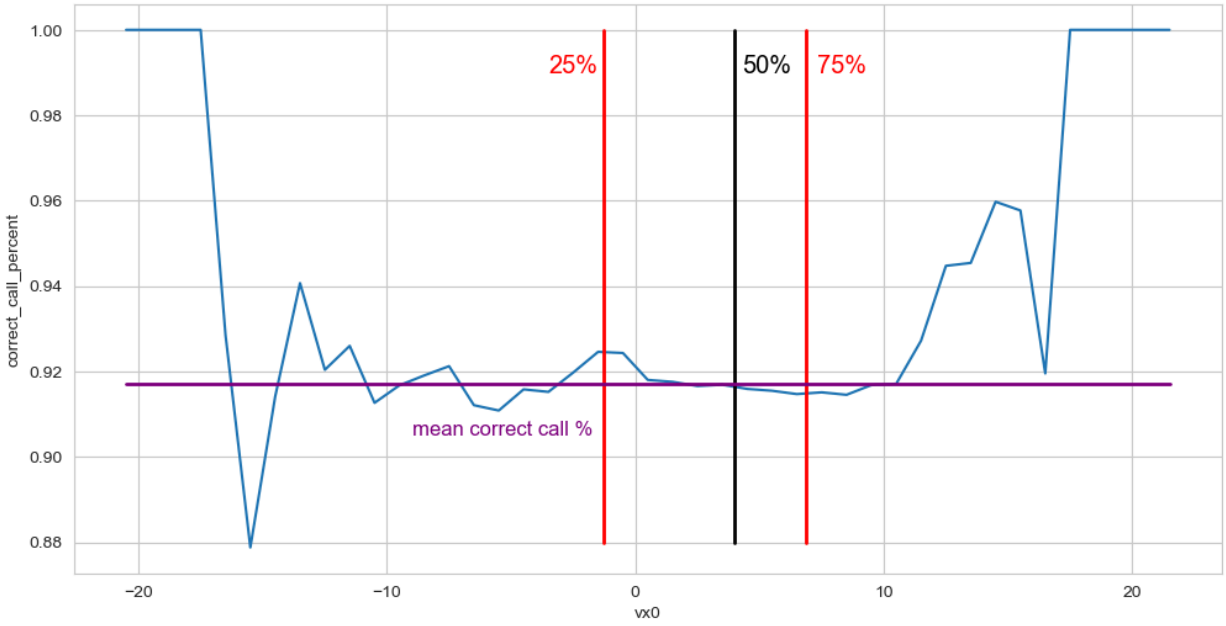
3.1.7- Velocity

3.1.7.1 - `vx0`

We split our data into buckets with a constant width of 1.

In [160…

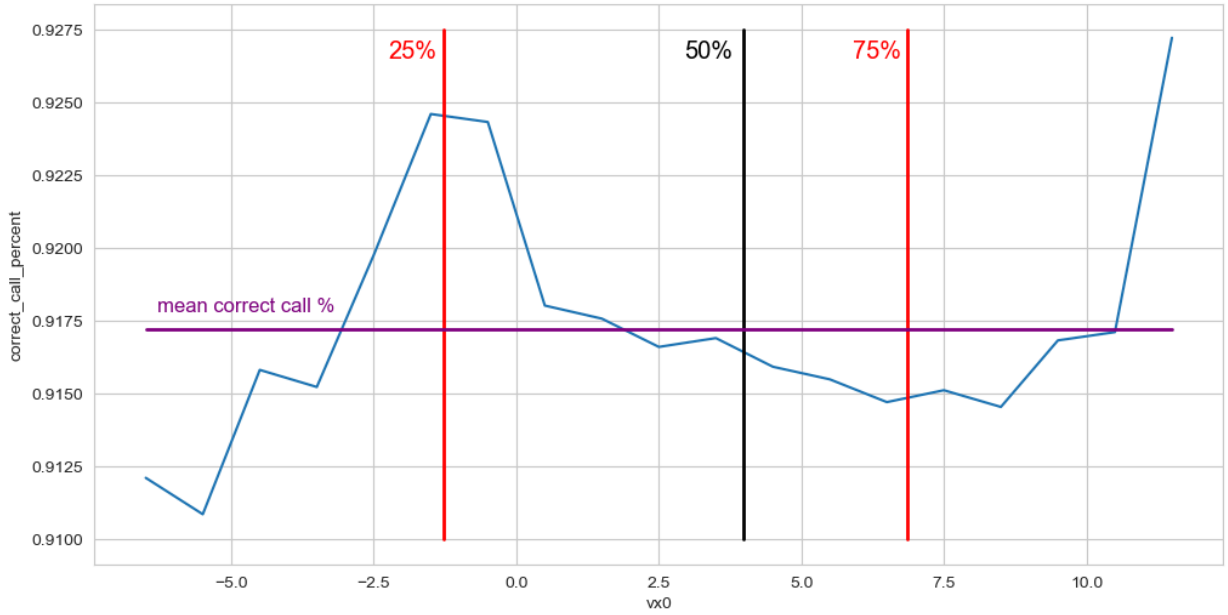
```
vx0_percents = []
vx0_index = []

for index in range(-210,215,10):
    temp_list = all_features_df[
        (index*0.1 <= all_features_df['vx0']) &
        (all_features_df['vx0'] < (index*0.1)+1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        vx0_percents.append(temp_list[0])
        vx0_index.append(index*0.1+0.5)

vx0_stats = pd.DataFrame({'vx0':vx0_index, 'correct_call_percent':vx0_percents})

plt.figure(figsize=(12,6))
sns.lineplot(data=vx0_stats, x='vx0', y='correct_call_percent')
plt.plot([-1.28, -1.28], [0.88, 1], color='red', linewidth=2)
plt.plot([3.98, 3.98], [0.88, 1], color='black', linewidth=2)
plt.plot([6.85, 6.85], [0.88, 1], color='red', linewidth=2)
plt.text(-3.5, 0.99, '25%', color='red', fontsize='x-large')
plt.text(4.3, 0.99, '50%', color='black', fontsize='x-large')
plt.text(7.3, 0.99, '75%', color='red', fontsize='x-large')
plt.plot([-20.5, 21.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-9, 0.905, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a horizontal velocity between -1.28 and 6.85, so we zoom in on that region:

In [161…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=vx0_stats[
    (-7 < vx0_stats['vx0']) &
    (vx0_stats['vx0'] < 12)
    ],
    x='vx0',
    y='correct_call_percent'
    )
plt.plot([-1.28, -1.28], [0.91, 0.9275], color='red', linewidth=2)
plt.plot([3.98, 3.98], [0.91, 0.9275], color='black', linewidth=2)
plt.plot([6.85, 6.85], [0.91, 0.9275], color='red', linewidth=2)
plt.text(-2.25, 0.9265, '25%', color='red', fontsize='x-large')
plt.text(2.95, 0.9265, '50%', color='black', fontsize='x-large')
plt.text(5.9, 0.9265, '75%', color='red', fontsize='x-large')
plt.plot([-6.5, 11.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-6.3, 0.9178, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```

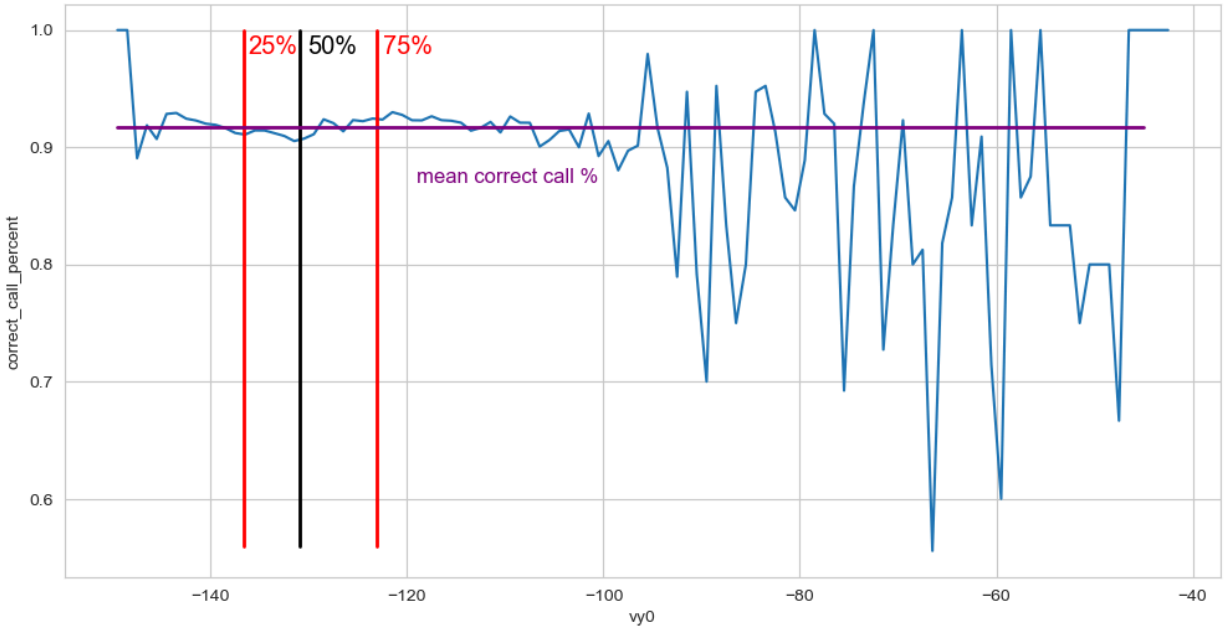


In [162…
```
# For summary statistics, run the following:
# all_features_df.vx0.describe()
```

3.1.7.2 - `vy0`

We split our data into buckets with a constant width of 1.

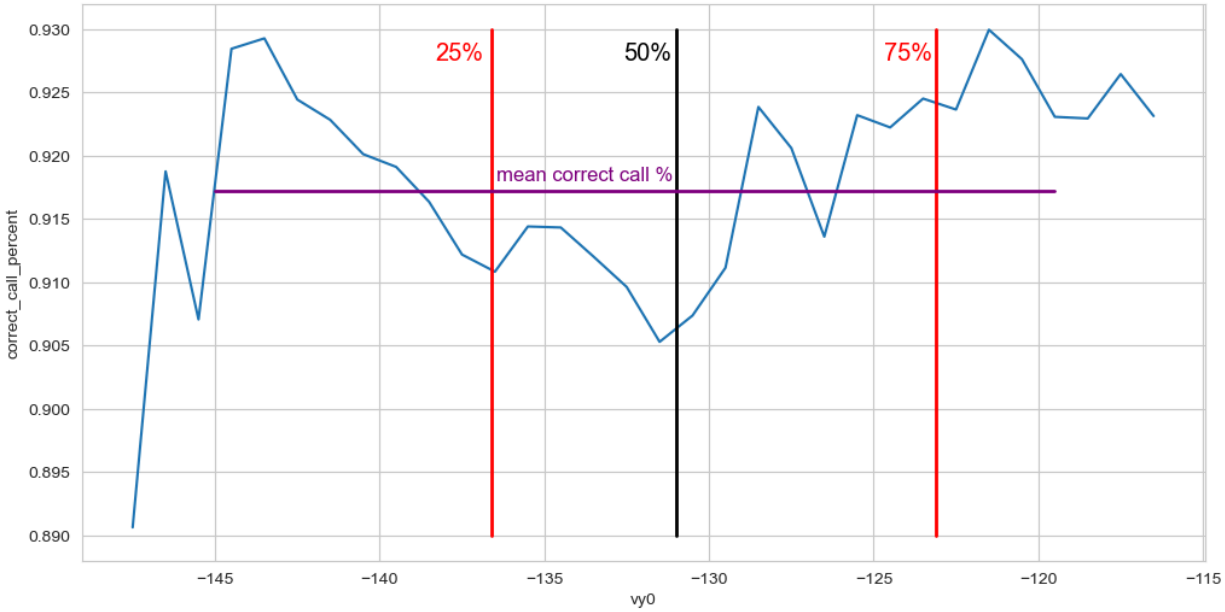
In [163…
```
vy0_percents = []
vy0_index = []

for index in range(-152,-44,1):
    temp_list = all_features_df[
        (index <= all_features_df['vy0']) &
        (all_features_df['vy0'] < index+1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        vy0_percents.append(temp_list[0])
        vy0_index.append(index+2.5)

vy0_stats = pd.DataFrame({'vy0':vy0_index, 'correct_call_percent':vy0_percents})

plt.figure(figsize=(12,6))
sns.lineplot(data=vy0_stats, x='vy0', y='correct_call_percent')
plt.plot([-136.6, -136.6], [0.56, 1], color='red', linewidth=2)
plt.plot([-131, -131], [0.56, 1], color='black', linewidth=2)
plt.plot([-123.1, -123.1], [0.56, 1], color='red', linewidth=2)
plt.text(-136.2, 0.98, '25%', color='red', fontsize='x-large')
plt.text(-130.2, 0.98, '50%', color='black', fontsize='x-large')
plt.text(-122.4, 0.98, '75%', color='red', fontsize='x-large')
plt.plot([-149.5, -45], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-119, 0.87, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a depth velocity between -136.6 and -123.1, so we zoom in on that region:

In [164…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=vy0_stats[
    (-148 < vy0_stats['vy0']) &
    (vy0_stats['vy0'] < -116)
    ],
    x='vy0',
    y='correct_call_percent'
    )
plt.plot([-136.6, -136.6], [0.89, 0.93], color='red', linewidth=2)
plt.plot([-131, -131], [0.89, 0.93], color='black', linewidth=2)
plt.plot([-123.1, -123.1], [0.89, 0.93], color='red', linewidth=2)
plt.text(-138.3, 0.9275, '25%', color='red', fontsize='x-large')
plt.text(-132.6, 0.9275, '50%', color='black', fontsize='x-large')
plt.text(-124.7, 0.9275, '75%', color='red', fontsize='x-large')
plt.plot([-145, -119.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-136.45, 0.918, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [165…

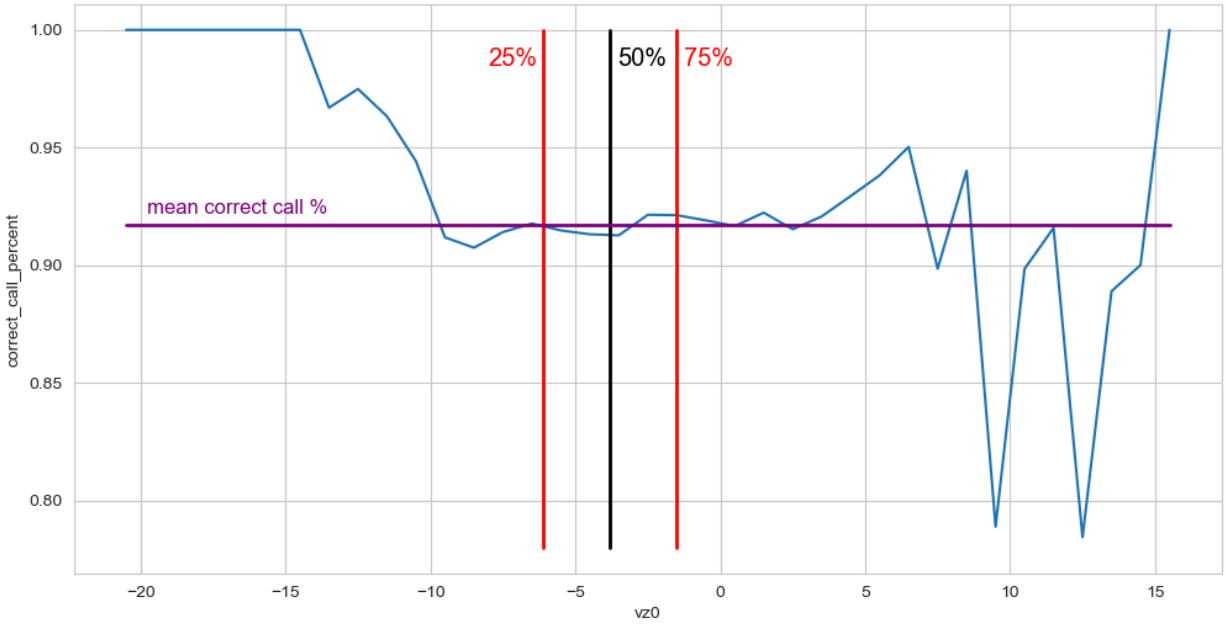
```
# For summary statistics, run the following:
# all_features_df.vy0.describe()
```

3.1.7.3 - `vz0`

We split our data into buckets with a constant width of 1.

In [166…

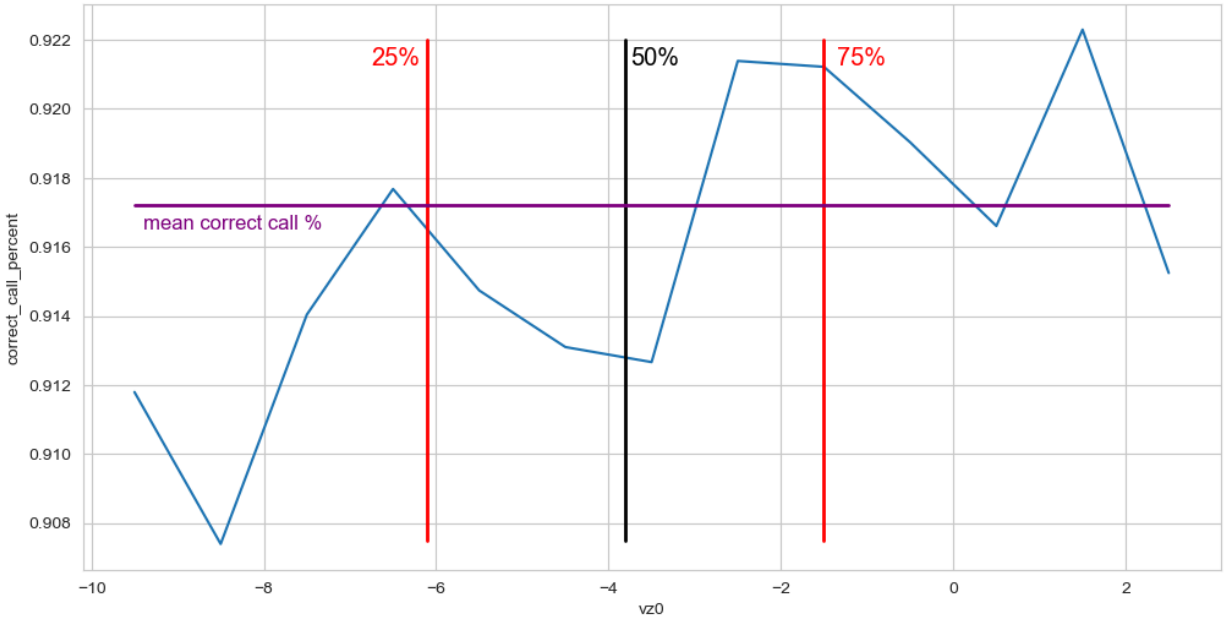
```
vz0_percents = []
vz0_index = []

for index in range(-21,16,1):
    temp_list = all_features_df[
        (index <= all_features_df['vz0']) &
        (all_features_df['vz0'] < index+1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        vz0_percents.append(temp_list[0])
        vz0_index.append(index+0.5)

vz0_stats = pd.DataFrame({'vz0':vz0_index, 'correct_call_percent':vz0_percents})

plt.figure(figsize=(12,6))
sns.lineplot(data=vz0_stats, x='vz0', y='correct_call_percent')
plt.plot([-6.1, -6.1], [0.78, 1], color='red', linewidth=2)
plt.plot([-3.8, -3.8], [0.78, 1], color='black', linewidth=2)
plt.plot([-1.5, -1.5], [0.78, 1], color='red', linewidth=2)
plt.text(-8.0, 0.985, '25%', color='red', fontsize='x-large')
plt.text(-3.55, 0.985, '50%', color='black', fontsize='x-large')
plt.text(-1.25, 0.985, '75%', color='red', fontsize='x-large')
plt.plot([-20.5, 15.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-19.8, 0.922, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a vertical velocity between -6.1 and -1.5, so we zoom in on that region:

In [167…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=vz0_stats[
    (-10 < vz0_stats['vz0']) &
    (vz0_stats['vz0'] < 3)
    ],
    x='vz0',
    y='correct_call_percent'
    )
plt.plot([-6.1, -6.1], [0.9075, 0.922], color='red', linewidth=2)
plt.plot([-3.8, -3.8], [0.9075, 0.922], color='black', linewidth=2)
plt.plot([-1.5, -1.5], [0.9075, 0.922], color='red', linewidth=2)
plt.text(-6.75, 0.92125, '25%', color='red', fontsize='x-large')
plt.text(-3.75, 0.92125, '50%', color='black', fontsize='x-large')
plt.text(-1.35, 0.92125, '75%', color='red', fontsize='x-large')
plt.plot([-9.5, 2.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-9.4, 0.9165, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



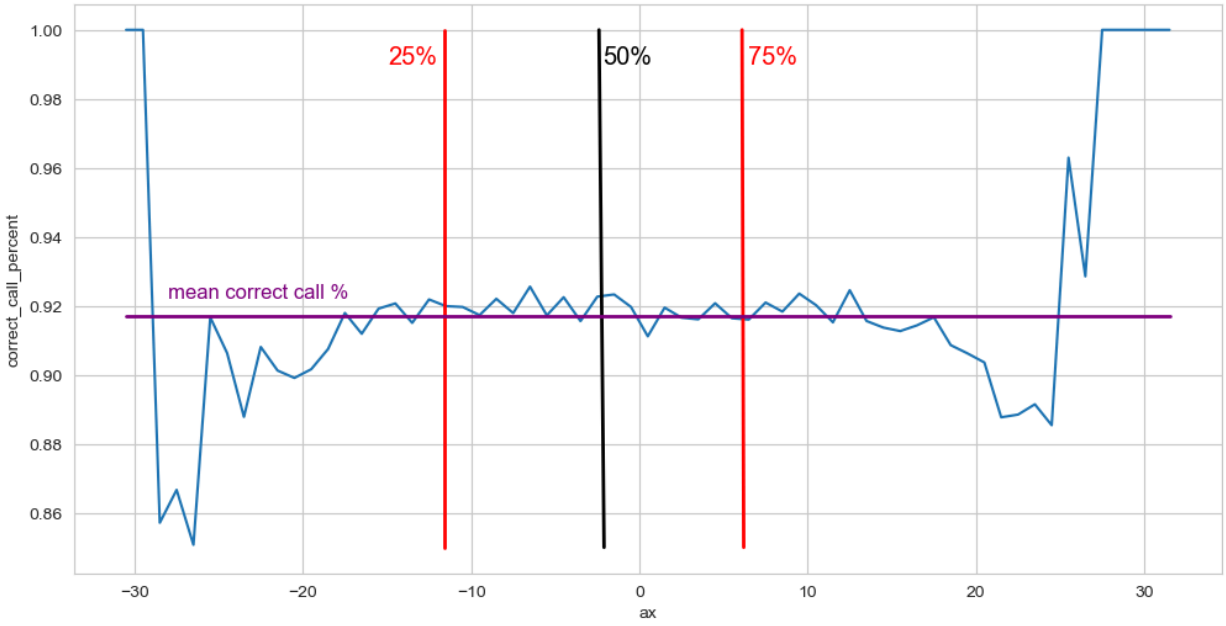
In [168…
```
# For summary statistics, run the following:
# all_features_df.vz0.describe()
```

3.1.8 - Acceleration

3.1.8.1 - ax

We split our data into buckets with a constant width of 1.

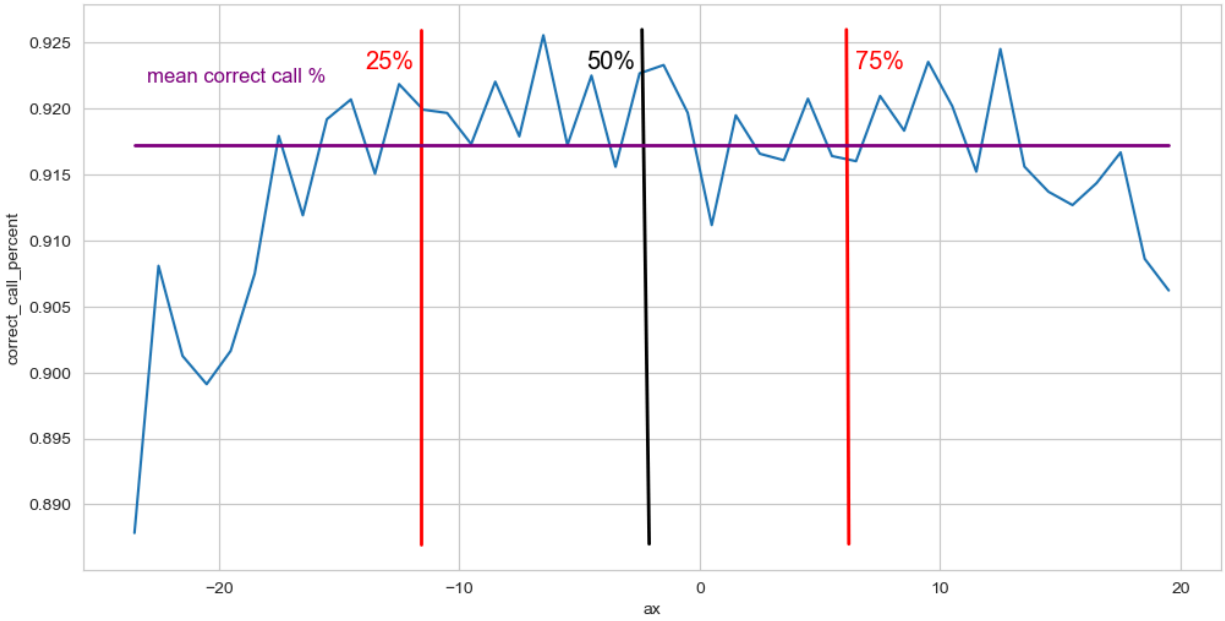
In [169…
```
ax_percents = []
ax_index = []

for index in range(-31,32,1):
    temp_list = all_features_df[
        (index <= all_features_df['ax']) &
        (all_features_df['ax'] < index+1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        ax_percents.append(temp_list[0])
        ax_index.append(index+0.5)

ax_stats = pd.DataFrame({'ax':ax_index, 'correct_call_percent':ax_percents})

plt.figure(figsize=(12,6))
sns.lineplot(data=ax_stats, x='ax', y='correct_call_percent')
plt.plot([-11.6, -11.6], [0.85, 1], color='red', linewidth=2)
plt.plot([-2.1, -2.4], [0.85, 1], color='black', linewidth=2)
plt.plot([6.2, 6.1], [0.85, 1], color='red', linewidth=2)
plt.text(-14.9, 0.99, '25%', color='red', fontsize='x-large')
plt.text(-2.2, 0.99, '50%', color='black', fontsize='x-large')
plt.text(6.45, 0.99, '75%', color='red', fontsize='x-large')
plt.plot([-30.5, 31.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-28, 0.922, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a release spin rate between -11.6 and 6.2, so we zoom in on that region:

In [170…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=ax_stats[
    (-24 < ax_stats['ax']) &
    (ax_stats['ax'] < 20)
    ],
    x='ax',
    y='correct_call_percent'
    )
plt.plot([-11.6, -11.6], [0.887, 0.926], color='red', linewidth=2)
plt.plot([-2.1, -2.4], [0.887, 0.926], color='black', linewidth=2)
plt.plot([6.2, 6.1], [0.887, 0.926], color='red', linewidth=2)
plt.text(-13.9, 0.923, '25%', color='red', fontsize='x-large')
plt.text(-4.7, 0.923, '50%', color='black', fontsize='x-large')
plt.text(6.45, 0.923, '75%', color='red', fontsize='x-large')
plt.plot([-23.5, 19.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-23, 0.922, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [171…

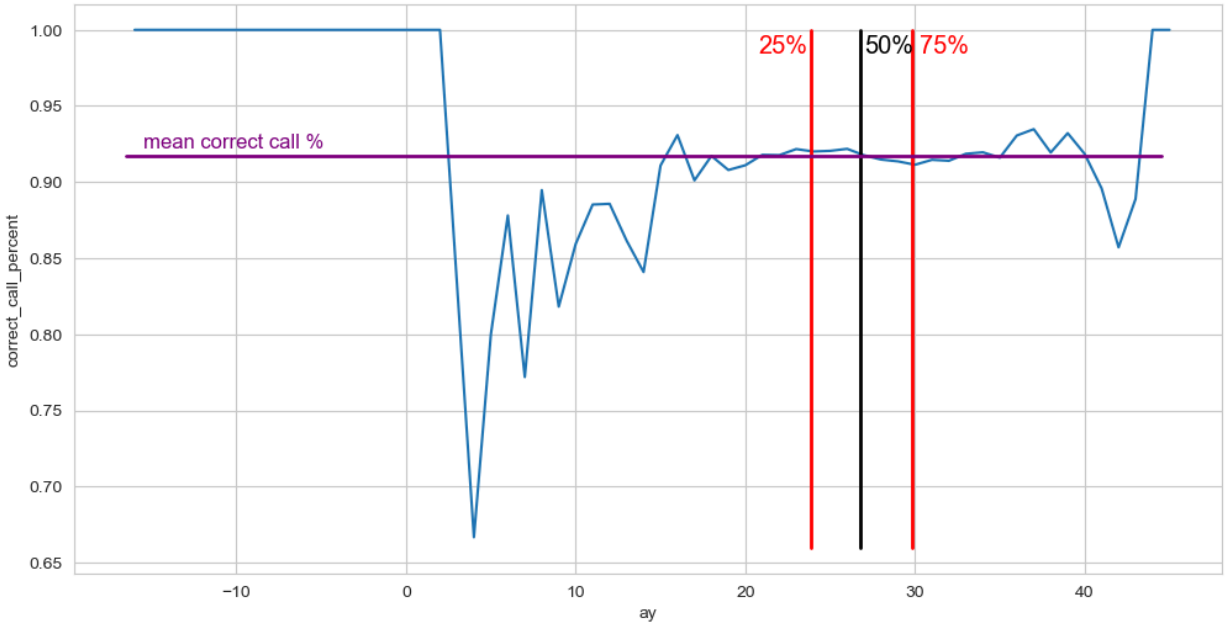
```
# For summary statistics, run the following:
# all_features_df.ax.describe()
```

3.1.8.2 - ay

We split our data into buckets with a constant width of 1.

In [172…

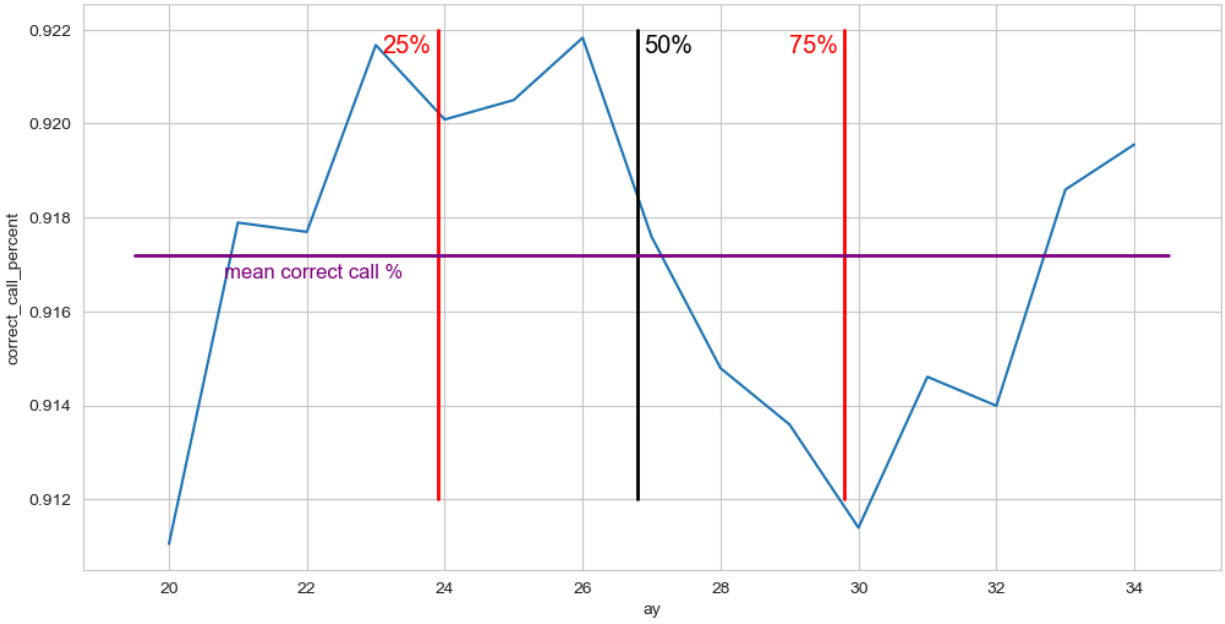
```
ay_percents = []
ay_index = []

for index in range(-17,45,1):
    temp_list = all_features_df[
        (index <= all_features_df['ay']) &
        (all_features_df['ay'] < index+1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        ay_percents.append(temp_list[0])
        ay_index.append(index+1)

ay_stats = pd.DataFrame({'ay':ay_index, 'correct_call_percent':ay_percents})

plt.figure(figsize=(12,6))
sns.lineplot(data=ay_stats, x='ay', y='correct_call_percent')
plt.plot([23.9, 23.9], [0.66, 1], color='red', linewidth=2)
plt.plot([26.8, 26.8], [0.66, 1], color='black', linewidth=2)
plt.plot([29.8, 29.8], [0.66, 1], color='red', linewidth=2)
plt.text(20.8, 0.985, '25%', color='red', fontsize='x-large')
plt.text(27, 0.985, '50%', color='black', fontsize='x-large')
plt.text(30.3, 0.985, '75%', color='red', fontsize='x-large')
plt.plot([-16.5, 44.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-15.5, 0.922, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur depth acceleration between 23.9 and 29.8, so we zoom in on that region:

In [173…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=ay_stats[
    (19 < ay_stats['ay']) &
    (ay_stats['ay'] < 35)
    ],
    x='ay',
    y='correct_call_percent'
    )
plt.plot([23.9, 23.9], [0.912, 0.922], color='red', linewidth=2)
plt.plot([26.8, 26.8], [0.912, 0.922], color='black', linewidth=2)
plt.plot([29.8, 29.8], [0.912, 0.922], color='red', linewidth=2)
plt.text(23.1, 0.9215, '25%', color='red', fontsize='x-large')
plt.text(26.89, 0.9215, '50%', color='black', fontsize='x-large')
plt.text(29.0, 0.9215, '75%', color='red', fontsize='x-large')
plt.plot([19.5, 34.5], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(20.8, 0.9167, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [174…

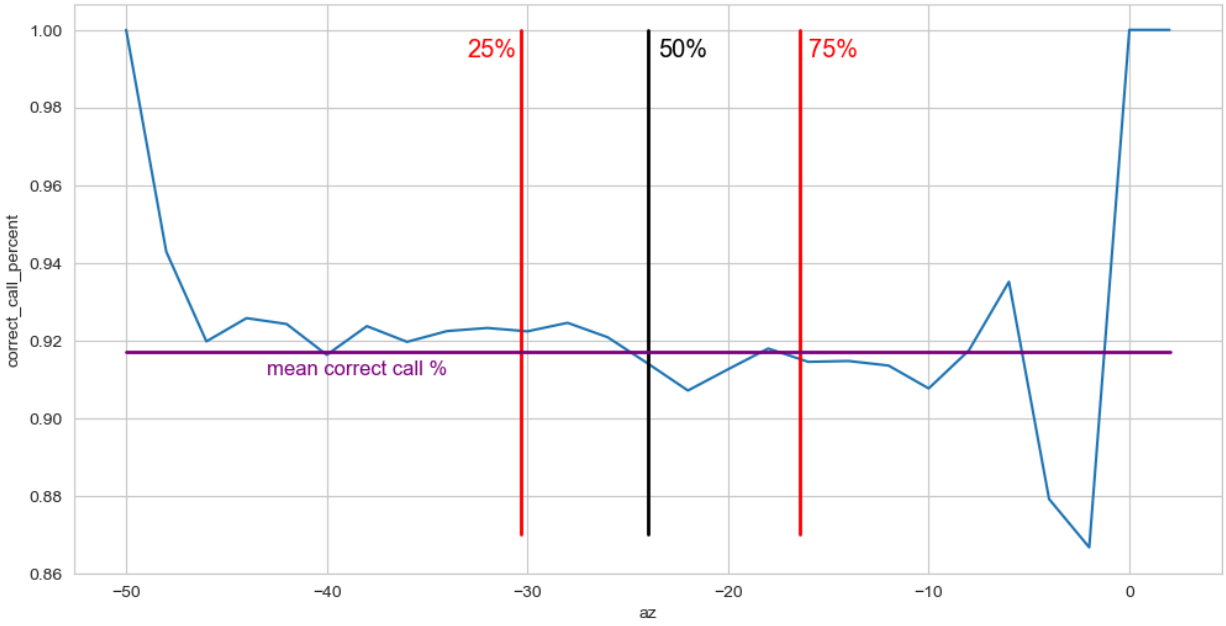
```
# For summary statistics, run the following:
# all_features_df.ay.describe()
```

3.1.8.3 - az

We split our data into buckets with a constant width of 1.

In [175…

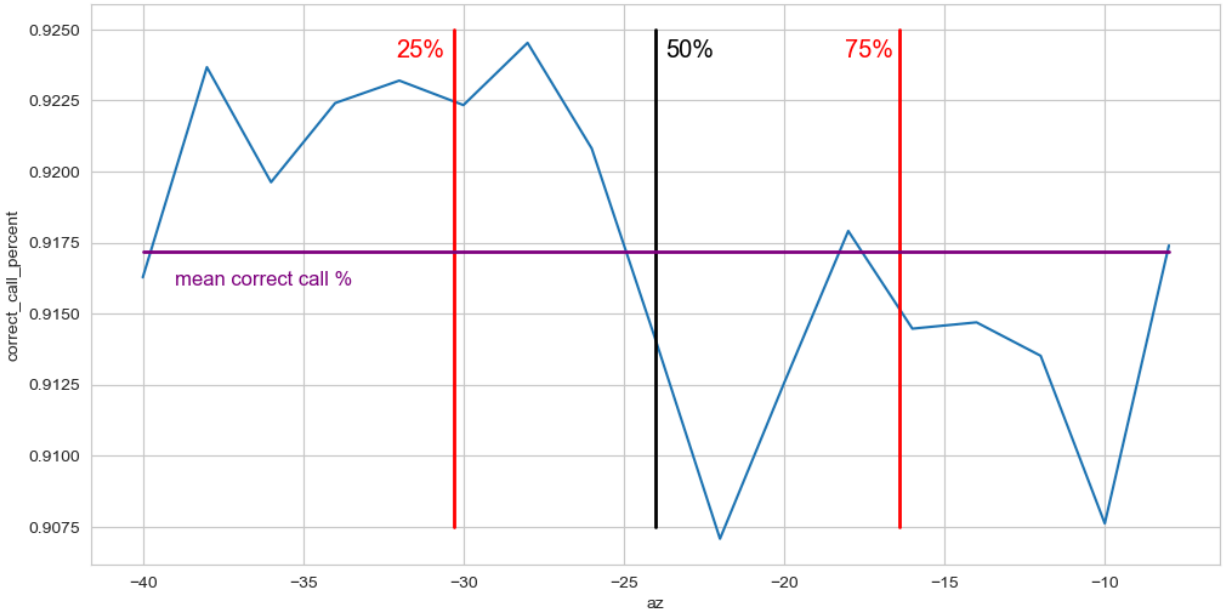
```
az_percents = []
az_index = []

for index in range(-51,3,2):
    temp_list = all_features_df[
        (index <= all_features_df['az']) &
        (all_features_df['az'] < index+1)
        ].correct_call.value_counts(normalize=True).to_list()
    if temp_list != []:
        az_percents.append(temp_list[0])
        az_index.append(index+1)

az_stats = pd.DataFrame({'az':az_index, 'correct_call_percent':az_percents})

plt.figure(figsize=(12,6))
sns.lineplot(data=az_stats, x='az', y='correct_call_percent')
plt.plot([-30.3, -30.3], [0.87, 1], color='red', linewidth=2)
plt.plot([-24, -24], [0.87, 1], color='black', linewidth=2)
plt.plot([-16.4, -16.4], [0.87, 1], color='red', linewidth=2)
plt.text(-33, 0.993, '25%', color='red', fontsize='x-large')
plt.text(-23.5, 0.993, '50%', color='black', fontsize='x-large')
plt.text(-16, 0.993, '75%', color='red', fontsize='x-large')
plt.plot([-50, 2], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-43, 0.911, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



Note that 50% of all pitches occur with a vertical acceleration between -30.3 and -16.4, so we zoom in on that region:

In [176…

```
plt.figure(figsize=(12,6))
sns.lineplot(data=az_stats[
    (-42 < az_stats['az']) &
    (az_stats['az'] < -6)
    ],
    x='az',
    y='correct_call_percent'
    )
plt.plot([-30.3, -30.3], [0.9075, 0.925], color='red', linewidth=2)
plt.plot([-24, -24], [0.9075, 0.925], color='black', linewidth=2)
plt.plot([-16.4, -16.4], [0.9075, 0.925], color='red', linewidth=2)
plt.text(-32.1, 0.924, '25%', color='red', fontsize='x-large')
plt.text(-23.7, 0.924, '50%', color='black', fontsize='x-large')
plt.text(-18.1, 0.924, '75%', color='red', fontsize='x-large')
plt.plot([-40, -8], [0.9172, 0.9172], color='purple', linewidth=2)
plt.text(-39, 0.916, 'mean correct call %', color='purple', fontsize='large')
plt.show()
```



In [177…

```
# For summary statistics, run the following:
# all_features_df.az.describe()
```

3.2 - Feature Refinement

After examining each feature in `Subsection 3.1`, it seems that the most important non-location features for correctly predicting balls and strikes will be a subset of

- `sz_top`,
- `effective_speed`,
- `release_spin_rate`,
- `count`,
- `pfx_z`,
- `vx0`,
- `vy0`,
- `vz0`, and
- `ay`.

While it is a location-based feature, we will also include `zone` at some points, for completeness.

Next, we will restrict our data frame to the aforementioned features and remove any pitches with missing values.

In [178…

```
cols_of_all_features_df_for_refined_df = [
    'binary_bs',
    'correct_call',
    'zone',
    'sz_top',
    'effective_speed',
    'release_spin_rate',
```

```
    'balls',
    'strikes',
    'pfx_z',
    'vx0',
    'vy0',
    'vz0',
    'ay'
]

refined_df = all_features_df[cols_of_all_features_df_for_refined_df]

refined_df = refined_df[
    (refined_df['binary_bs'].isna() == False) &
    (refined_df['zone'].isna() == False) &
    (refined_df['sz_top'].isna() == False) &
    (refined_df['effective_speed'].isna() == False) &
    (refined_df['release_spin_rate'].isna() == False) &
    (refined_df['balls'].isna() == False) &
    (refined_df['strikes'].isna() == False) &
    (refined_df['pfx_z'].isna() == False) &
    (refined_df['vx0'].isna() == False) &
    (refined_df['vy0'].isna() == False) &
    (refined_df['vz0'].isna() == False) &
    (refined_df['ay'].isna() == False)
]

refined_df
```

Out[178]:

	binary_bs	correct_call	zone	sz_top	effective_speed	release_spin_rate	balls	strikes	pfx_z
0	1	correct_call	7.0	3.33	90.0	2009.0	2	1	0.55
1	0	correct_call	14.0	3.33	88.9	1944.0	1	1	0.58
2	0	correct_call	13.0	3.33	89.9	1761.0	0	0	0.42
3	1	correct_call	8.0	3.60	89.2	1948.0	3	1	0.75
4	1	correct_call	5.0	3.51	88.5	1938.0	3	0	0.65
...	...	...	...	...	...	...	...	...	...
358476	1	correct_call	4.0	3.45	88.3	2206.0	0	0	1.27
358477	1	correct_call	7.0	3.82	87.5	2185.0	1	0	1.26
358478	0	correct_call	11.0	3.79	87.9	2082.0	0	0	1.23
358479	0	correct_call	3.0	3.39	86.7	2277.0	0	1	1.15
358480	1	correct_call	8.0	3.40	86.6	2214.0	0	0	1.25

356464 rows × 13 columns

We now create a `pitch_count` feature, instead of using `balls` and `strikes` separately. There are three pitches with a pitch count of either 4-1 or 4-2; we remove those here as well.

Additionally, we create a binary version of `correct_call`, which we denote by `binary_cc`.

In [179…

```
pitch_count_list = []
binary_cc_list = []

for index in refined_df.index:
    balls = refined_df.at[index, 'balls']
    strikes = refined_df.at[index, 'strikes']
    correct_call = refined_df.at[index, 'correct_call']
    pitch_count_list.append(str(balls)+'-'+str(strikes))
    if correct_call == 'correct_call':
        binary_cc_list.append(0)
    else:
        binary_cc_list.append(1)

refined_df = refined_df.assign(
    pitch_count=pitch_count_list,
    binary_cc=binary_cc_list
)

refined_df = refined_df[
    (refined_df['pitch_count'] != '4-1') &
    (refined_df['pitch_count'] != '4-2')
]
```

3.3 - Baseline Models

For our forward feature selection, we need a baseline model for comparison.

When predicting ball/strike calls, we will use a baseline model that randomly predicts balls and strikes at a rate corresponding to the number of balls and strikes present in the entire data frame.

When predicting correct calls, we will use a baseline model that randomly predicts correct calls at a rate corresponding to the number of correct calls present in the entire data frame.

In [180…

```
refined_df.binary_bs.value_counts(normalize=True)
```

Out[180]:

```
0    0.671111
1    0.328889
Name: binary_bs, dtype: float64
```

In [181…

```
feature_selection_bs_baseline_accs = []

np.random.seed(129)

for obs in range(1000):
    rand_draw = np.random.binomial(n=1, p=0.33, size=len(refined_df))
    feature_selection_bs_baseline_accs.append( accuracy_score(refined_df.binary_bs.val

feature_selection_bs_baseline_stats = pd.DataFrame(data={
    'feature_selection_bs_baseline_accuracy':feature_selection_bs_baseline_accs
})

feature_selection_bs_baseline_stats.describe()
```

Out[181]:

	feature_selection_bs_baseline_accuracy
count	1000.000000
mean	0.558181
std	0.000788
min	0.555432
25%	0.557663
50%	0.558210
75%	0.558725
max	0.561026

In [182…

```
refined_df.binary_cc.value_counts(normalize=True)
```

Out[182]:

```
0    0.91727
1    0.08273
Name: binary_cc, dtype: float64
```

In [183…

```
feature_selection_cc_baseline_accs = []

np.random.seed(129)

for obs in range(1000):
    rand_draw = np.random.binomial(n=1, p=0.08, size=len(refined_df))
    feature_selection_cc_baseline_accs.append( accuracy_score(refined_df.binary_bs.val

feature_selection_cc_baseline_stats = pd.DataFrame(data={
    'feature_selection_cc_baseline_accuracy':feature_selection_cc_baseline_accs
})

feature_selection_cc_baseline_stats.describe()
```

Out[183]:

	feature_selection_cc_baseline_accuracy
count	1000.000000
mean	0.643736
std	0.000462
min	0.642365
25%	0.643411
50%	0.643728
75%	0.644065
max	0.645094

3.4 - Feature Selection via Forward Selection

Feature selection by forward selection is a process for determining the most important features in a model, which we summarize below:

- Step 0: Create a baseline model for comparison
- Step 1: For each available feature, create a new model by adding only that feature to the baseline model and record how that models performs
- Step 2: Pick the feature with the model that performs best
- Step 3: If the best performing model does not improve on the baseline model, the features of your baseline model are your most important features. If the best performing model does improve on the baseline model, add your best feature to the baseline model and go back to step 1 with this new baseline model.

We now define functions that will which allow us to procede with forward feature selection.

- The `create_preprocessor` function takes in a list of model features and creates a preprocessor for a model using those features.

- The `forward_selection_round` function takes in
  a list of features that have already been selected (`current_features`) that will be present in each model, a list of features that will be tested one at a time
  (`remaining_features`), and
  ** a random state for repeatability.

This function iterates through the list of remaining features and returns a dictionary whose keys are entries in `remaining features` and whose values are the mean accuracy of running a 10-fold cross validation on a model with the current features and the new feature.

- The `forward_selection` function takes in
  a list of all possible features (`all_features`), and a baseline accuracy
  (`baseline_accuracy`).

This function begins by running `forward_selection_round` with `current_features=[]` , `remaining_features=all_features` , and `state=101` .

After the dictionary is returned, it iterates through to find the feature with the highest mean accuracy. If none of the features beat the baseline model, `forward_selection` returns an empty list. Otherwise, it runs `forward_selection_round` with `current_features` updated to include the feature with the highest mean accuracy from the previous run of `forward_selection_round` , `remaining_features` has the feature added to `current_features` removed, `state` is increased by 17, and the accuracy threshold is updated to the highest mean accuracy from the previous run of `forward_selection_round` .

This process repeats until none of the values in the returned dictionary are higher than the current accuracy threshold. At that point, `forward_selection` returns the list of selected features.

In [184…

```
def create_preprocessor(features:list):
    cat_feats = []
    cont_feats = []
    for feat in features:
        if feat in ['pitch_count', 'zone']:
            cat_feats.append(feat)
        else:
            cont_feats.append(feat)
    if (cont_feats != []) and (cat_feats != []):
        preprocessor = ColumnTransformer(
            transformers=[
                ('categorical', OneHotEncoder(handle_unknown='ignore'), cat_feats),
                ('continuous', StandardScaler(), cont_feats)
            ],
            remainder='passthrough'
        )
    elif (cont_feats != []):
        preprocessor = ColumnTransformer(
            transformers=[
                ('continuous', StandardScaler(), cont_feats)
            ],
            remainder='passthrough'
        )
    else:
        preprocessor = ColumnTransformer(
            transformers=[
                ('categorical', OneHotEncoder(handle_unknown='ignore'), cat_feats),
            ],
            remainder='passthrough'
        )
    return preprocessor


def forward_selection_round(current_features:list, remaining_features:list, state:int)
    mean_acc_dict = {}
    cur_feats = current_features.copy()
    for feat in remaining_features:
        test_features = cur_feats.copy()
        test_features.append(feat)
        accs = []
        preprocessor = create_preprocessor(test_features)
        kfold = StratifiedKFold(10, shuffle=True, random_state=state)
        for train_index, test_index in kfold.split(refined_df, refined_df.binary_bs):
            # Splitting data
            split_train = refined_df.iloc[train_index]
            split_test = refined_df.iloc[test_index]

            # Normalization pipeline
            pipeline = Pipeline([
                ('log_reg_preprocessor', preprocessor),
                ('log_reg', LogisticRegression(penalty=None, max_iter=5000))
            ])

            # Fitting
            pipeline.fit(split_train[test_features], split_train.binary_bs)

            # Predictions
            split_pred = pipeline.predict(split_test[test_features])
```

```
            # Evaluation
            accs.append( accuracy_score(split_test.binary_bs, split_pred) )
        acc_total = 0
        for acc in accs:
            acc_total += acc
        mean_acc = round(acc_total / len(accs), 4)
        mean_acc_dict[feat] = mean_acc
    return mean_acc_dict


def forward_selection(all_features:list, baseline_accuracy):
    current_features = []
    remaining_features = all_features.copy()
    current_acc = baseline_accuracy
    improving = True
    state = 101
    selection_round = 1
    while improving == True:
        print("Round " + str(selection_round) + " of feature selection is occurring")
        new_accs = forward_selection_round(current_features, remaining_features, state
        new_feat = 'None'
        for key in new_accs.keys():
            if new_accs[key] > current_acc:
                new_feat = key
                current_acc = new_accs[key]
        if new_feat == 'None':
            improving = False
        else:
            current_features.append(new_feat)
            remaining_features.remove(new_feat)
            state += 17
            selection_round += 1
    print(' ')
    return current_features
```

3.4.1 - Predicting Ball/Strike Calls, Including `zone` Feature

In [185…

```
forward_selection_list = [
    'pitch_count',
    'zone',
    'sz_top',
    'effective_speed',
    'release_spin_rate',
    'pfx_z',
    'vx0',
    'vy0',
    'vz0',
    'ay'
]

print('The possible features which can be picked in this forward selection are')
for feat in forward_selection_list:
    print('-) ' + feat)

print(' ')

forward_selected_features = forward_selection(forward_selection_list, 0.5582)
```

```
print("The features selected by forward selection are:")
for index in range(len(forward_selected_features)):
    print('#' + str(index+1) + '. ' + forward_selected_features[index])
```

```
The possible features which can be picked in this forward selection are
-) pitch_count
-) zone
-) sz_top
-) effective_speed
-) release_spin_rate
-) pfx_z
-) vx0
-) vy0
-) vz0
-) ay

Round 1 of feature selection is occurring
Round 2 of feature selection is occurring

The features selected by forward selection are:
#1. zone
```

3.4.2 - Predicting Ball/Strike Calls, Not Including `zone` Feature

In [186…

```
forward_selection_list = [
    'pitch_count',
    'sz_top',
    'effective_speed',
    'release_spin_rate',
    'pfx_z',
    'vx0',
    'vy0',
    'vz0',
    'ay'
]

print('The possible features which can be picked in this forward selection are')
for feat in forward_selection_list:
    print('-) ' + feat)

print(' ')

forward_selected_features = forward_selection(forward_selection_list, 0.5582)

print("The features selected by forward selection are:")
for index in range(len(forward_selected_features)):
    print('#' + str(index+1) + '. ' + forward_selected_features[index])
```

```
The possible features which can be picked in this forward selection are
-) pitch_count
-) sz_top
-) effective_speed
-) release_spin_rate
-) pfx_z
-) vx0
-) vy0
-) vz0
-) ay

Round 1 of feature selection is occurring
Round 2 of feature selection is occurring

The features selected by forward selection are:
#1. pitch_count
```

3.4.3 - Baseline Model for Predicting Correct Calls and Updating `forward_selection_round`

Our original `forward_selection_round` was defined to predict ball/strike calls. We now update it to predict correct calls.

In [187…

```python
def forward_selection_round(current_features:list, remaining_features:list, state:int):
    mean_acc_dict = {}
    cur_feats = current_features.copy()
    for feat in remaining_features:
        test_features = cur_feats.copy()
        test_features.append(feat)
        accs = []
        preprocessor = create_preprocessor(test_features)
        kfold = StratifiedKFold(10, shuffle=True, random_state=state)
        for train_index, test_index in kfold.split(refined_df, refined_df.binary_cc):
            # Splitting data
            split_train = refined_df.iloc[train_index]
            split_test = refined_df.iloc[test_index]

            # Normalization pipeline
            pipeline = Pipeline([
                ('log_reg_preprocessor', preprocessor),
                ('log_reg', LogisticRegression(penalty=None, max_iter=5000))
            ])

            # Fitting
            pipeline.fit(split_train[test_features], split_train.binary_cc)

            # Predictions
            split_pred = pipeline.predict(split_test[test_features])

            # Evaluation
            accs.append( accuracy_score(split_test.binary_cc, split_pred) )
        acc_total = 0
        for acc in accs:
            acc_total += acc
        mean_acc = round(acc_total / len(accs), 4)
        mean_acc_dict[feat] = mean_acc
    return mean_acc_dict
```

3.4.4 - Predicting Correct Calls, Including `zone` Feature

In [188…

```
forward_selection_list = [
    'pitch_count',
    'zone',
    'sz_top',
    'effective_speed',
    'release_spin_rate',
    'pfx_z',
    'vx0',
    'vy0',
    'vz0',
    'ay'
]

print('The possible features which can be picked in this forward selection are')
for feat in forward_selection_list:
    print('-) ' + feat)

print(' ')

forward_selected_features = forward_selection(forward_selection_list, 0.6437)

print("The features selected by forward selection are:")
for index in range(len(forward_selected_features)):
    print('#' + str(index+1) + '. ' + forward_selected_features[index])
```

```
The possible features which can be picked in this forward selection are
-) pitch_count
-) zone
-) sz_top
-) effective_speed
-) release_spin_rate
-) pfx_z
-) vx0
-) vy0
-) vz0
-) ay

Round 1 of feature selection is occurring
Round 2 of feature selection is occurring

The features selected by forward selection are:
#1. pitch_count
```

3.4.5 - Predicting Correct Calls, Not Including `zone` Feature

In [189…

```
forward_selection_list = [
    'pitch_count',
    'sz_top',
    'effective_speed',
    'release_spin_rate',
    'pfx_z',
    'vx0',
```

```
    'vy0',
    'vz0',
    'ay'
]

print('The possible features which can be picked in this forward selection are')
for feat in forward_selection_list:
    print('-) ' + feat)

print(' ')

forward_selected_features = forward_selection(forward_selection_list, 0.6437)

print("The features selected by forward selection are:")
for index in range(len(forward_selected_features)):
    print('#' + str(index+1) + '. ' + forward_selected_features[index])
```

```
The possible features which can be picked in this forward selection are
-) pitch_count
-) sz_top
-) effective_speed
-) release_spin_rate
-) pfx_z
-) vx0
-) vy0
-) vz0
-) ay

Round 1 of feature selection is occurring
Round 2 of feature selection is occurring

The features selected by forward selection are:
#1. pitch_count
```

3.4.6 - Summary

According to forward feature selection, we see that `pitch_count` is the most important non-positional feature for predicting both umpire ball/strike calls and for predicting whether an umpire makes a correct call.

3.5 - Feature Selection via L^1 Regularization

When using logistic regression with an L^1 penalty, we can decrease the hyperparameter `C` to increase the regularization strength. Doing so will force the the coefficients in our model towards 0, with the coefficients on the least important features going towards 0 faster. As such, we can look at how small we must make `C` for each of our coefficients to become 0 as a way of determining which features are most important.

We will use this approach both for trying to predict ball/strike calls and for trying to predict correct umpire calls.

We do not include the `zone` feature at all in this subsection.

3.5.1 - Predicting Ball/Strike Calls

We begin my creating models for all of our different `C` values and recording the coefficients for each model.

In [190…

```
bs_C_values = [
    0.00007,
    0.00008,
    0.00009,
    0.0001,
    0.0002,
    0.0003,
    0.0004,
    0.0005,
    0.0006,
    0.0007,
    0.0008,
    0.0009,
    0.001,
    0.01,
    0.1,
    1,
    10,
    100,
    1000
]

regularization_features = [
    'pitch_count',
    'sz_top',
    'effective_speed',
    'release_spin_rate',
    'pfx_z',
    'vx0',
    'vy0',
    'vz0',
    'ay'
]

bs_regularization_coefficients = np.zeros((10*len(bs_C_values), 20))
bs_regularization_features_from_model = []

regularization_preprocessor = ColumnTransformer(
    transformers=[
        ('categorical', OneHotEncoder(handle_unknown='ignore'), ['pitch_count']),
        ('continuous', StandardScaler(), [
            'sz_top',
            'effective_speed',
            'release_spin_rate',
            'pfx_z',
            'vx0',
            'vy0',
            'vz0',
            'ay'
        ])
    ],
    remainder='passthrough'
```

```
)

bs_regularization_kfold = StratifiedKFold(10, shuffle=True, random_state=943)

for index in range(len(bs_C_values)):
    c_val = bs_C_values[index]
    counter = 10*(index)
    print('Current C Value: ' + str(bs_C_values[index]))
    for train_index, test_index in bs_regularization_kfold.split(refined_df, refined_d
        # Splitting data
        split_train = refined_df.iloc[train_index]
        split_test = refined_df.iloc[test_index]

        # Normalization pipeline
        bs_regularization_pipeline = Pipeline([
            ('log_reg_preprocessor', regularization_preprocessor),
            ('log_reg', LogisticRegression(C=c_val, penalty="l1", max_iter=5000, solve
        ])

        # Fitting
        bs_regularization_pipeline.fit(split_train[regularization_features], split_tra

        # Coefficients
        bs_regularization_coefficients[counter,:] = bs_regularization_pipeline['log_re
        counter += 1
        bs_regularization_features_from_model.append(bs_regularization_pipeline[:-1].g

print(' ')
print('Done')
```

```
Current C Value: 7e-05
Current C Value: 8e-05
Current C Value: 9e-05
Current C Value: 0.0001
Current C Value: 0.0002
Current C Value: 0.0003
Current C Value: 0.0004
Current C Value: 0.0005
Current C Value: 0.0006
Current C Value: 0.0007
Current C Value: 0.0008
Current C Value: 0.0009
Current C Value: 0.001
Current C Value: 0.01
Current C Value: 0.1
Current C Value: 1
Current C Value: 10
Current C Value: 100
Current C Value: 1000

Done
```

Next, we create a data frame with the coefficients for each model created. Note that we run a 10-fold cross validation for each value of `C`.

In [191…

```
bs_regularization_df_indices = []

for index in range(len(bs_C_values)):
    for counter in range(10):
```

```
        bs_regularization_df_indices.append(bs_C_values[index])

bs_regularization_df = pd.DataFrame(
    bs_regularization_coefficients,
    columns=bs_regularization_features_from_model[0],
    index=bs_regularization_df_indices
)

bs_regularization_df = bs_regularization_df.rename(columns={
    'categorical__pitch_count_0-0':'0-0',
    'categorical__pitch_count_0-1':'0-1',
    'categorical__pitch_count_0-2':'0-2',
    'categorical__pitch_count_1-0':'1-0',
    'categorical__pitch_count_1-1':'1-1',
    'categorical__pitch_count_1-2':'1-2',
    'categorical__pitch_count_2-0':'2-0',
    'categorical__pitch_count_2-1':'2-1',
    'categorical__pitch_count_2-2':'2-2',
    'categorical__pitch_count_3-0':'3-0',
    'categorical__pitch_count_3-1':'3-1',
    'categorical__pitch_count_3-2':'3-2',
    'continuous__sz_top':'sz_top',
    'continuous__effective_speed':'effective_speed',
    'continuous__release_spin_rate':'release_spin_rate',
    'continuous__pfx_z':'pfx_z',
    'continuous__vx0':'vx0',
    'continuous__vy0':'vy0',
    'continuous__vz0':'vz0',
    'continuous__ay':'ay'
    }
)

bs_regularization_df
```

Out[191]:

	0-0	0-1	0-2	1-0	1-1	1-2	2-0	2-1	2
0.00007	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
0.00007	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
0.00007	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
0.00007	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
0.00007	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0000
...	...	...	...	...	...	...	...	...	...
1000.00000	0.681670	-0.369557	-1.568129	0.543771	-0.221942	-1.321867	0.734545	0.077777	-0.9281
1000.00000	0.685909	-0.368541	-1.565323	0.541175	-0.226683	-1.336041	0.728654	0.074409	-0.9180
1000.00000	0.681927	-0.370330	-1.566417	0.542068	-0.218732	-1.333756	0.732135	0.064584	-0.9211
1000.00000	0.686227	-0.366407	-1.571635	0.547084	-0.216582	-1.348181	0.746637	0.077982	-0.9370
1000.00000	0.685659	-0.359997	-1.582916	0.549235	-0.207971	-1.333100	0.722905	0.072491	-0.9222

190 rows × 20 columns

Finally, we create a new data frame where each value of `C` occurs only once; we do this by taking the average of each coefficient over the 10 models coming from cross validation.

In [192…

```
bs_regularization_coefficients_means = bs_regularization_coefficients = np.zeros((len(

for row in range(len(bs_C_values)):
    for col in range(len(bs_regularization_df.columns)):
        bs_regularization_coefficients_means[row:col] = round(bs_regularization_df.loc

bs_regularization_means_df = pd.DataFrame(
    bs_regularization_coefficients_means,
    columns=bs_regularization_df.columns.to_list(),
    index=bs_C_values
)

bs_regularization_means_df
```

Out[192]:

	0-0	0-1	0-2	1-0	1-1	1-2	2-0	2-1	2-2	3-0	3-1
0.00007	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.00008	0.0912	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.00009	0.1771	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.00010	0.2455	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.00020	0.5500	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.00030	0.6577	0.0000	-0.1139	0.2240	0.0000	-0.2008	0.0000	0.0000	0.0000	0.0000	0.0000
0.00040	0.6887	-0.0000	-0.3578	0.3320	0.0000	-0.4022	0.0000	0.0000	0.0000	0.0000	0.0000
0.00050	0.6940	-0.0243	-0.5356	0.3820	0.0000	-0.5489	0.0905	0.0000	-0.1163	0.0225	0.0000
0.00060	0.7059	-0.0663	-0.6560	0.4232	0.0000	-0.6451	0.2208	0.0000	-0.2182	0.2843	0.0000
0.00070	0.7145	-0.0967	-0.7481	0.4526	0.0000	-0.7175	0.3105	0.0000	-0.2943	0.4595	0.0000
0.00080	0.7207	-0.1196	-0.8212	0.4745	0.0000	-0.7742	0.3762	0.0000	-0.3534	0.5862	0.0000
0.00090	0.7256	-0.1376	-0.8808	0.4915	0.0000	-0.8198	0.4265	0.0000	-0.4007	0.6830	0.0000
0.00100	0.7294	-0.1521	-0.9303	0.5050	0.0000	-0.8574	0.4663	0.0000	-0.4396	0.7596	0.0000
0.01000	0.6909	-0.3446	-1.4916	0.5435	-0.1927	-1.2807	0.7083	0.0509	-0.8682	1.3154	0.3843
0.10000	0.6857	-0.3658	-1.5617	0.5459	-0.2182	-1.3306	0.7308	0.0734	-0.9181	1.3739	0.4369
1.00000	0.6855	-0.3677	-1.5687	0.5464	-0.2205	-1.3355	0.7333	0.0758	-0.9229	1.3800	0.4423
10.00000	0.6854	-0.3679	-1.5694	0.5464	-0.2207	-1.3360	0.7335	0.0761	-0.9234	1.3806	0.4429
100.00000	0.6854	-0.3679	-1.5694	0.5464	-0.2208	-1.3360	0.7335	0.0761	-0.9235	1.3807	0.4429
1000.00000	0.6854	-0.3679	-1.5694	0.5464	-0.2207	-1.3360	0.7335	0.0761	-0.9235	1.3807	0.4429

Note the smallest value of C where each coefficient is non-zero:

0.00008: `0-0`
0.00009:

0.0001:
0.0002: `release_spin_rate`
0.0003: `0-2` , `1-0` , `1-2` , `vy0`
0.0004:
0.0005: `0-1` , `2-0` , `2-2` , `3-0` , `effective_speed` ,
0.0006:
0.0007: `vx0` , `vz0`
0.0008:
0.0009:
0.001:
0.01: `1-1` , `2-1` , `3-1` , `3-2` , `sz_top` , `pfx_z` , `ay`

3.5.2 - Predicting Correct Calls

We begin my creating models for all of our different `C` values and recording the coefficients for each model.

In [193…

```
cc_C_values = [
    0.0001,
    0.0002,
    0.0003,
    0.0004,
    0.0005,
    0.0006,
    0.0007,
    0.0008,
    0.0009,
    0.001,
    0.002,
    0.003,
    0.004,
    0.005,
    0.006,
    0.007,
    0.008,
    0.009,
    0.01,
    0.1,
    1,
    10,
    100,
    1000
]

regularization_features = [
    'pitch_count',
    'sz_top',
    'effective_speed',
    'release_spin_rate',
    'pfx_z',
    'vx0',
    'vy0',
    'vz0',
```

```
    'ay'
]

cc_regularization_coefficients = np.zeros((10*len(cc_C_values), 20))
cc_regularization_features_from_model = []

regularization_preprocessor = ColumnTransformer(
    transformers=[
        ('categorical', OneHotEncoder(handle_unknown='ignore'), ['pitch_count']),
        ('continuous', StandardScaler(), [
            'sz_top',
            'effective_speed',
            'release_spin_rate',
            'pfx_z',
            'vx0',
            'vy0',
            'vz0',
            'ay'
        ])
    ],
    remainder='passthrough'
)

cc_regularization_kfold = StratifiedKFold(10, shuffle=True, random_state=943)

for index in range(len(cc_C_values)):
    c_val = cc_C_values[index]
    counter = 10*(index)
    print('Current C Value: ' + str(cc_C_values[index]))
    for train_index, test_index in cc_regularization_kfold.split(refined_df, refined_d
        # Splitting data
        split_train = refined_df.iloc[train_index]
        split_test = refined_df.iloc[test_index]

        # Normalization pipeline
        cc_regularization_pipeline = Pipeline([
            ('log_reg_preprocessor', regularization_preprocessor),
            ('log_reg', LogisticRegression(C=c_val, penalty="l1", max_iter=5000, solve
        ])

        # Fitting
        cc_regularization_pipeline.fit(split_train[regularization_features], split_tra

        # Coefficients
        cc_regularization_coefficients[counter,:] = cc_regularization_pipeline['log_re
        counter += 1
        cc_regularization_features_from_model.append(cc_regularization_pipeline[:-1].g

print(' ')
print('Done')
```

```
Current C Value: 0.0001
Current C Value: 0.0002
Current C Value: 0.0003
Current C Value: 0.0004
Current C Value: 0.0005
Current C Value: 0.0006
Current C Value: 0.0007
Current C Value: 0.0008
Current C Value: 0.0009
Current C Value: 0.001
Current C Value: 0.002
Current C Value: 0.003
Current C Value: 0.004
Current C Value: 0.005
Current C Value: 0.006
Current C Value: 0.007
Current C Value: 0.008
Current C Value: 0.009
Current C Value: 0.01
Current C Value: 0.1
Current C Value: 1
Current C Value: 10
Current C Value: 100
Current C Value: 1000

Done
```

Next, we create a data frame with the coefficients for each model created. Note that we run a 10-fold cross validation for each value of `C`.

In [194…

```
cc_regularization_df_indices = []

for index in range(len(cc_C_values)):
    for counter in range(10):
        cc_regularization_df_indices.append(cc_C_values[index])

cc_regularization_df = pd.DataFrame(
    cc_regularization_coefficients,
    columns=cc_regularization_features_from_model[0],
    index=cc_regularization_df_indices
)

cc_regularization_df = cc_regularization_df.rename(columns={
    'categorical__pitch_count_0-0':'0-0',
    'categorical__pitch_count_0-1':'0-1',
    'categorical__pitch_count_0-2':'0-2',
    'categorical__pitch_count_1-0':'1-0',
    'categorical__pitch_count_1-1':'1-1',
    'categorical__pitch_count_1-2':'1-2',
    'categorical__pitch_count_2-0':'2-0',
    'categorical__pitch_count_2-1':'2-1',
    'categorical__pitch_count_2-2':'2-2',
    'categorical__pitch_count_3-0':'3-0',
    'categorical__pitch_count_3-1':'3-1',
    'categorical__pitch_count_3-2':'3-2',
    'continuous__sz_top':'sz_top',
    'continuous__effective_speed':'effective_speed',
    'continuous__release_spin_rate':'release_spin_rate',
    'continuous__pfx_z':'pfx_z',
```

```
    'continuous__vx0':'vx0',
    'continuous__vy0':'vy0',
    'continuous__vz0':'vz0',
    'continuous__ay':'ay'
    }
)

cc_regularization_df
```

Out[194]:

	0-0	0-1	0-2	1-0	1-1	1-2	2-0	2-1	2-
0.0001	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
0.0001	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
0.0001	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
0.0001	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
0.0001	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
...	...	...	...	...	...	...	...	...	
1000.0000	0.074185	-0.300427	-1.056055	0.122388	-0.173387	-0.823925	0.191499	0.021168	-0.48718
1000.0000	0.070064	-0.298181	-1.065851	0.136060	-0.174191	-0.835526	0.177087	0.004178	-0.49683
1000.0000	0.069070	-0.310050	-1.040753	0.134277	-0.194380	-0.834798	0.188320	0.006197	-0.48910
1000.0000	0.071682	-0.300683	-1.059450	0.147749	-0.174765	-0.822584	0.179929	0.007244	-0.47762
1000.0000	0.069061	-0.304207	-1.057728	0.139076	-0.171512	-0.836668	0.174439	0.001579	-0.48331

240 rows × 20 columns

Finally, we create a new data frame where each value of `C` occurs only once; we do this by taking the average of each coefficient over the 10 models coming from cross validation.

In [195...

```
cc_regularization_coefficients_means = cc_regularization_coefficients = np.zeros((len(

for row in range(len(cc_C_values)):
    for col in range(len(cc_regularization_df.columns)):
        cc_regularization_coefficients_means[row:col] = round(cc_regularization_df.loc

cc_regularization_means_df = pd.DataFrame(
    cc_regularization_coefficients_means,
    columns=cc_regularization_df.columns.to_list(),
    index=cc_C_values
)

cc_regularization_means_df
```

Out[195]:

	0-0	0-1	0-2	1-0	1-1	1-2	2-0	2-1	2-2	3-0	3-1	3-2	sz_
0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0002	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0003	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0004	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0005	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0006	0.0010	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0007	0.0421	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0008	0.0737	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0009	0.0981	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0010	0.1175	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0020	0.1751	0.0000	-0.2998	0.1219	0.0000	-0.2548	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
0.0030	0.1722	-0.0460	-0.4861	0.1594	0.0000	-0.3998	0.0000	0.0	-0.0484	0.0	0.0000	0.0	0.(
0.0040	0.1598	-0.0960	-0.6005	0.1663	0.0000	-0.4889	0.0537	0.0	-0.1376	0.0	0.0000	0.0	0.(
0.0050	0.1544	-0.1243	-0.6708	0.1722	0.0000	-0.5422	0.0951	0.0	-0.1910	0.0	0.0000	0.0	0.(
0.0060	0.1483	-0.1457	-0.7220	0.1735	-0.0056	-0.5811	0.1190	0.0	-0.2299	0.0	0.0000	0.0	0.(
0.0070	0.1347	-0.1703	-0.7687	0.1653	-0.0301	-0.6185	0.1265	0.0	-0.2673	0.0	0.0000	0.0	0.(
0.0080	0.1233	-0.1900	-0.8056	0.1578	-0.0516	-0.6481	0.1306	0.0	-0.2969	0.0	0.0000	0.0	0.(
0.0090	0.1151	-0.2046	-0.8340	0.1527	-0.0676	-0.6706	0.1343	0.0	-0.3194	0.0	0.0034	0.0	0.(
0.0100	0.1093	-0.2156	-0.8562	0.1493	-0.0797	-0.6880	0.1379	0.0	-0.3367	0.0	0.0099	0.0	0.(
0.1000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
10.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
100.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(
1000.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0	0.0000	0.0	0.0000	0.0	0.(

Note the smallest value of C where each coefficient is non-zero:

0.0001:
0.0002:
0.0003:
0.0004:
0.0005: `sz_top`
0.0006: `0-0`
0.0007:
0.0008:

0.0009:
0.001:
0.002: `0-2`, `1-0`, `1-2`, `effective_speed`, `pfx_z`
0.003: `0-1`, `2-2`, `release_spin_rate`
0.004: `2-0`, `ay`
0.005: `vy0`
0.006: `1-1`, `vz0`
0.007:
0.008: `vx0`
0.009: `3-1`
0.01:
0.1:

Always zero `2-1`, `3-0`, `3-2`

3.5.3 - Summary

We note that `count` was an important feature when using L^1 regularization for both predicting ball/strike calls and for predicting correct umpire calls.

For predicting balls/strikes, `release_spin_rate` and `vy0` were the next most important features according to L^1 regularization.

For predicting correct umpire calls, `sz_top` was our most important feature according to L^1 regularization. The next set of features worth considering would be `effective_speed` and `pfx_z`.

It is understandable `sz_top` is very important for predicting correct umpire calls; the top and bottom of the zone change from batter to batter, whereas the left and right sides of the zone do not. Furthermore, there is greater variation in `sz_top` than `sz_bot` (with respective standard deviations of 0.205 and 0.122 in `all_features_df`), despite being correlated.

Furthermore, `effective_speed` (release speed adjusted by release extension) and `pfx_z` (vertical movement) are conventionally expected predictors of correct calls.

Appendix A - Data Collection, Cleaning, and Processing

In this appendix, we describe how the data for this project was obtained, cleaned, and processed across three notebooks.

A.1 - Accessing Raw Pitch Data

In the notebook `2023 MLB Pitch Data - Month by Month Queries using Pybaseball`, we install the pybaseball package and use it to download Statcast data from Baseball Savant. Given the amount of data we requested, we split the request into six queries. More specifically, we request Statcast data from

- 20 Mar 2023 through 30 Apr 2023, which we export as `month1.csv`;
- 01 May 2023 through 31 May 2023, which we export as `month2.csv`;
- 01 Jun 2023 through 30 Jun 2023, which we export as `month3.csv`;
- 01 Jul 2023 through 31 Jul 2023, which we export as `month4.csv`;
- 01 Aug 2023 through 31 Aug 2023, which we export as `month5.csv`;
- 01 Sep 2023 through 01 Oct 2023, which we export as `month6.csv`.

A.2 - Merging, Cleaning, and Processing Pitch Data I

In the notebook `2023 MLB Pitch Data - Merging, Cleaning, and Processing`, we begin by reading in `month1.csv`, `month2.csv`, `month3.csv`, `month4.csv`, `month5.csv`, and `month6.csv` as data frames, which we then merge together. Next, we restrict to pitches which were either balls or called strikes; pitches where an umpire has to actively make a ball/strike call. We then restrict to Statcast features which will either be considered in our project or remain necessary for further preprocessing.

We will soon join this data frame with umpire data, so we now describe obtaining the umpire data.

A.3 - Accessing Umpire Data I

In the notebook `2023 MLB Umpire Data`, we use `BeautifulSoup` from the `bs4` package to obtain home plate umpire data from Baseball Reference. We begin by defining three functions:

- `get_umpire_info_from_url()` which takes in the URL of a box score from baseball reference and returns the home plate umpire's name as a string;
- `get_umpire_info_from_request()` which takes in a request from a URL of a box score from baseball reference and returns the home plate umpire's name as a string; and
- `get_team_info()` which takes in a three letter team abbreviation (e.g. 'BOS'), iterates through possible URLs for box scores with that team as the home team, records the necessary information for each box score, and exports the umpire information as a CSV (e.g. `BOS_umpire.csv`).

After defining these functions, there is code in place to scrape the information for each team (although this code is currently commented out, as the data has already been scraped).

Next, we read in the umpire information for each team as a separate data frame, merge these data frames together, and export this single data frame as `umpires.csv`.

A.4 - Merging, Cleaning, and Processing Pitch Data II

Switching back to `2023 MLB Pitch Data - Merging, Cleaning, and Processing`, we read in `umpires.csv` left join it with our pitch data. However, there are still pitches with missing umpire information, as games which are part of a doubleheader (when two teams play two games on the same day) are given a different URL formatting on Baseball Reference.

As such, we create a data frame of pitches with missing umpire information and export the necessary information as `missing_umpires.csv`.

A.5 - Accessing Umpire Data II

Going back to `2023 MLB Umpire Data`, we read in the `missing_umpires.csv` and see that all but one game with missing umpire is a doubleheader. Now that we have the date and home team for each doubleheader, we scrape the home plate umpire for each of these games (as well as the one exception), create a data frame with the umpire data, and export the data frame as `missing_umpires_return.csv`.

A.6 - Merging, Cleaning, and Processing Pitch Data III

We again go back to `2023 MLB Pitch Data - Merging, Cleaning, and Processing`, where we read in `missing_umpires_return.csv` and join it to our data frame, which completes the joining of umpire information.

Next, we proceed to feature engineering, where we create

- `binary_bs`, a binary version of ball/strike calls;
- `hscw`, which determines if the pitch was in the heart, shadow, chase, or waste portion of the zone;
- `true_ball/strike`, which determines if the pitch was actually a ball or strike;
- `correct_call`, which determines if the umpire's call (`ball/strike`) matches with `true_ball/strike`;
- `plate_x_mag` and `plate_x_dir`, which respectively record the magnitude and sign of `plate_x`.

We then create a data frame with all the features outlined in `Sections 2,3`, which we then export as `large_model_data.csv`.

Finally, we create a data frame with only the features outlined in `Section 2`, which we then export as `small_model_data.csv`.