A Statistical Analysis of the MLB Strikeout Rate

Samuel Northam

Master of Science, Data Analytics, Western Governors University

Dr. Daniel Smith, PhD

October 2022

Table of Contents

Part I: Research Question

"Statistics are the lifeblood of baseball. In no other sport are so many available and studied so assiduously by participants and fans" (Koppett 1967). With the abundance of baseball data stored in various online databases, such as baseball-reference or FanGraphs, performing statistical analysis on baseball is easier than ever. Teams use analytics to position their fielders in specific ways, pitchers use analytics to determine how they will pitch to a batter, and the league uses analytics to help see how effective rule changes have been.

The American League and the National League have existed for over 130 years, but they did not merge to form the Major League Baseball (MLB) until 1903 when they played the first World Series (Attanasio, 2022). Since the first world series, the league has expanded significantly. It has expanded to include 3 divisions within each league and went from 16 teams to 30 teams. The league has also since added 24 additional games a season. With all these changes the league has undertaken, one thing has stayed consistent: statistics.

The basic baseball statistics have not changed since the game's inception. Batting Average, Runs Batted In (RBIs), and Home Runs have always been tracked for hitters while Wins, Earned Run Average (ERA), and Strikeouts have been tracked for pitchers. As technology advanced, so did the categories that statisticians tracked. With the creation of the Society for American Baseball Research (SABR) in the early 1980s, a new class of statistics was created: Advanced Analytics (Birnbaum, 2022). Advanced Analytics include On Base Percentage, Slugging Percentage, and WAR. These newer analytics attempt to answer more complex questions that the simple analytics could not answer. Despite the creation of more advanced statistics, teams and scouts often still use the basic statistics to help develop a game plan.

This study will focus on simple statistics to see how the game has evolved over time and to see if the future of the sport can be predicted. The main statistic that will be analyzed will be the MLB strikeout rate, which is the total number of strikeouts per 100 at bats: $\frac{\mathrm{K}}{\mathrm{AB}} * 100$. The study will analyze historical records to answer the question "What extent can the Strikeout Rate in the MLB can be forecasted using Time Series Analysis?" Data will be collected for a period of 120 years (1903 to 2022). The study will utilize Time Series Analysis to use previous years strikeout rate to predict the strikeout rate for the next 10 years. The hypothesis of the study is "The Strikeout Rate in an MLB Season *can* be forecasted with 90% accuracy".

Part II: Data Collection

Single season player stats were downloaded from the baseball-reference website (baseball-reference.com) for the years 1903 to 2022. The number of at bats and strikeouts were collected for every player and aggregated by season. The overall MLB strikeout rate was calculated by dividing the total Strikeouts by the total At Bats and multiplying by 100.

Data was collected using the *Pybaseball* Python library, which simplified the data collection process (Jldbc, 2022). The *batting_stats* function was used to download single season player stats which were saved to a csv file, as shown in Figure 1. After downloading the individual player data, the DataFrame was saved to a csv file that was over 87,000 rows. Each row contained the player's ID, season, name, team, total strikeouts, and total at bats, as shown in Figure 2.

One advantage of using the *Pybaseball* library is it made the web scraping process very easy. Writing a Python script to web scrape Baseball-Reference would be very difficult because the website has all of the season data on different pages with unique URLs. The *Pybaseball* library already had functions to pull season level data for players so very little code was needed.

One disadvantage with the dataset is the frequency it was collected. Ideally data would be collected at a higher frequency than yearly, but the *batting_stats* function only allows for yearly or monthly intervals, and since not every season was played in the same months, this would not work. There is another function from the *Pybaseball* library that might help solve this problem, *batting_stats_range* (Jldbc, 2022). The *batting_stats_range* function returns player stats between the range of dates specified, but since the mid-season point varies from season to season, mid-season splits could not be calculated. Another problem with this was that the *batting_stats_range* function only had data after 1968.

```
# loop through all seasons from 1903 to 2022
# and download players stats for each season
startYear = 1903
endYear = 2022
for i in np.arange(startYear,endYear+1):
    stats = batting_stats(start_season=i,
                          end_season=i,
                          league='all', # both leagues
                          qual=0, # no minimum number of at bats
                          ind=1,
                          stat_columns = ['SO','AB']) # only get strikeouts and at bats
    if(i==startYear): # write to file to ensure file is 'new'
        stats.to_csv('battingStatsRaw.csv',header=True,index=False,mode='w')
    else: # append instead of overwriteing.  Also dont include header
        stats.to_csv('battingStatsRaw.csv',header=False,index=False,mode='a')
    if(i%20==0): #print out when each decade is complete
        print(f'Done with {i}')
```

```
Done with 1920
Done with 1940
Done with 1960
Done with 1980
Done with 2000
Done with 2020
```

```
df = pd.read_csv('battingStatsRaw.csv')[['Season','AB','SO']]
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 87467 entries, 0 to 87466
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Season  87467 non-null  int64
 1   AB      87467 non-null  int64
 2   SO      83794 non-null  float64
```

Figure 1: Downloading player data from 1903-2022

```
1011128,1931,Billy Rogell,DET,17,185
1003271,1931,Bill Dickey,NYY,20,477
1002464,1931,Ripper Collins,STL,24,279
1007145,1931,Lew Krausse,PHA,0,2
1011296,1931,Red Ruffing,NYY,13,109
1012865,1931,Danny Taylor,CHC,46,270
1013456,1931,Joe Vosmik,CLE,30,591
1003785,1931,Woody English,CHC,80,634
1004399,1931,Chick Fullis,NYG,13,302
1000139,1931,Ethan Allen,NYG,15,298
1005458,1931,Gabby Hartnett,CHC,48,380
1003911,1931,Bibb Falk,CLE,13,161
1002848,1931,Tony Cuccinello,CIN,28,575
```

Figure 2: Season, Name, Team, At Bats, and Strikeouts for every player

Part III: Data Extraction and Preparation

For this analysis, Python was used alongside Jupyter Notebook and VScode. Python was chosen because of the large number of public libraries that are available (Python, 2022). The *Statsmodels* library contains many useful functions for performing Time Series Analysis, including the *ARIMA* and *tsaplots* functions (statsmodels, 2022). Scripts were written in the Jupyter Notebook format since being able to run one chunk of code at a time made the whole process much faster than using Python scripts. VScode was chosen as the IDE because of the extensions and themes that it offers over the standard Jupyter Lab environment.

After the data was downloaded, exploratory data analysis was performed to ensure the data was in the correct format. The first thing checked for was missing values. Figure 3 shows the code used to discover missing values in the *Strikeout* column, which was missing almost 4,000 values. Before deciding what to do with the missing values, the total null values were computed for each season. As seen in Figure 4, the seasons 1903 through 1912 are missing a large number of null values. These seasons were then filtered out and the season level totals for all seasons after 1912 were computed. Season level data for 1903-1912 was downloaded manually from baseball-reference.com. The *concat* function from the *Pandas* library was used to create a DataFrame with season level data from 1903-2022 (Pandas, 2022). The code for this can be seen in Figure 5 and Figure 6.

```
df = pd.read_csv('battingStatsRaw.csv')[['Season','AB','SO']]
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 87467 entries, 0 to 87466
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Season  87467 non-null  int64
 1   AB      87467 non-null  int64
 2   SO      83794 non-null  float64
dtypes: float64(1), int64(2)
memory usage: 2.0 MB
```

```
df.head(5)
```

	Season	AB	SO
0	1903	3	NaN
1	1903	0	NaN
2	1903	0	NaN
3	1903	0	NaN
4	1903	6	NaN

```
df.isna().sum()
```

```
Season        0
AB            0
SO         3673
```

Figure 3: Checking for null values

```
# checking which seasons have null values
seasons = []
nulls = []
for season in df['Season'].unique():
    seasonData = df.loc[df['Season'] == season]
    seasons.append(season)
    nulls.append(seasonData['SO'].isna().sum())
numNull = pd.DataFrame({'Season':seasons,'Null':nulls})
numNull = numNull.loc[numNull['Null']!=0]
numNull.sort_values(by='Season',ascending=True)
```

	Season	Null
0	1903	362
1	1904	360
2	1905	385
3	1906	409
4	1907	411
5	1908	428
6	1909	497
7	1910	249
8	1911	269
9	1912	303

Figure 4: Which seasons have nulls?

```
summary = reduced.groupby(['Season']).sum().reset_index()
```

```
summary.head(5)
```

	Season	AB	SO
0	1913	81213	9282
1	1914	122489	14743
2	1915	121704	14020
3	1916	81923	9534
4	1917	82055	8680

Now I will load in the data for 1903-1912.
This was downloaded manually and saved to a csv file

```
summary1912 = pd.read_csv('1903to1912.csv')
```

```
summary1912.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Season  10 non-null     int64
 1   AB      10 non-null     int64
 2   SO      10 non-null     int64
dtypes: int64(3)
memory usage: 368.0 bytes
```

Figure 5: Getting season summaries for 1913-2022

```
allSeasonsSummary = pd.concat([summary1912, summary])
```

```
allSeasonsSummary
```

	Season	AB	SO
0	1903	75439	7899
1	1904	82488	9299
2	1905	81842	9523
3	1906	80061	9110
4	1907	80304	8836
...	...	...	...
105	2018	165432	41207
106	2019	166651	42823
107	2020	59030	15586
108	2021	161941	42145
109	2022	163465	40812

120 rows × 3 columns

Figure 6: Creating DataFrame with all seasons

The first step in the data preparation process is to create a column for the *Strikeout Rate*, the total number of strikeouts per 100 at bats: $\frac{\mathrm{K}}{\mathrm{AB}} * 100$. Figure 7 shows the code used to calculate *Strikeout Rate*. The next step in the data preparation process was to convert the season into a *PeriodIndex* to make the Time Series Analysis process smoother. The *period_range* function from *Pandas* was used to set the index of the DataFrame. Finally, all columns except *Strikeout Rate* were dropped from the DataFrame. The code for the final two steps is shown in Figure 8.

An advantage of using the *Pandas* library was how easy it made data manipulation. Columns could be easily created, dropped, or modified and DataFrames could be easily saved and loaded from CSV files.

One disadvantage of creating a *PeriodIndex* is it made visualizing the data a little less intuitive. The *plot* function from the *Matplotlib* library shows an error when a *PeriodIndex* is used rather than a traditional index. To avoid this issue, the *arange* function from the *NumPy* library was passed instead of the index of the DataFrame.

```
# calculate strikeouts per 100 at bats
allSeasonsSummary['Rate'] = round((allSeasonsSummary['SO'] / allSeasonsSummary['AB'])*100,2)
```

```
allSeasonsSummary.tail()
```

	Season	AB	SO	Rate
105	2018	165432	41207	24.91
106	2019	166651	42823	25.70
107	2020	59030	15586	26.40
108	2021	161941	42145	26.02
109	2022	163465	40812	24.97

Figure 7: Calculating strikeout rate

```
df.index = pd.period_range('1903','2022',freq='Y')
df = df[['Rate']]
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
PeriodIndex: 120 entries, 1903 to 2022
Freq: A-DEC
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Rate    120 non-null    float64
dtypes: float64(1)
```

Figure 8: Converting index to a period and dropping unneeded columns

Part IV: Analysis

After all data preparation was completed, the data set contained 120 rows with 2 columns, *Season* and *Rate*. Figure 9 shows the historic MLB Strikeout Rate. There is a clear positive trend in the *Strikeout Rate*.

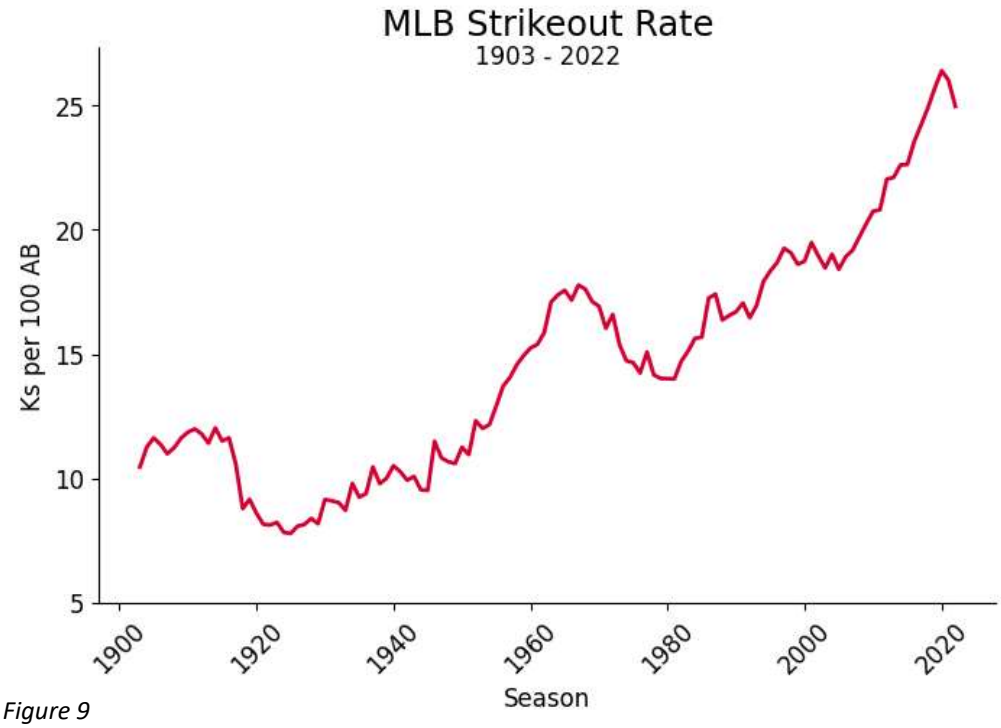


Figure 9

This process is very clearly not stationary, but a Dicky-Fuller test was still conducted. The null hypothesis is that "the series is not stationary" and the alternative hypothesis is that "the series is stationary" (Prabhakaran, 2022). The code and output for this test can be seen in figure 10. The P-Value for this test was 0.99. Since this is not less than any traditionally used alpha value, the null hypothesis cannot be rejected. There is no evidence that this process is stationary. The first difference is then taken in order to obtain a stationary series. Figure 11 shows the code used to take the difference of the series, a visualization of the differenced series, and the Dickey-Fuller test. As seen in Figure 11, this process is stationary and can now be further analyzed.

```
dickey = adfuller(df['Rate'])
print(f'Test Statistic: {dickey[0]}\nP-Value: {dickey[1]}')
```

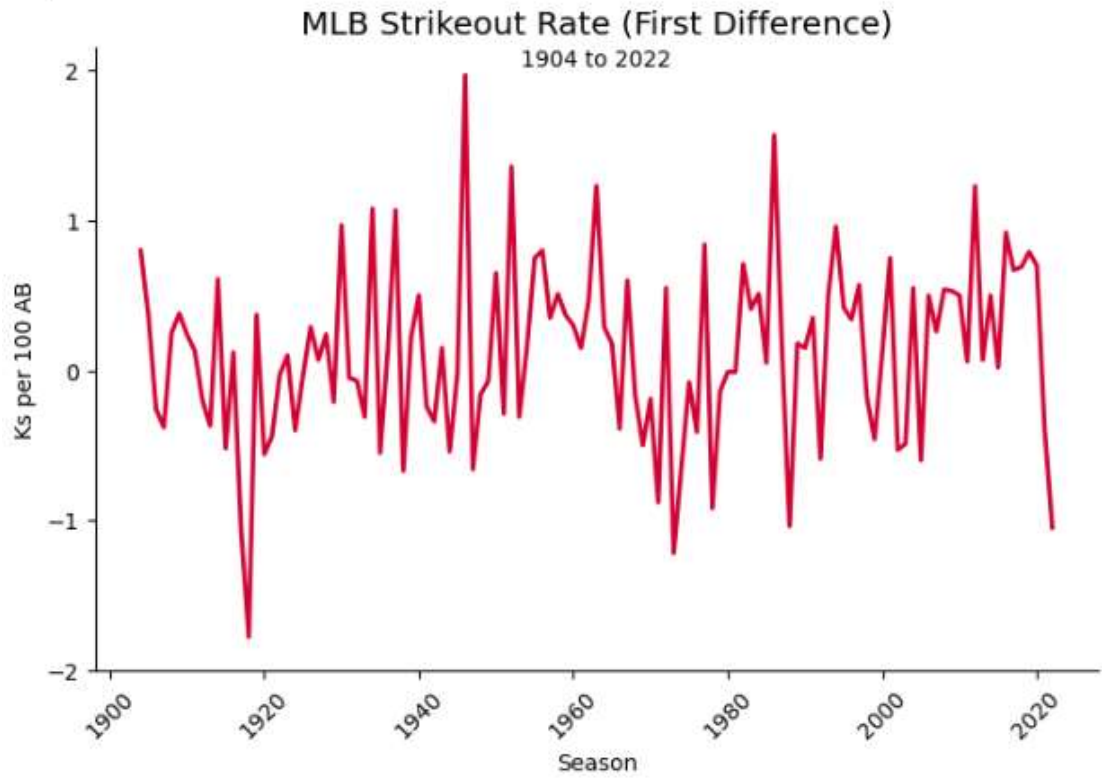
```
Test Statistic: 0.803940105582916
P-Value: 0.9916995032284984
```

Figure 10: ADF Test and Results

```
firstDifference = pd.DataFrame({'Rate':df['Rate'].diff()})
firstDifference = firstDifference.iloc[1:,]
```

```
# plot strikeout rate
plt.clf()
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111)
plt.plot(np.arange(1904,2023,1),firstDifference['Rate'],c='#D50032',linewidth=2)
plt.xlabel('Season')
plt.ylabel('Ks per 100 AB')
plt.title('MLB Strikeout Rate (First Difference)')
ax.text(y=.97,x=.5,s='1904 to 2022',transform=ax.transAxes, ha="center")
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.tick_params(bottom=True, top=False, left=True, right=False)
ax.tick_params(labelbottom=True, labeltop=False, labelleft=True, labelright=False)
plt.yticks(ticks=np.arange(-2,2.5,1))
plt.xticks(np.arange(1900,2021,20))
plt.xticks(rotation=45)
plt.show()
```

```
<Figure size 1500x500 with 0 Axes>
```




```
# check if process is stationary
dickey = adfuller(firstDifference['Rate'])
print(f'Test Statistic: {dickey[0]}\nP-Value: {dickey[1]}')
```

```
Test Statistic: -4.727596686209663
P-Value: 7.466119310757567e-05
```

Figure 11:Calculating, Visualizing, and Testing the first difference

The first steps to analyzing a stationary dataset are to look at the Autocorrelation and Partial-Autocorrelation plots. The *plot_acf* and *plot_pacf* functions from the *statsmodels.graphics. tsaplots* library simplified this process by calculating and visualizing the results automatically (statsmodels, 2022). Figure 12 shows the code used to calculate the ACF and PACF plots. Both the ACF and PACF show a cutoff at lag 3 after, suggesting an AR 2 or 3 component and an MA 2 or 3 component (Chatfield & Xing, 2019).

```
# plot acf function
tsaplots.plot_acf(firstDifference['Rate'],lags=10, auto_ylims=True, zero=False);

# plot pacf function
tsaplots.plot_pacf(firstDifference['Rate'], method='ywm',lags=10, auto_ylims=True, zero=False);
```

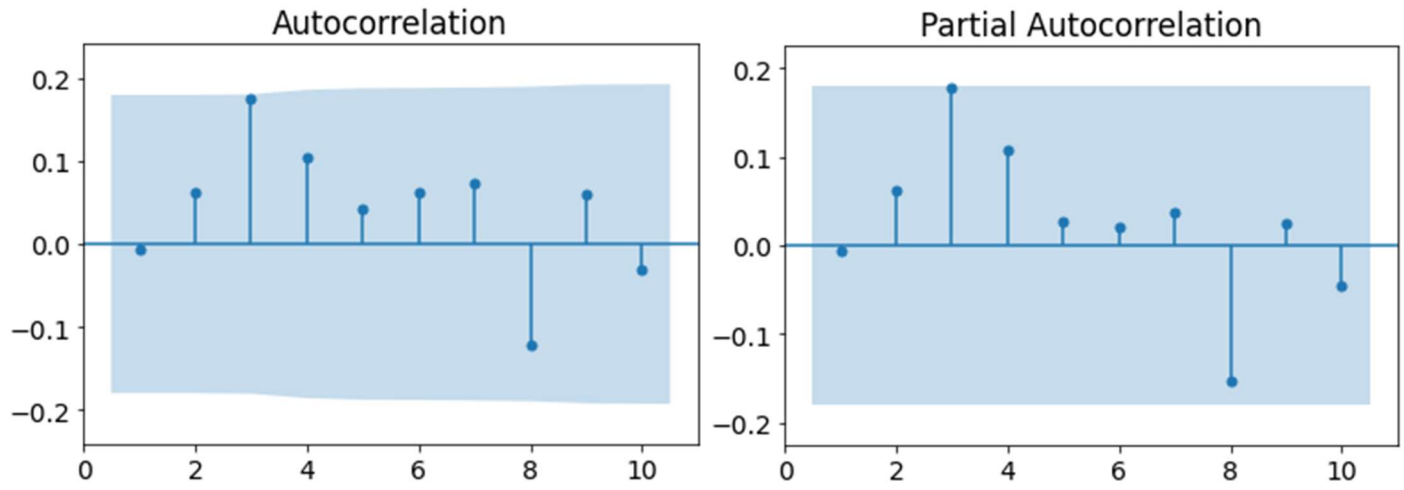


Figure 12: Generating and Visualizing the ACF and PACF

A dataset needs to have at least 180 observations in order to perform a seasonal decomposition to check for seasonality (Tran, 2022). Since this dataset only has 120 observations, a seasonal decomposition could not be performed. The next step of the analysis is to split the dataset into a training and a testing dataset. The *iloc* function from the *Pandas* library made this step very easy (Pandas, 2022). The training dataset contained 70% of the observations and ranged from 1903 to 1986. The testing dataset contained the remaining 30%

and ranged from 1986 to 2022. 70% was chosen in order to leave enough validation data to test how well the model performs (Radecic, 2022). Figure 13 shows the code used to split the dataset.

```
# split into a test and train dataset
train = df.iloc[:int(len(df)*.7),]
test = df.iloc[int(len(df)*.7)-1:,]
```

Figure 13: Splitting the dataset

After splitting the dataset into a train and a test dataset, the *auto_arima* function from the *pmd.arima* library was used to fit many different ARIMA models onto the training dataset. The *auto_arima* function calculates the Akaike's Information Criterion Corrected, AICC, for each model and returns the model with the lowest value (pmdarima, 2022). Figure 14, on the following page, shows the code and output of the *auto_arima* function call. The results of the *auto_arima* showed that an ARIMA (2, 1, 3) is the best fit for the Training dataset with an AICC of 155.63. Figure 15, below, shows the summary of this ARIMA model.

An advantage to using the *auto_arima* function was that I was able to fit many models automatically without having to write a lot of code. This allowed me to test hundreds of models and compare them against each other to obtain the best possible model.

One disadvantage with splitting the dataset this way was that it did not leave much data to train the model with. After splitting the data, only 84 seasons were left. This low number of datapoints is likely the reason that most models perform very poorly with this data.

SARIMAX Results

Dep. Variable:	Rate	**No. Observations:**	84
Model:	ARIMA(2, 1, 3)	**Log Likelihood**	-70.196
Date:	Wed, 19 Oct 2022	**AIC**	152.391
Time:	12:18:26	**BIC**	166.904
Sample:	12-31-1903	**HQIC**	158.222
	- 12-31-1986		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	1.8384	0.093	19.839	0.000	1.657	2.020
ar.L2	-0.9202	0.093	-9.908	0.000	-1.102	-0.738
ma.L1	-2.0862	0.489	-4.267	0.000	-3.045	-1.128
ma.L2	1.4404	0.641	2.246	0.025	0.184	2.697
ma.L3	-0.2406	0.183	-1.313	0.189	-0.600	0.118
sigma2	0.2999	0.172	1.748	0.080	-0.036	0.636

Ljung-Box (L1) (Q):	0.01	**Jarque-Bera (JB):**	7.68
Prob(Q):	0.94	**Prob(JB):**	0.02
Heteroskedasticity (H):	0.93	**Skew:**	0.18
Prob(H) (two-sided):	0.84	**Kurtosis:**	4.45

Figure 15: Summary of the ARIMA (2, 1, 3)

```
auto = auto_arima(train,
                  max_p=5,
                  max_q=5,
                  max_d=2,
                  max_order=14,
                  information_criterion='aicc',
                  seasonal=False,
                  supress_warings=True,
                  approximation=False,
                  error_action='warn',
                  stepwise=False,
                  trace=True)
```

```
ARIMA(0,1,0)(0,0,0)[0] intercept   : AICC=158.515, Time=0.08 sec
ARIMA(0,1,1)(0,0,0)[0] intercept   : AICC=160.406, Time=0.07 sec
ARIMA(0,1,2)(0,0,0)[0] intercept   : AICC=160.036, Time=0.08 sec
ARIMA(0,1,3)(0,0,0)[0] intercept   : AICC=160.659, Time=0.12 sec
ARIMA(0,1,4)(0,0,0)[0] intercept   : AICC=160.611, Time=0.14 sec
ARIMA(0,1,5)(0,0,0)[0] intercept   : AICC=160.346, Time=0.21 sec
ARIMA(1,1,0)(0,0,0)[0] intercept   : AICC=160.338, Time=0.03 sec
ARIMA(1,1,1)(0,0,0)[0] intercept   : AICC=162.397, Time=0.11 sec
ARIMA(1,1,2)(0,0,0)[0] intercept   : AICC=157.279, Time=0.14 sec
ARIMA(1,1,3)(0,0,0)[0] intercept   : AICC=157.394, Time=0.20 sec
ARIMA(1,1,4)(0,0,0)[0] intercept   : AICC=159.624, Time=0.27 sec
ARIMA(1,1,5)(0,0,0)[0] intercept   : AICC=160.998, Time=0.35 sec
ARIMA(2,1,0)(0,0,0)[0] intercept   : AICC=161.517, Time=0.07 sec
ARIMA(2,1,1)(0,0,0)[0] intercept   : AICC=159.864, Time=0.15 sec
ARIMA(2,1,2)(0,0,0)[0] intercept   : AICC=156.710, Time=0.25 sec
ARIMA(2,1,3)(0,0,0)[0] intercept   : AICC=155.633, Time=0.68 sec
ARIMA(2,1,4)(0,0,0)[0] intercept   : AICC=162.027, Time=0.49 sec
ARIMA(2,1,5)(0,0,0)[0] intercept   : AICC=163.510, Time=0.68 sec
ARIMA(3,1,0)(0,0,0)[0] intercept   : AICC=159.428, Time=0.13 sec
ARIMA(3,1,1)(0,0,0)[0] intercept   : AICC=159.917, Time=0.14 sec
ARIMA(3,1,2)(0,0,0)[0] intercept   : AICC=inf, Time=0.50 sec
ARIMA(3,1,3)(0,0,0)[0] intercept   : AICC=157.540, Time=0.65 sec
ARIMA(3,1,4)(0,0,0)[0] intercept   : AICC=inf, Time=0.89 sec
ARIMA(3,1,5)(0,0,0)[0] intercept   : AICC=inf, Time=0.90 sec
ARIMA(4,1,0)(0,0,0)[0] intercept   : AICC=160.443, Time=0.11 sec
ARIMA(4,1,1)(0,0,0)[0] intercept   : AICC=162.171, Time=0.23 sec
ARIMA(4,1,2)(0,0,0)[0] intercept   : AICC=156.471, Time=0.70 sec
ARIMA(4,1,3)(0,0,0)[0] intercept   : AICC=inf, Time=0.71 sec
ARIMA(4,1,4)(0,0,0)[0] intercept   : AICC=inf, Time=0.88 sec
ARIMA(4,1,5)(0,0,0)[0] intercept   : AICC=inf, Time=1.10 sec
ARIMA(5,1,0)(0,0,0)[0] intercept   : AICC=162.218, Time=0.26 sec
ARIMA(5,1,1)(0,0,0)[0] intercept   : AICC=164.383, Time=0.34 sec
ARIMA(5,1,2)(0,0,0)[0] intercept   : AICC=166.622, Time=0.60 sec
ARIMA(5,1,3)(0,0,0)[0] intercept   : AICC=inf, Time=0.80 sec
ARIMA(5,1,4)(0,0,0)[0] intercept   : AICC=inf, Time=0.79 sec
ARIMA(5,1,5)(0,0,0)[0] intercept   : AICC=inf, Time=1.05 sec

Best model:  ARIMA(2,1,3)(0,0,0)[0] intercept
Total fit time: 14.924 seconds
```

Figure 14: The auto_arima function

Figure 15: Summary of the ARIMA(2, 1, 3)

The *get_forecast* function can be used to on the ARIMA model to forecast the next 36 years (1987 to 2022). This function returns an object can be used to create a DataFrame with the mean forecast, standard error of the forecast, and the upper and lower bounds of the forecast confidence interval. The alpha value used to create the confidence interval was 0.1. This creates a 90% confidence interval. Figure 16, below, shows the code used to create the forecast and confidence interval.

```
nAhead = len(test)
forecast = results.get_forecast(steps=nAhead)
alphaVal = .1
forecastFrame = forecast.summary_frame(alpha=alphaVal)
forecastFrame.columns = ['mean','se','lower','upper']
forecastFrame.head()
```

	mean	se	lower	upper
1987	17.370278	0.552176	16.462030	18.278527
1988	17.755143	0.695907	16.610477	18.899808
1989	18.101269	0.836009	16.726157	19.476381
1990	18.383414	0.986717	16.760409	20.006419
1991	18.583587	1.154534	16.684548	20.482625

Figure 16:Creating the forecast

These forecasted values can now be compared to the observed values from the test dataset to determine how accurate the model is. Figure 17 and Figure 18, on the next page, show the forecasted values plotted against the observed values.

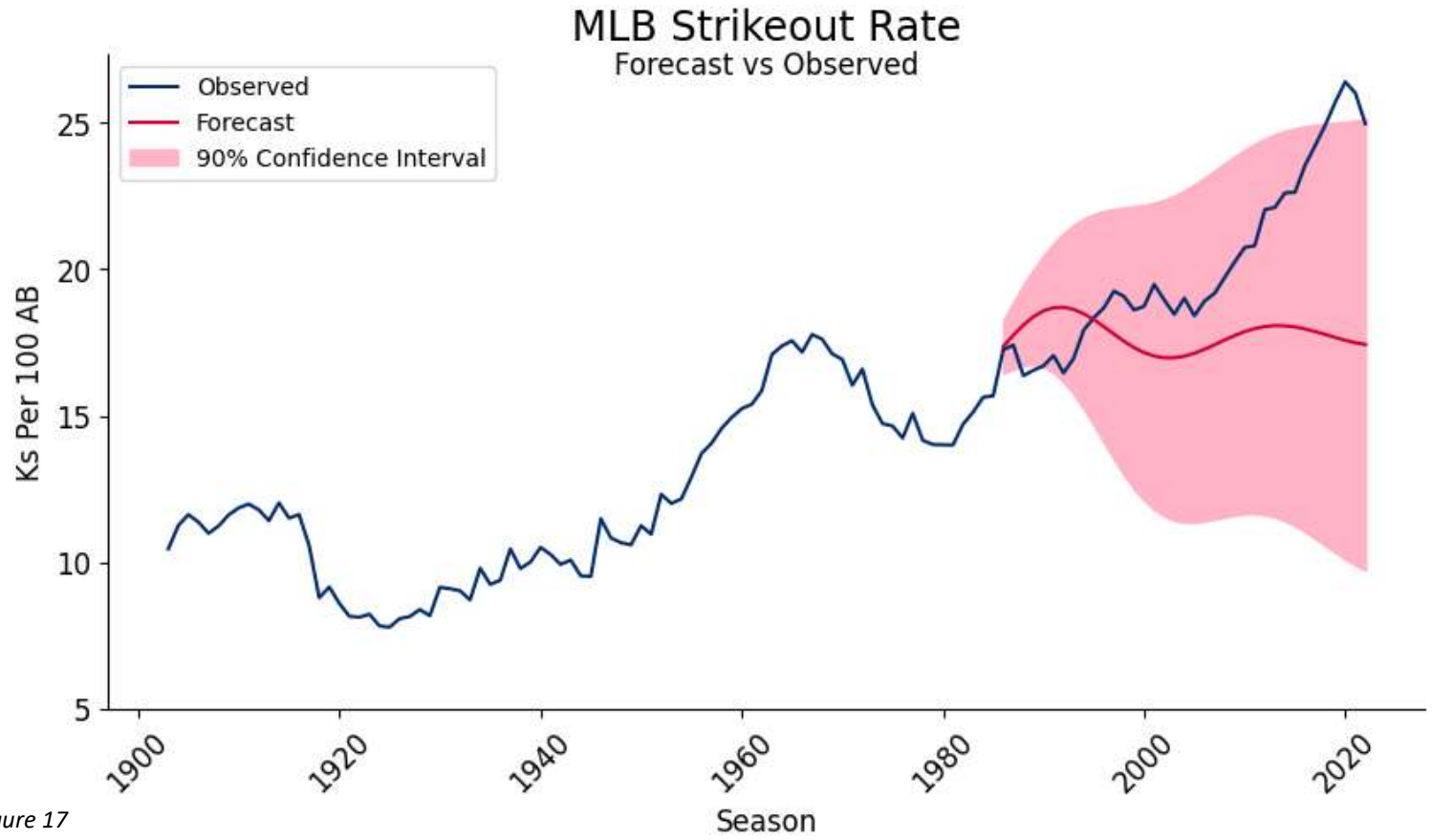


Figure 17

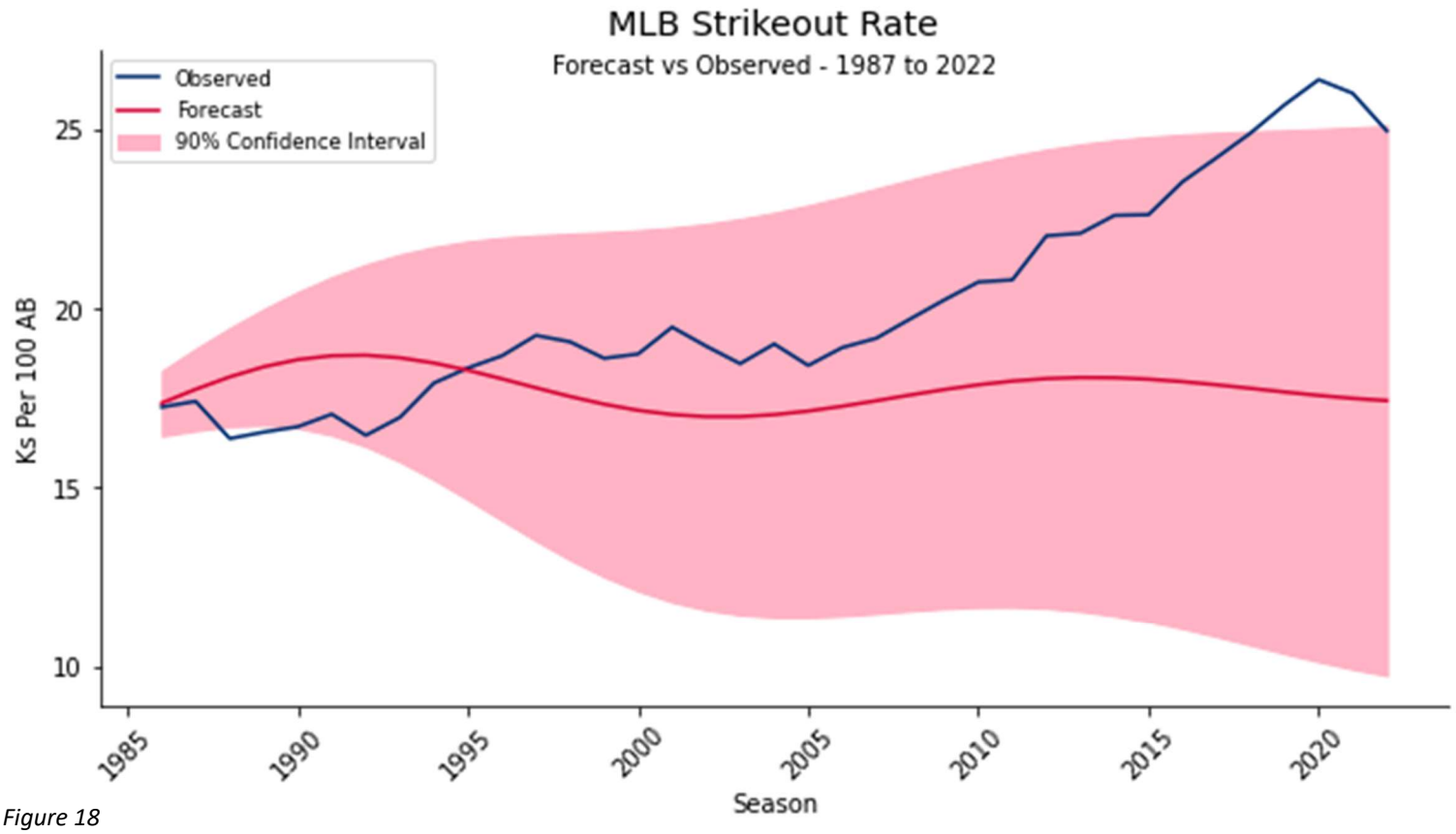


Figure 18

It is clear in Figures 17 and 18 that the model fails to capture the true year to year variance and instead predicts more broad increases and decreases in the MLB Strikeout Rate. Despite failing to capture the true variance, the model does a decent job at forecasting the Strikeout Rate. The 90% confidence interval manages to capture all but a couple seasons. Three of the seasons it doesn't capture are the 2019, 2020, and 2021 seasons, which happen to be the three seasons with the highest recorded Strikeout Rate. The other season that the confidence interval did not accurately capture were the 1988 and 1989 seasons which saw a slight dip from previous seasons.

The forecasted values for 1987 to 2022 can now be compared against the observed values from the testing dataset. The metric used to compare forecasted vs observed is the Mean Absolute Percentage Error (MAPE). MAPE is defined as the average absolute percent error for each time period divided by the total number of periods (Glen, 2022). The formula for MAPE is as follows: $\frac{1}{n} * \Sigma \left[\frac{Observed - Forecast}{Observed}\right]$. Figure 19 shows two ways to calculate MAPE. One way is to manually loop through the observed and forecasted values. Another way to calculate MAPE is with the *mean_absolute_percentage_error* function from the *sklearn.metrics* library (SKLearn, 2022).

```
actual = test['Rate']
APE = []
for i in range(len(forecastFrame['mean'])):
    if(actual[i] != 0): #otherwise we get an infinite MAPE
        error = abs(actual[i] - forecastFrame['mean'][i])/(actual[i])
        #print(error)
        APE.append(error)
MAPE = np.mean(APE)
print(f'Mean Absolute Percentage Error = {round(MAPE*100,2)}%')
```

```
Mean Absolute Percentage Error = 13.65%
```

```
MAPE = mean_absolute_percentage_error(
            y_true = actual,
            y_pred=forecastFrame['mean'])
print(f'Mean Absolute Percentage Error = \
{round(MAPE*100,2)}%')
```

```
Mean Absolute Percentage Error = 13.66%
```

Figure 19: Two ways to calculate MAPE

The final model has a MAPE of 13.66% which indicates low but acceptable accuracy. Ideally, we would have a MAPE less than 5%, but since our model failed to capture the season-to-season variability, this score is expected.

After creating and scoring the ARIMA (2, 1, 3) model, future values for the MLB Strikeout Rate could be calculated. Figure 20 shows the code used to calculate a 10-year forecast of the MLB Strikeout Rate.

```
forecastLength = 10
forecastModel = ARIMA(df,order=auto.get_params()['order'])
forecastResults = forecastModel.fit()
futureForecast = forecastResults.get_forecast(steps=forecastLength)
futureForecastFrame = futureForecast.summary_frame(alpha=alphaVal)
futureForecastFrame.columns = ['mean','se','lower','upper']
futureForecastFrame.head()
```

	mean	se	lower	upper
2023	25.292431	0.585612	24.329185	26.255677
2024	25.277514	0.812124	23.941690	26.613338
2025	25.048203	1.009815	23.387205	26.709202
2026	24.899247	1.240154	22.859376	26.939119
2027	24.792213	1.475296	22.365566	27.218859

Figure 20: Forecasting the future of the MLB

Figure 21, on the following page, shows the 10-year forecast and confidence interval compared to the historical MLB Strikeout Rate. Figure 22 shows a closer view of the 10-year forecast and confidence interval.

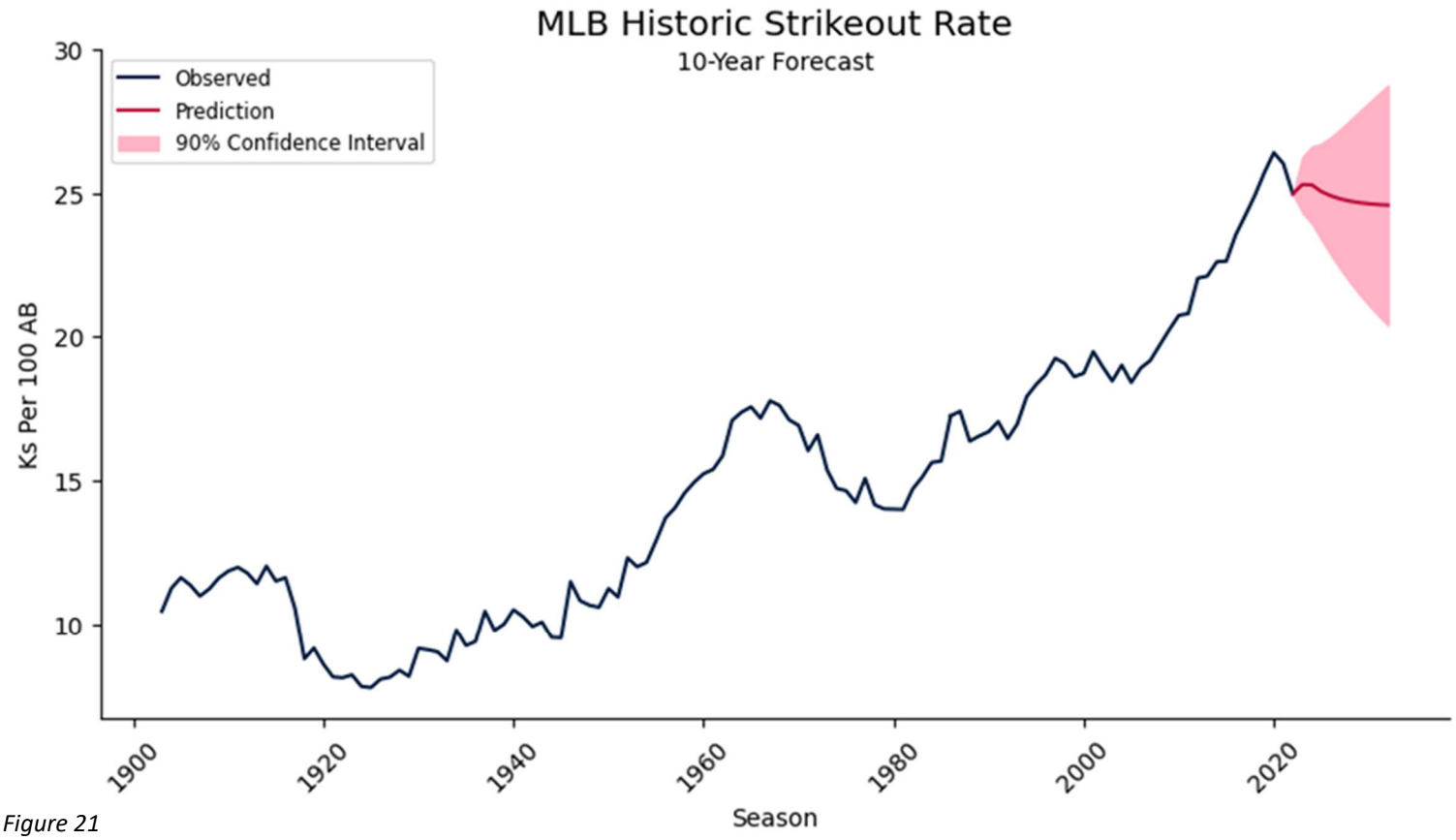


Figure 21

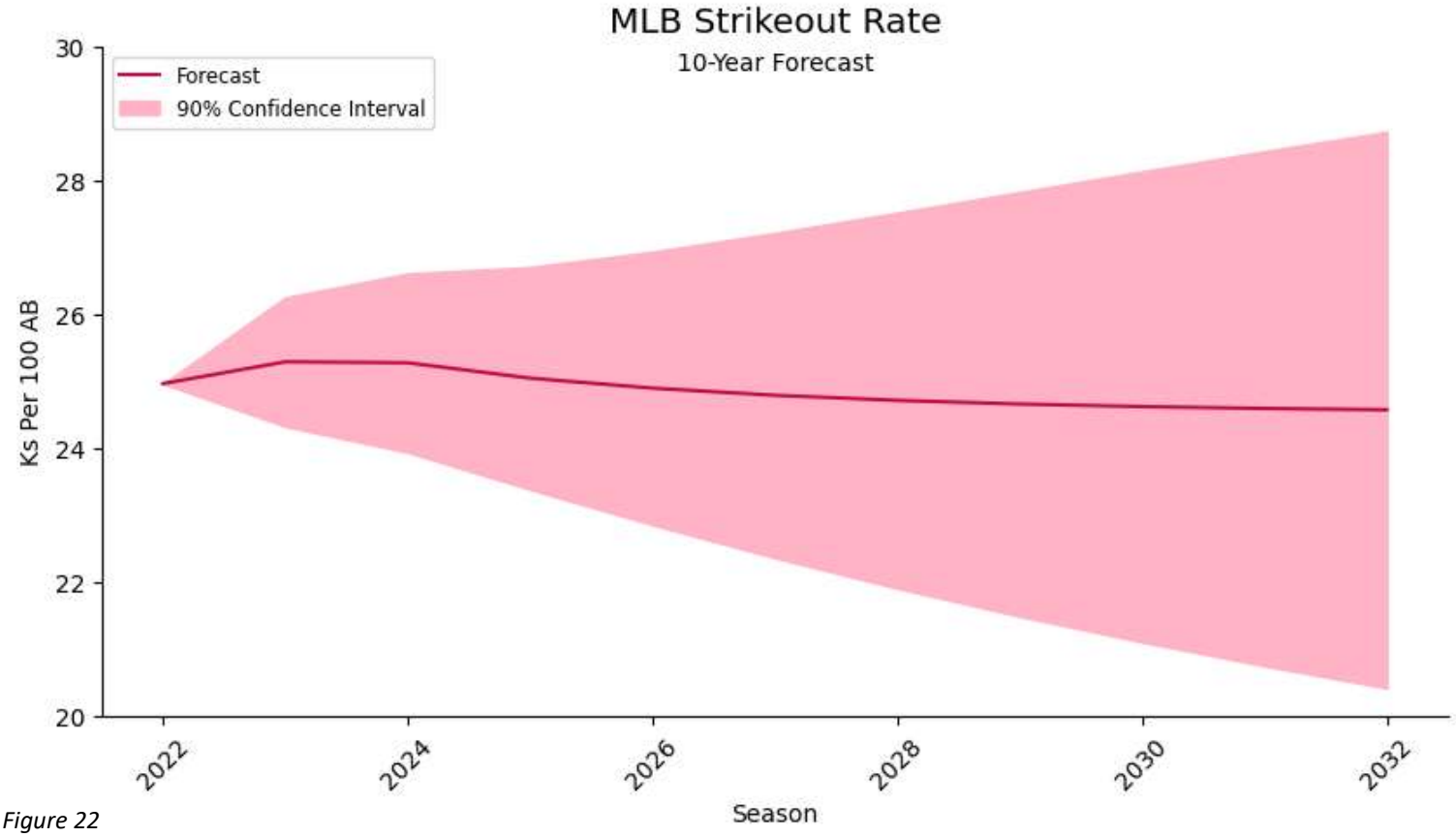


Figure 22

The mean prediction for the 10-year forecast stays fairly consistent at around 25 Strikeouts per 100 At Bats. This is due to the model failing to capture the true variability of the MLB Strikeout Rate which creates a fairly flat forecast. This prediction seems logical as the MLB Strikeout Rate was recently increasing at a rapid rate and reached a historic high before the MLB introduced multiple rule changes regarding pitchers. These changes sought to eliminate pitcher's using foreign substances to gain advantages over hitters (Verducci, 2022). MLB hoped that this rule change would reduce strikeouts and increase total offense. With these rule changes, the strikeout rate will likely reduce back to normal levels before leveling out again.

Part V: Data Summary and Implications

Time Series Analysis can be used to forecast future values by building models with historic data. The final model in this analysis failed to accurately capture the true season-to-season variability due to a lack of data and the frequency data was collected at. Despite this, the model still performed well against the testing dataset and had a MAPE of 13.6%. The confidence interval of the forecast also manages to properly capture the trend of the MLB Strikeout Rate compared to the testing dataset. The results of this analysis show that despite a lack of data, it is possible to forecast the Strikeout Rate in an MLB Season with 90% accuracy.

The MLB could compare the Strikeout Rate the next 10 years to the forecast that the model made to see the effect of rule changes. Since the MLB wants to see an increase in offense, they will likely make more and more rule changes to benefit the offense. If despite these changes the Strikeout Rate continues to increase relative to the models prediction, the MLB will know that their changes were ineffective. This will allow the MLB to know if they need to switch strategies before it is too late.

The main limit to this analysis is the number of datapoints. The lack of datapoints led to a model that did not accurately capture the season-to-season variance of the MLB Strikeout Rate. Another limit is that the data was taken at a yearly frequency. Time Series Analysis would likely perform much better with biannually, quarterly, or weekly frequencies when dealing with a time period of this size. Including more datapoints would result in a model that more accurately captures the trend and variance of the MLB Strikeout Rate.

One suggestion for further research would be to recreate this study with weekly measurements rather than yearly. Weekly data would show how the MLB Strikeout Rate varies throughout the season and would give far more datapoints for the model to train on. It would

also allow for Seasonal Components to be included in a SARIMA model which would result in a model that could more accurately and precisely forecast the MLB Strikeout Rate.

Another suggestion would be to use biannual measures of the strikeout rate. A model may be able to use MLB Strikeout Rate at the All-Star game to predict what it will be once the season ends. This would also give the model double the number of datapoints, leading to a more accurate model.

References

Attanasio, E. (2022). This Great Game. Retrieved October 17, 2022, from https://thisgreatgame.com/

Birnbaum, P. (2022). A guide to sabermetric research. Society for American Baseball Research. Retrieved October 17, 2022, from https://sabr.org/sabermetrics

Chatfield, C. & Xing, H. (2019). The analysis of Time Series: An introduction with R. Chapman & Hall/CRC.

Glen, S. (2022, May 27). Mean absolute percentage error (MAPE). Statistics How To. Retrieved October 20, 2022, from https://www.statisticshowto.com/mean-absolute-percentage-error-mape/

Jldbc. (2022). JLDBC/Pybaseball: Pull current and historical baseball statistics using Python (Statcast, Baseball Reference, fangraphs). GitHub. Retrieved October 17, 2022, from https://github.com/jldbc/pybaseball

Pandas. (2022). Pandas documentation#. pandas 1.5.0 documentation. Retrieved October 17, 2022, from https://pandas.pydata.org/docs/

pmdarima. (2022). Pmdarima User guide. pmdarima 2.0.1 documentation. Retrieved October 19, 2022, from http://alkaline-ml.com/pmdarima/user_guide.html#user-guide

Prabhakaran, S. (2022, April 4). Augmented Dickey-Fuller (ADF) test. Machine Learning Plus. Retrieved October 19, 2022, from https://www.machinelearningplus.com/time-series/augmented-dickey-fuller-test/

Python. (2022). About Python. Python.org. Retrieved October 19, 2022, from https://www.python.org/about/

SKLearn. (2022). Scikit Learn User guide. scikit. Retrieved October 20, 2022, from https://scikit-learn.org/stable/user_guide.html

statsmodels. (2022). User guide. statsmodels. Retrieved October 19, 2022, from https://www.statsmodels.org/stable/user-guide.html

Tran, K. (2022, April 20). How to detect seasonality, outliers, and changepoints in Your time series. Medium. Retrieved October 20, 2022, from https://towardsdatascience.com/how-to-detect-seasonality-outliers-and-changepoints-in-your-time-series-5d0901498cff

Verducci, T. (2022, March 25). MLB cracks down on sticky stuff again, suspects cheating resumed. Sports Illustrated. Retrieved October 20, 2022, from https://www.si.com/mlb/2022/03/25/sticky-stuff-further-crackdown