AMATH 582 Final Project

Daniel Burnham and Shabab Ahmed
Github link: https://github.com/burnhamdr/NBA-Decade-Comparison

March 19, 2020

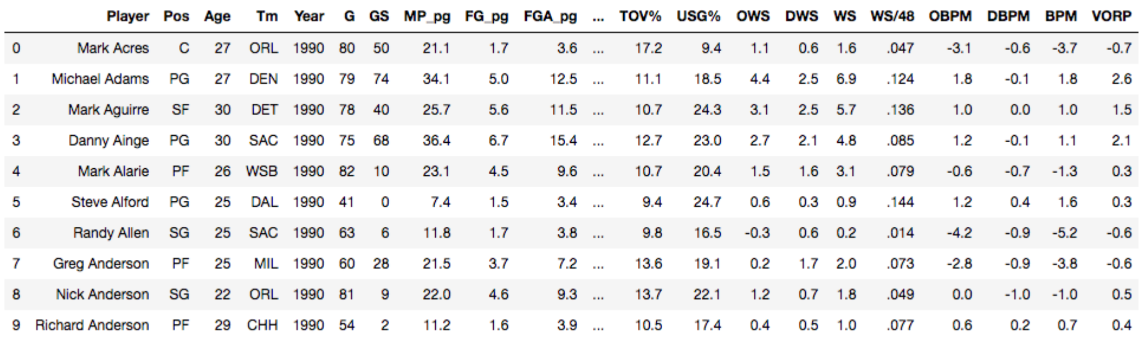
Abstract

This project uses Principal component analysis (PCA) as a foundation to exploring characterization of basketball player performance data in two phases. Finding useful feature spaces within traditionally tracked "box-score" statistics is of interest due to the potential that these feature spaces might present for predicting future player performance. The first phase of work presented here begins with a PCA exploratory investigation of basketball player statistics taken from "basketball-reference.com". We observe that the first two principal components capture about 60% of the variance in the data. After conducting PCA, the project shifts to machine learning techniques with the aim being to predict player characteristics namely the decade played in and listed position. These characteristics are of interest because successful prediction would lend credence to the idea that there is a play style signature within box-score statistics that is unique to position and/or the decade of the sport. In particular, we use Linear Discrimination Analysis (LDA) to classify these characteristics. After cross validating over 500 trials, LDA is only 38% successful in classifying players based on the decade they played in. Classifying the players based on their position achieved 50% accuracy. Naive Bayes classifier performance for both classification tasks achieved a similar accuracy rate. In the second project phase, the feature space was instead investigated for its potential to predict an "advanced statistic" of Win Shares which is associated with an individual player's contribution to a team's success. Using a logistic regression classifier marginal success was achieved in using 1990s decade data to predict the most recent decade of WS statistics. Overall, this work presents as a promising avenue of inquiry for understanding and subsequently leveraging the trends in player performance data to predict future success.

1 Introduction

1.1 Basketball

Basketball is a team sport played on a 94 by 50 foot court with the objective being to pass a singular ball through a basket. At any given moment one team is defending one of the two baskets while the other is attempting to score by shooting the ball through the hoop of the basket. From this simple foundation an elaborate sport filled with athleticism and strategy emerges. In order to conceptualize player and team performance, i.e. how well a player is contributing to the teams success, "box-score" statistics are tabulated during the run of play. For each game the traditional tracked box-score statistics are: total minutes played, points scored, rebounds, assists, steals, blocks, field goal percentage, and fouls. Points scored relates to the number of times a player successfully shoots the ball through the hoop. The rebounds statistic refers to the number of times a player secures the ball after the opposing team shoots and misses on their attempt to score. Assists refer to the number of times a player passes the ball to a player on their team who then subsequently scores. Steals and blocks are defensive statistics capturing how often a player either takes the ball from an opposing player (steal) or physically prevents the ball from entering the hoop (block) during an opposing player's attempt at scoring. Field goal percentage is simply the number of successful tries at scoring divided by the overall number of attempts. Lastly, fouls record the number of times a player makes illegal physical contact with a player of the opposing team. Building from these basic descriptive statistics, modern statistics categories expand these measures by normalizing these categories by minutes played and also by the number of possessions (i.e. how many opportunities a team has during a game to score). Beyond these, "advanced" statistics are also calculated either as combinations of box-score measures or relationships



	Player	Pos	Age	Tm	Year	G	GS	MP_pg	FG_pg	FGA_pg	...	TOV%	USG%	OWS	DWS	WS	WS/48	OBPM	DBPM	BPM	VORP
0	Mark Acres	C	27	ORL	1990	80	50	21.1	1.7	3.6	...	17.2	9.4	1.1	0.6	1.6	.047	-3.1	-0.6	-3.7	-0.7
1	Michael Adams	PG	27	DEN	1990	79	74	34.1	5.0	12.5	...	11.1	18.5	4.4	2.5	6.9	.124	1.8	-0.1	1.8	2.6
2	Mark Aguirre	SF	30	DET	1990	78	40	25.7	5.6	11.5	...	10.7	24.3	3.1	2.5	5.7	.136	1.0	0.0	1.0	1.5
3	Danny Ainge	PG	30	SAC	1990	75	68	36.4	6.7	15.4	...	12.7	23.0	2.7	2.1	4.8	.085	1.2	-0.1	1.1	2.1
4	Mark Alarie	PF	26	WSB	1990	82	10	23.1	4.5	9.6	...	10.7	20.4	1.5	1.6	3.1	.079	-0.6	-0.7	-1.3	0.3
5	Steve Alford	PG	25	DAL	1990	41	0	7.4	1.5	3.4	...	9.4	24.7	0.6	0.3	0.9	.144	1.2	0.4	1.6	0.3
6	Randy Allen	SG	25	SAC	1990	63	6	11.8	1.7	3.8	...	9.8	16.5	-0.3	0.6	0.2	.014	-4.2	-0.9	-5.2	-0.6
7	Greg Anderson	PF	25	MIL	1990	60	28	21.5	3.7	7.2	...	13.6	19.1	0.2	1.7	2.0	.073	-2.8	-0.9	-3.8	-0.6
8	Nick Anderson	SG	22	ORL	1990	81	9	22.0	4.6	9.3	...	13.7	22.1	1.2	0.7	1.8	.049	0.0	-1.0	-1.0	0.5
9	Richard Anderson	PF	29	CHH	1990	54	2	11.2	1.6	3.9	...	10.5	17.4	0.4	0.5	1.0	.077	0.6	0.2	0.7	0.4

Figure 1: Example of the first 10 columns of the dataset used in our analysis

between player box-score statistics and team performance measures. The statistical descriptions of basketball continue to evolve and provide the data on which the work presented here builds.

1.2 Data set

The data used in this project is courtesy of the tabulation, calculation, and public access facilitation work done by the Sports Reference sites organization. Sports Reference has been providing sports statistics reference material online since 2000 and represents the gold standard of publicly available sports statistic data for baseball, basketball, football, hockey, and soccer. The data set used in our work is a concatenation of the sports reference basketball statistics for various categories (per game, per minute, per possession, and advanced) for basketball players in the National Basketball Association (NBA) over the last three decades. Figure (1) contains an example of the first 10 columns of the dataset after it has been extracted and processed. This is the dataset that we use for our analysis.

1.3 Overview

Though the beauty of basketball in a large sense is firmly rooted in the play of the sport itself and cannot be adequately reduced to descriptive statistics, there is value in the analysis of trends in statistics in order to quantify styles of play to facilitate future player and team success. This is exactly the goal the work presented here seeks to accomplish. Efforts made towards this goal are spilt in this report into two sections. The first section addresses the feasibility of identifying position specific statistical signatures across three decades of player data and also whether there are decade specific data features capable of classifying player data by the decade of play. The second section of the report takes a subtly different approach by instead of seeking style of play descriptive features, elucidating how player specific performance measures relate to team success. This is accomplished by using a classification method to predict player Win Share quartile.

1.3.1 Style of Play Description

To begin with, PCA of the data set is carried out to identify the axes of highest variance and visualize along these axes if categorical features of the data describe observed separations or clustering of data points. This qualitative assessment is used to inform categorical features that would be promising data classes to predict from player statistics data. Subsequently various machine learning techniques, namely Linear Discriminant Analysis (LDA), and Naive Bayes classification, are used to predict the player position and decade played in. Successful prediction of these categories would imply that there are strong underlying signatures of play style by position played and time period of play within player box-score statistics. This is a question of interest because the sport of basketball has changed in many metrics describing league wide average statistics over the last three decades. Often these changes in the sport are discussed in terms of team pace, types of shots taken, and overall player skill specialization. These concepts are not explicitly tabulated in any individual box-score measure and as such it is of interest to investigate whether the collection of box-score statistics across time is a predictor of the observed higher level changes in the sport.

1.3.2 Player and Team Success Relationship

Beyond discussion and description of the evolution of the sport, modern basketball is fervently focused on assessing and maximizing player talent in an effort to achieve team success. To participate in the exploration of this relationship between individual player performance and team success, logistic regression classification is used to predict player specific Win Shares (WS) within one decade of player data. This model is then used to predict future decades with the hope being that past relationships between team wins and player performance can be used to predict the ability of a future player to contribute to team wins. Win Shares are an example of what was referred to earlier as advanced statistics. Win Shares attempts to allocate credit for team success to the players on the team. The statistic is calculated using player specific, as well as team and league-wide data and the sum of player win shares on a given team is roughly equivalent to the team's total annual wins.[2] Because the calculation of the statistic depends on more than simply individual player box-score statistics, it is interesting to ask to what degree individual player performance metrics reflect and thus predict the Win Shares that player is responsible for. In many instances team management wants to determine to what degree a specific player will contribute to their team's success simply based on how the player has played on other teams in the NBA or other basketball leagues (e.g. NCAA, NBL Australia, Spanish Liga ACB, Chinese Basketball Association, etc.). Win Shares presents as a promising advanced stat for categorizing players towards this goal, and if an expected NBA WS measure can be predicted simply with player box-score metrics this could help ensure the best talent is selected for constructing a team of players.

2 Theoretical Background

This section contains some of the theory behind the work done in this project. For supplemental theoretical background see Appendix B.

2.1 Principal Component Analysis

Principal component analysis (PCA) is a statistical method that uses the Singular Value Decomposition (SVD) to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. SVD is descibed briefly in Appendix B. The first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components.

Therefore, PCA is used to find low-dimensional reductions of seemingly complex data. The intuition for PCA builds from the identification of two characteristics of complex data sets: noise and redundancy. If the data is too noisy, then it might be extremely hard to extract the true dynamics. The key measure of the signal-to-noise ratio: SNR $= \sigma^2_{signal}/\sigma^2_{noise}$, where the ratio is given as the ratio of variances of the signal and noise fields. A high SNR gives almost noiseless data whereas a low SNR indicates that the underlying signal is noisy. The second issue is redundancy and the covariance matrix is important in identifying redundancy between data sets.

Covariance measures the statistical dependence/independence between two variables and strongly statistically dependent variables can be considered as redundant observations of the system. The covariance matrix is given by

$$C_X = \frac{1}{n-1} X X^T \tag{1}$$

C_X is a square symmetric matrix whose diagonal represents the variance of particular measurements. The off-diagonal terms are the covariances between measurement types. Large off-diagonal entries correspond to redundancy and large diagonal terms represent the dynamics of interest. Hence, we want to represent C_X so that it is diagonal and the diagonals are ordered from largest to smallest. This is exactly what the SVD does. The SVD diagonalizes by working in the appropriate pair of bases $\hat{U}$ and V and each singular direction captures as much energy as possible as measured by the singular values. If we define the transformed variable

$$Y = \hat{U}^* X \tag{2}$$

where $\hat{U}$ is from Eq.(14). Then:

$$C_Y = \frac{1}{n-1}\hat{\Sigma}^2 \tag{3}$$

The covariance matrix of Y is then diagonal and so we have found basis where each direction is independent of each other. We standardize X by subtracting the row mean, scale it by a factor of $\frac{1}{n-1}$ then compute the reduced SVD of its transpose to get:

$$X^T = \hat{U}\hat{\Sigma}V^* \tag{4}$$

where each column of $\hat{U}$ contains the modes(principal components), the diagonal entries of $\hat{\Sigma}$ contain the singular values whose magnitudes reflect their relative importance. The square of the singular values gives the variance of the data projected onto the PCA modes as shown by Eq.(3).Each column of V contains the projection of the data onto the modes. This is what `Python`'s `PCA` function outputs and what we use in our further analysis.

2.2 Classification methods

2.2.1 Linear Discrimination Analysis

In our training algorithm we use statistical information from PCA in order to make a decision about what categories the players in our dataset belongs to. We essentially want to project the training set into a new basis function in a way that the projection produces well separated statistical distribution between the different groups of the training set. The idea of the LDA was first proposed by Fisher in the context of taxonomy.The goal of LDA is to find a suitable projection that maximizes the distance between the inter-class data while minimizing the intra-class data.

In the case of classification of two distinct classes, the above idea results in consideration of the following mathematical formulation: Construct a project $\mathbf{w}$ such that

$$\mathbf{w} = \arg\max_{\mathbf{w}} \frac{\mathbf{w^T S_b w}}{\mathbf{w^T S_w w}} \tag{5}$$

where the scatter matrices for between-class variance $\mathbf{S_b}$ and within-class variance $\mathbf{S_w}$ data are given by

$$\mathbf{S_B} = (\mu_\mathbf{2} - \mu_\mathbf{1})(\mu_\mathbf{2} - \mu_\mathbf{1})^\mathbf{T} \tag{6}$$

$$\mathbf{S_w} = \sum_{\mathbf{j=1}}^{\mathbf{2}} \sum_{\mathbf{x}} (\mathbf{x} - \mu_\mathbf{j})(\mathbf{x} - \mu_\mathbf{j})^\mathbf{T} \tag{7}$$

where μ terms are arrays of the average value for each class. From these scatter matrices, Eq.(5) can be solved by way of solving of the eigenvalues problem

$$\mathbf{S_B} = \lambda \mathbf{S_w w} \tag{8}$$

where the eigenvector of the maximum eigenvalue calculated from Eq.(8) is the projectio basis for LDA to maximally separate the two classes.

2.2.2 Logistic Regression

Logistic Regression is a widely used method for classification of a dichotomized output variable Y, and works by attempting to model the conditional probability of Y taking on a particular value (i.e. $Pr(Y = 1|X = x)$) as a function of the inputs. A problem in using simple linear regression for this model is that the modelled probability must be between 0 and 1, and a linear function is unbounded. Also it is common to confront systems where the relationship between the conditional probability and the inputs is not linear. The idea behind logistic regression is to instead let the log of the probability be a linear function of of the inputs. By performing a logistic transformation of the log probability the model becomes bounded between 0 and 1. The model is thus given by Eq.(9) where x indicates the predictors and β the model parameters. [3]

$$log\left(\frac{p(x))}{1-p(x))}\right) = \beta_0 + x\beta \tag{9}$$

The classification in a two class case will occur for one class when $p \geq 0.5$ and the other class when $p < 0.5$. To predict classes instead of probabilities, maximum likelihood estimation is used to estimate the parameters of a probability distribution by maximizing a likelihood function. In this case the log likelihood function will be maximized. [3]

$$\ell(\beta_0, \beta) = \sum_{n}^{i=1} y_i log(p(x_i)) + (1-y_i)log(1-p(x_i)) \tag{10}$$

In Eq.(10) the features are captured by the x_i values and each class is represented by a y_i. At this stage a "penalty" can be added to the classification in an effort to promote sparsity in the model. In the work presented here an L1 penalty is used as defined in Eq.(11).

$$\ell^*(\beta_0, \beta) = \ell\beta_0, \beta) - \lambda \sum_{n}^{i=1} \|\beta_i\| \tag{11}$$

Following this penalty, the derivative of the likelihood function is solved numerically to find the maximum likelihood predictions for each class. Overall, logistic regression draws a linear decision boundary between classes and is a unique linear classifier in that the probability of predicting either class depends the proximity to the boundary. [3]

2.2.3 Naive Bayes Algorithm

This is a supervised algorithm that is based upon Bayes rule. Suppose we two classes of data that are 0 and 1 we create a score that gives us good separation. We construct the following ratio:

$$\frac{P(1|x)}{P(0|x)} \tag{12}$$

where $P(1|x)$ is the probability of having class 1 given the data x and $P(0|x)$ is the probability of having class 0 given the data x. This can be rewritten as:

$$\frac{P(1|x)}{P(0|x)} = \frac{P(x|1)P(1)}{P(x|0)P(0)} \tag{13}$$

These quantities are really use to compute and we involve prior information to create our score. Using the ratio, we can get a metric that gives us a good way of separting the data. In our case, we have three classes for each test and we use `Python`'s in-built function to do this analysis. We use the Gaussian Naive Bayes where we assume that probability distribution is Gaussian.

3 Algorithm Implementation and Development

This section contains the algorithmic implementation of the data processing and analysis concepts used in the project.

3.1 Data Extraction and Processing

The first stage in seeking to address the questions of this project is aggregating the data of interest. As mentioned earlier the data was taken from various basketball-reference.com player statistics pages. The website is organized such that there is a site for a given statistical category for each year. This structure allowed for the implementation an iterative web scraping script for streamlining the concatenation of player data from each category (per game, per minute, per possession, and advanced) across the time range of interest (1990-2019). Upon extracting the raw data, a python DataFrame object (from the pandas library) was used to structure the data for each player and year. The following steps were then completed to prepare the data for analysis:

1. Filter out players who played less than 20 games in a year
2. Cast numerical data represented as strings to numeric values
3. Address missing, or NaNs that result from typecasting
4. Standardize position categories, in the case of multi-position players, assign them as their primary position.
5. If a player moved teams during a year (i.e. box-score statistics recorded for more than one team) take the averages of the stats from each team, and create a single entry for that player for the given year.

3.2 Performing PCA

These are the steps taken after the data has been extracted and processed as described above. We have a dataframe with players grouped together by the decade they played in.

1. We standardize the data and convert into matrix form
2. Apply PCA to the data
3. Extract the first two principal components and create a dataframe with the columns of principal components, decades and position
4. Calculate the percentage variance explained and plot the spectrum of modes
5. Plot the first two principal components by decade and by position
6. Divide the dataframe into three different dataframes according to the decade
7. Repeat steps 2-5 for each of the dataframes obtained in step 6

3.3 Linear Discriminant Analysis

3.3.1 Decade

1. Repeat steps 1 and 2 from Section(3.2)
2. Create a matrix where the columns are the projections of the data onto the first two principal components
3. Extract the values that are going to be used as classes. This corresponds to the values in the 'Decade' column
4. Create unique class names according to the classes derived above and translate to integer values
5. Initialize accuracy vector
6. Define how the original data is going to be divided into training set and test set. In our case, 1/7th of the data is used for training.

7. Create training and test set (`Python`) does this randomly)
8. Run `Python`'s in built LDA function and calculate accuracy rate
9. Repeat steps 7-8 500 times and append the accuracy rate to the accuracy vector
10. Calculate the average accuracy and plot the accuracy for each trial
11. Plot the confusion matrix

3.3.2 Position

This follows the same procedure as above. However, we first sort the dataframe using the column 'Pos'. Instead of using the values in the 'Decade' column for Step 3 we use values in the 'Pos' columns as classes. This is because we are trying to classify the players based on position. The rest of the algorithm is the same.

3.4 Gaussian Naive Bayes

This follows the same procedure as above, except, in Step 8 we use `Python`'s in built Gaussian Naive Bayes command to run the analysis.

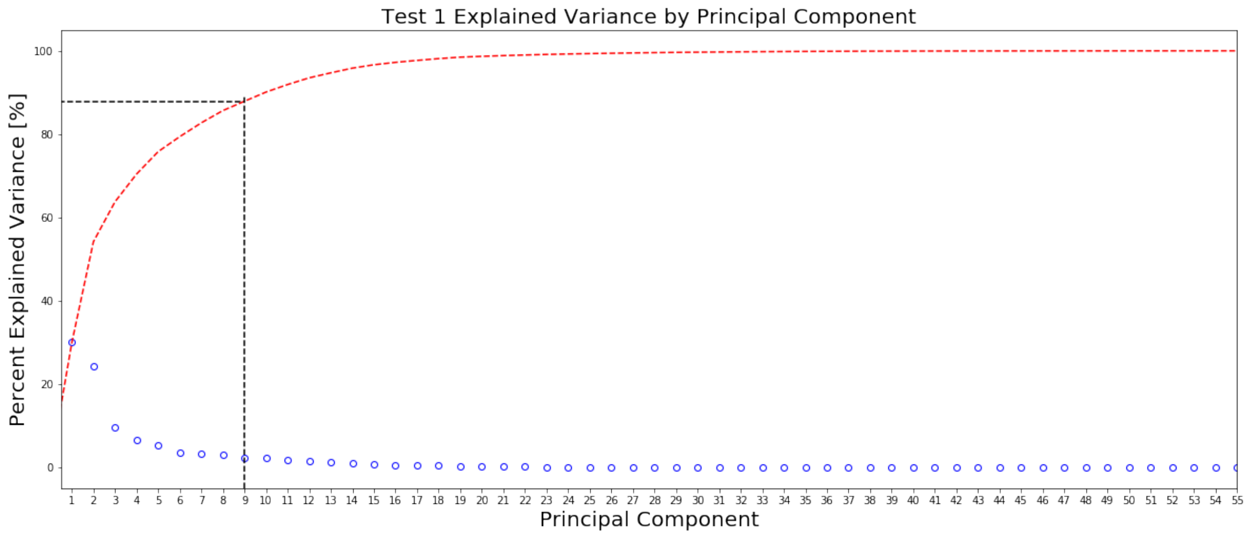


Figure 2: The percentage explained variance by the principal components.

4 Computational Results

This section contains the computational results.

4.1 PCA

We conducted Principal Component analysis on the dataset. Figure (2) contains the percentage explained variance by the modes. we can see that they are two significant modes as the rest tail off to zero. Using the red-dashed line we can see that these two modes explain about 60% of the variance and so we are going to focus on these two components only.

We plot the projection of the data onto the two principal components by the decades the players played in. The left panel of Figure(3) contains the scatter plot for these projections. We can see that they are clustered together indicating that it might be harder to classify them based on decade. There is a lot of overlap between the classes as they are practically on top of one another. Hence, we try to plot the projection

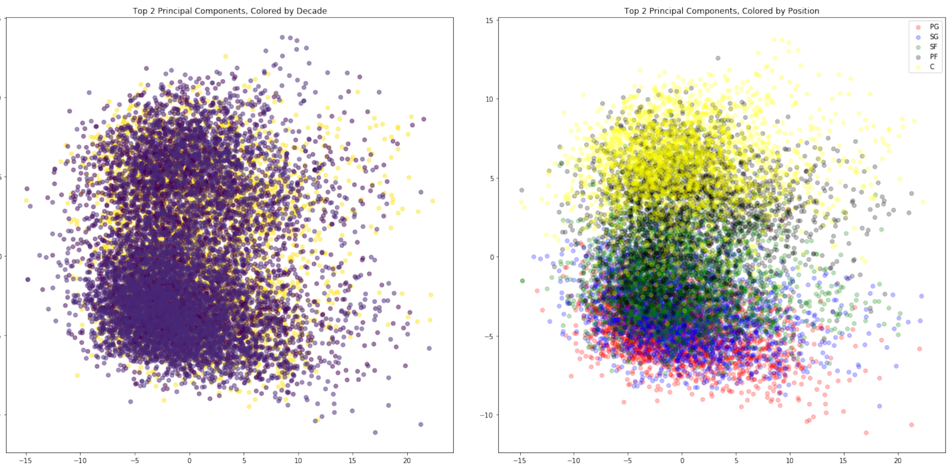


Figure 3: The left panel contains the projection of the data onto the first two principal modes by decade. The right panel contains the same projection by decade. Legend for the right panel: yellow: C, black: PF, green: SF, blue: SG, red: PG.

by position of the players. The right panel of Figure(3) contains the projection by position. We can see that there is more separation between the classes in this case. The position C seems to be most easily distinguishable compared to the other positions. As a result of the right panel, we decide to check out the projections of the data onto first two modes by position for each of the decades. The top left panel contains the project for 1990-1999. We can see that the clustering is more sparese as the positions are more distinguishable. The clustering becomes more and more dense as we pass through the decades. The bottom panel contains the plot for 2010-2019. We can see that plot is the most dense in this case and all the dots seem to be collapsing towards one dot. This can be seen as a validation about recent claims in basketball that players in general are becoming more adapt to playing different roles, that is, specialized roles are becoming less and less apparent. Having produced these plots we move onto classification algorithms to classify the players.

4.2 Classification

We begin by performing Linear discriminant analysis using the decades as labels. Figure (5) contains the confusion matrix for a particular simulation and the accuracy over the trials. We can see that the decades gets misclassified a lot and that players are classified to play in 2000-2009 a lot more than the others. The accuracy plot shows us that the accuracy hovers around 36% to 42% for the trials. The average accuracy over the trials was around 38$.

The data classification did not produce high accuracy when used decades as labels. Hence, we decided to use position as labels. Figure(6) contains the confusion matrix for one particular simulation and the plot of accuracy. We see that positions C and PG are classified most accurately. This makes sense just by looking at the right panel of Figure (3) where the yellow dots corresponding to C are most distinguishable as are the red dots corresponding to PG. We also see that positions PG, SF and SG rarely get classified as C or PF. However, it is important to note this is for one particular simulation and this might not hold in general. The accuracy hovers around 46-50% and the average accuracy is around 50%. This is still not great and we further explore by looking at particular decades. The results were not that insightful as the average accuracy rates were about the same. Figure(7) contains the confusion matrix for a particular simulation using 2010-2019. One interesting thing is that C gets classified correctly the most but there are no clear winners amongst the rest. This maybe due to the collapsing effect of player's skills we talked about earlier. The Reciever Opeartion Characteristic (ROC) curve for LDA classifier for all these cases are included in Appendix C.

We also ran a Naive Bayes Classifier on the dataset using the position as labels. The average accuracy rate was also about 50%. The plots for this analysis is included in Appendix C.

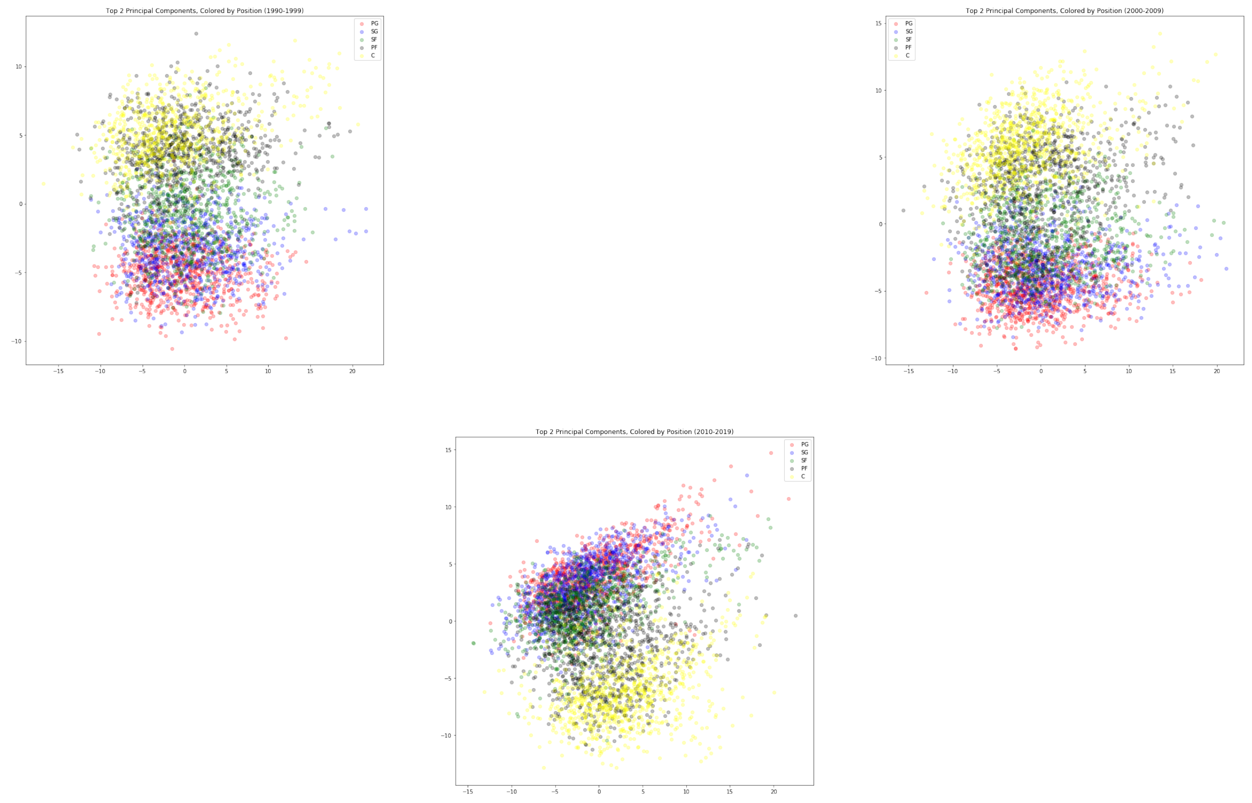


Figure 4: The top left panel contains the projection by position for 1990-1999, the top right panel contains the projection by position for 2000-2009 and the bottom panel contains the projection by position for 2010-2019. The legend is the same as Figure(3).

Given the fact that we were not able to achieve high accuracy in classifying the players based on the decade or the position they played in, we decide to use different labels to classify the players.

4.3 Win Shares Statistic Investigation

4.3.1 Win Shares Top Performers

Win Shares appear to be a good indicator of player quality as it integrates not only individual player box-scores but how this relates to the success of the team in the form of wins. To push this metric further, a measure of consistent player contributions to winning would be counting how many years a given player had a WS statistic in the top 75% of all players. Players that accrue many of what is referred to in our data as "WS years" are players that consistently contribute to wins and are seen as a simple model of a successful player.

As a brief aside, the list of players with more than 16 years of top quartile WS performance is: Shaquille O'Neal*, Jason Kidd*, Kevin Garnett, Ray Allen*, Kobe Bryant, Tim Duncan, Vince Carter, Dirk Nowitzki, Paul Pierce, Andre Miller, Pau Gasol, Joe Johnson (with * indicating hall of fame players). The player currently with the most WS years is Jason Kidd who played for 21 years in the NBA and produced top 75% WS statistics every year of his career.

4.3.2 Win Shares Quartile Classification

Building from these illustrative results WS is pursued as a potential statistic for classifying the success of a player on a team. To begin with, a Logistic regression classifier is developed taking all statistical player data except advanced statistics (per game, per minute, per possession) as input and predicting the

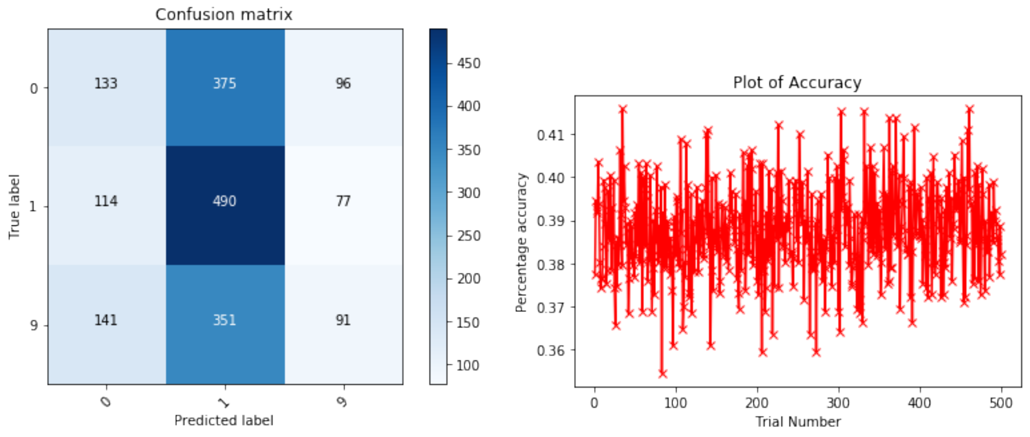


Figure 5: The left panel contains a plot of the confusion matrix for a particular simulation and the right panel contains the plot of accuracy over the trials (by decade).

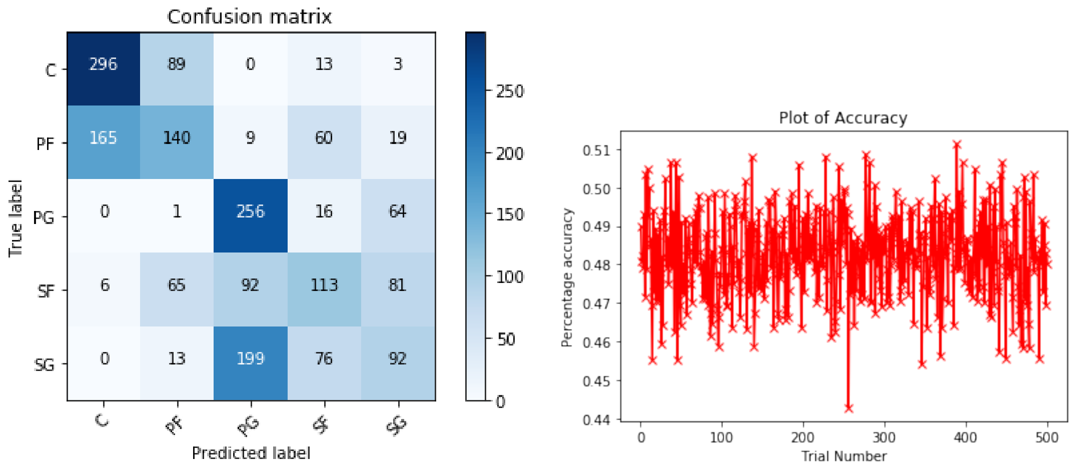


Figure 6: The left panel contains a plot of the confusion matrix for a particular simulation and the right panel contains the plot of accuracy over the trials (by position).

WS quartile as output. This was first done for only players of the 1990s decade (1990-1999). The model performed moderately well on this classification task with accuracy score: 0.726343, precision: 0.713258, recall: 0.726343, and F1 score: 0.714578 (Figure 9). Cross-validation was done for 50 iterations with cross-validated accuracy results presented in Appendix C.

This was model trained for only players of the 1990s decade was then fit to data from the next decade (2000-2009). The model performed moderately well on this classification task with accuracy score: 0.719424, precision: 0.711661, recall: 0.719424, and F1 score: 0.714492 (Figure 10). Cross-validation was done for 50 iterations with cross-validated accuracy results presented in Appendix C.

This was model trained for only players of the 1990s decade was again fit to data from the last decade (2010-2019). The model performed moderately well on this classification task with accuracy score: 0.687902, precision: 0.688837, recall: 0.687902, and F1 score: 0.687907 (Figure 11). Cross-validation was done for 50 iterations with cross-validated accuracy results presented in Appendix C.

Lastly this process was repeated but instead a model trained for only players of the 2000s decade was fit to data from the last decade (2010-2019). The model performed moderately well on this classification task with accuracy score: 0.689052, precision: 0.677217, recall: 0.689052, and F1 score: 0.674251 (Figure 12). Cross-validation was done for 50 iterations with cross-validated accuracy results presented in Appendix C.

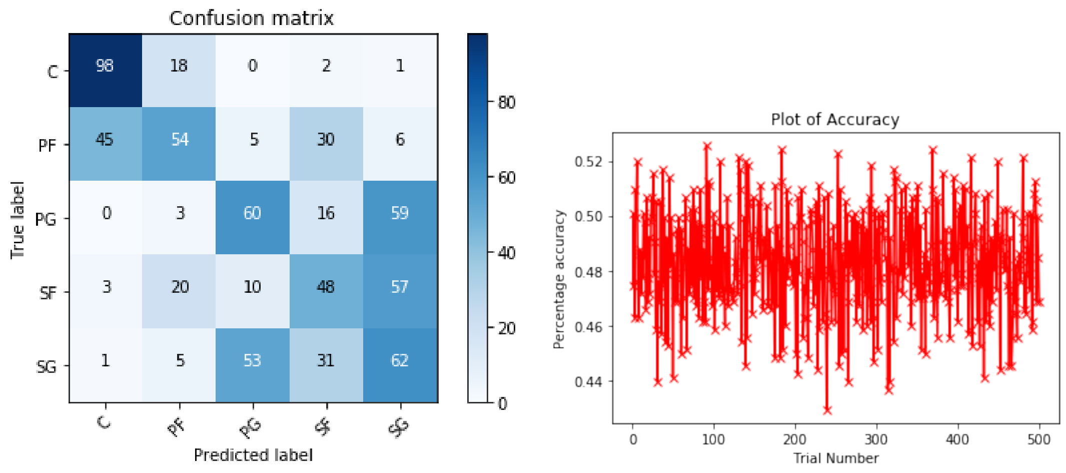


Figure 7: The left panel contains a plot of the confusion matrix for a particular simulation and the right panel contains the plot of accuracy over the trials (by position for 2010-2019).

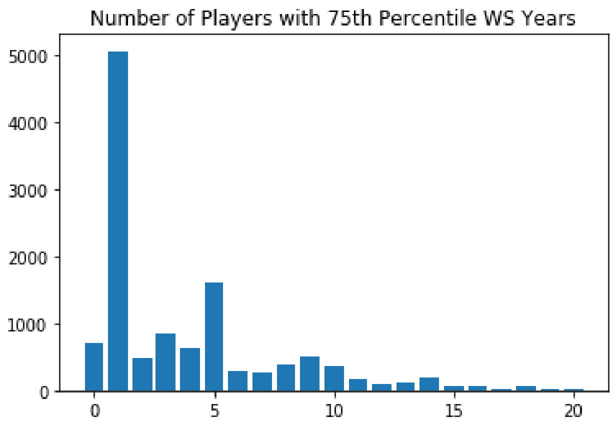


Figure 8: Plot of WS year statistic distribution. WS year is a measure of how many years a specific player has had a WS in the top 75% (i.e. top quartile) of all players for that year. This illustrates how uncommon consistent top performance is in the league, and why this statistic may be an indicator of player quality.

To connect the work using PCA to investigate player box-score data dimensionality reduction, the classification models predicting WS for various decades were tested in the case that training instead occurred on the top principal components of player statistics instead of the raw data. The training data selected for this analysis is player data from the 1990s and the WS values for the resulting model to predict were of the 2010s decade players. The model developed from the top two principal components performed poorly relative to previous results on raw data with accuracy score: 0.450552, precision: 0.302938, recall: 0.450552, and F1 score: 0.319604 (Figure 13). The performance of the model improved slightly when the top 10 principal components were used for training with ccuracy score: 0.542778, precision: 0.524458, recall: 0.542778, and F1 score: 0.445564 (Figure 14). As before, cross-validation was done for 50 iterations with cross-validated accuracy results presented in Appendix C.

5 Summary and Conclusions

Using Principal Component analysis, we were able to see that two principal components explained about 60% of the variance in the massive data set we started off with. Plotting the projection of the two principal

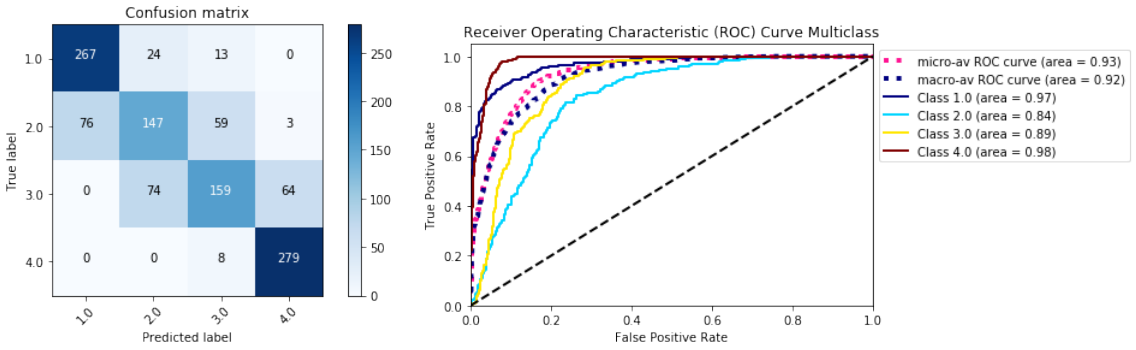


Figure 9: Predicting WS quartile for players of the 1990s (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for logistic regression classifier.

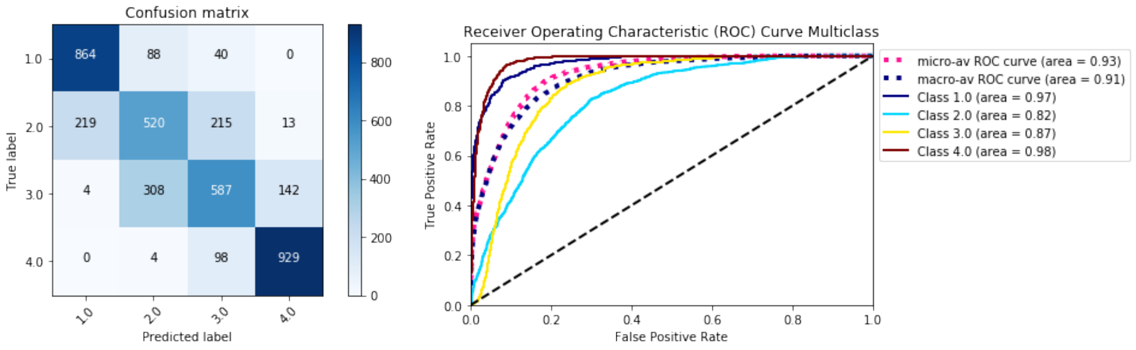


Figure 10: Predicting WS quartile for players of the 2000s using model fit to 1990s player data. (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for logistic regression classifier.

components allowed us to see that there some differentiation could be driven by player position but no identifiable separation was observed based on the decade the players played in. We also noticed that in splitting the data set by each decade and then viewing the projections along the first two principal components by position, a sort of collapsing effect is evident by which the position clusters began to overlap more in recent decades compared to the 1990s. This could be seen as proof of the statement that basketball players are increasingly becoming less specialized in their abilities and are more broadly skilled. Subsequently using machine learning techniques to classify players based on position did not yield highly accurate results. Our average accuracy was roughly 50% and this was consistent across splitting the data by each decade and by using Gaussian Naive Bayes Classifier. This could have been due to the number of modes we used. This prompted us to look at a different metric in order to classify the players. In the next phase of our investigation, the Win Shares statistic was demonstrated to be sufficiently descriptive of player quality (Figure 8). Using raw box-score statistics as training data inputs was generally successful in predicting WS across decade. However, when PCA was used to reduce the dimensionality down to 2 and 10 dimensions, logistic regression classifier performance was diminished. Overall, WS still seem to be an interesting statistic to predict, but the performance will need to be tuned either by using different classification methods or modulating model parameters. In the future, it would be interesting to instead of classifying players WS into quartiles, predict how many years of top quartile performance can be expected based on a player's annual box score statistics. This could be especially useful in predicting the success of young, unproven players. A data set of college basketball player statistics could be aggregated with the statistics of professional players and be subsequently used to train a classifier to predict how well new instances of college basketball players will perform in the NBA. Aside from this, we are interested in also exploring unsupervised learning models to continue searching

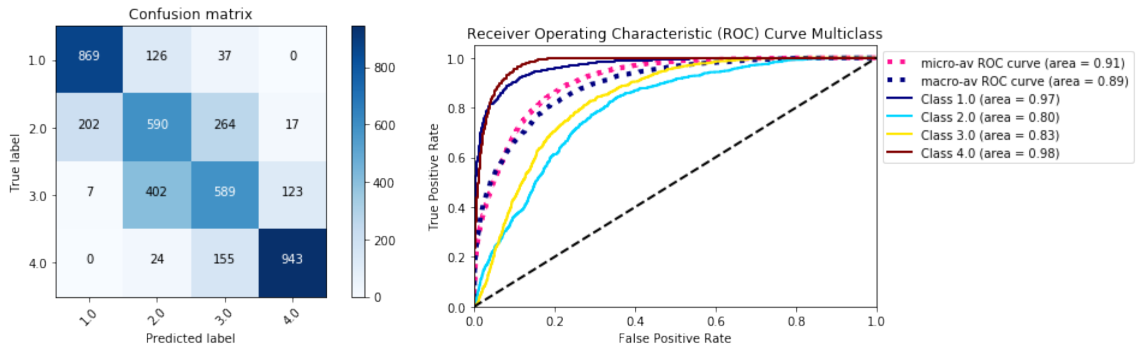


Figure 11: Predicting WS quartile for players of the 2010s using model fit to 1990s player data. (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for logistic regression classifier.

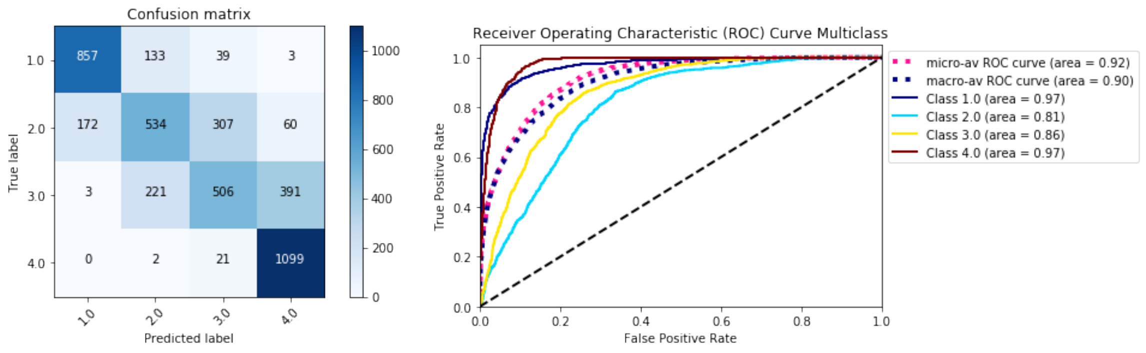


Figure 12: Predicting WS quartile for players of the 2010s using model fit to 2000s player data. (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for logistic regression classifier.

for feature spaces that better describe player performance.

References

[1] Jose Nathan Kutz. *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press, 2013.

[2] Basketball Reference. *NBA Win Shares*. Available at `https://www.basketball-reference.com/about/ws.html` (Copyright 2000-2020).

[3] Cosma Rohilla Shalizi. “Advanced Data Analysis from an Elementary Point of View”. In: 2012.

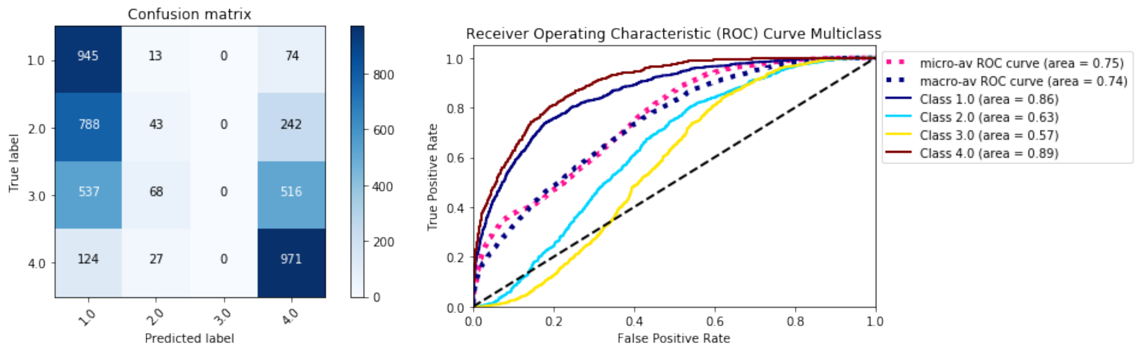


Figure 13: Predicting WS quartile for players of the 2010s using model fit to 1990s player data trained on first two principal components of player data. (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for logistic regression classifier.

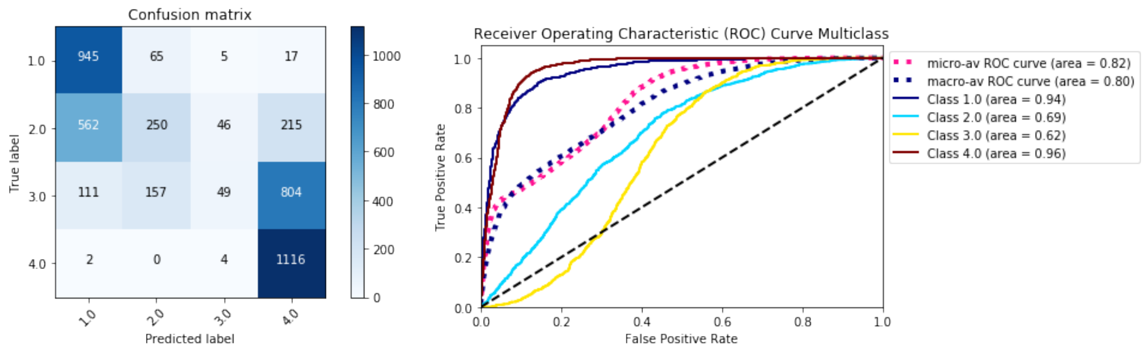


Figure 14: Predicting WS quartile for players of the 2010s using model fit to 1990s player data trained on first ten principal components of player data. (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for logistic regression classifier.

Appendix A `Python` Functions

This section contains some of the important `Python` functions used.

- `BeautifulSoup(html)`

  `Beautiful Soup` is a `Python` library for pulling data out of HTML and XML files. We pass the urls (`html`) of the webpages containing the datasets into the `BeautifulSoup` to create a soup object of the content of the webpage. We can then use various Beautiful Soup methods on the object.

- `sklearn.decomposition.PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None)`

  It uses the LAPACK implementation of the full SVD or a randomized truncated SVD depending on the shape of the input data and the number of components to extract. In our case, we extract all the components.

- `sklearn.model_selection.train_test_split(*arrays, **options)`

  Split arrays or matrices into random train and test subsets

- `sklearn.discriminant_analysis.LinearDiscriminantAnalysis(solver='svd', shrinkage=None, priors=None, n_components=None, store_covariance=False, tol=0.0001)`

Linear Discriminant Analysis. A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule. The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix. The fitted model can also be used to reduce the dimensionality of the input by projecting it to the most discriminative directions.

- `sklearn.naive_bayes.GaussianNB(priors=None, var_smoothing=1e-09)`

  This implements the Gaussian Naive Bayes algorithm for classification

- `sklearn.linear_model.LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)`

  Logistic Regression (aka logit, MaxEnt) classifier. In our project we have a multiclass case so the training algorithm uses the one-vs-rest (OvR) scheme since we set the '`multi_class`' option is set to '`ovr`'.

- `sklearn.metrics.roc_auc_score(y_true, y_score, average='macro', sample_weight=None, max_fpr=None, multi_class='raise', labels=None)`
  Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores. This implementation can be used with multiclass classification and that is exactly what we need.

Appendix B Supplemental Theoretical Background

B.1 Singular Value Decomposition:

As we learned from our textbook [1], SVD of a $m \times n$ matrix A is the factorization of the form:

$$A = \hat{U}\hat{\Sigma}V^* \tag{14}$$

where $\hat{U}$ is a $m \times n$ matrix with orthonormal columns, $\hat{\Sigma}$ is a $n \times n$ diagonal matrix with non negative entries and V is a $n \times n$ unitary matrix. This is called the *reduced SVD* of A. The following theorem guarantees the existence of such a factorization of any matrix A:

Theorem 1: *Every matrix $A \in C^{mxn}$ has a SVD. Furthermore, the singular values, that is, diagonal entries of Σ are uniquely determined.*

Another important theorem involving SVD is the following:

Theorem 2: *If the rank of A is r, then there are r nonzero singular values.*
Theorem 2 can help us determine the rank of a system by just considering the singular values. Many other important theorems regarding the SVD are given in [1]. SVD makes it possible for every matrix to be diagonal if the proper bases for the domain and the range are used. SVD provides a type of least-square fitting algorithm, allowing us to project matrix onto low-dimensional representations in a formal, algorithmic way.

B.2 Classifier Performance Metrics

In assessing the performance of the classification algorithms confusion matrices and Receiver Operating Characteristic Curves will be used.

A confusion matrix for a classification problems simply indicates the frequency of correct and erroneous class predictions in the test data. The counts indicate how often a class is predicted as itself or if it is frequently misclassified as another class. This is important information in assessing the performance and thus suitability of the classification model.

A Receiver Operating Characteristics (ROC) curve is a plot of False Positive Rate (FPR) versus True Positive Rate (TPR) for the classification of a given data class. These rates can be calculated using a macro-average strategy where the metric will be determined independently for each class before taking the average, or with a micro-average which instead collects all the contributions regardless of class before calculating the average FPR and TPR. In the case where a class imbalance may exist, the micro-average is the preferred method for calculating the ROC curve as it will not all the contributions of one class to disproportionately skew the curve trajectory. The ROC curve is a probability curve that indicates the relationship between the distributions of true positive and true of negative classifications. The area under the curve (AUC) is thus a measure of the degree of separability between a positive and negative classification.

Appendix C Additional Plots

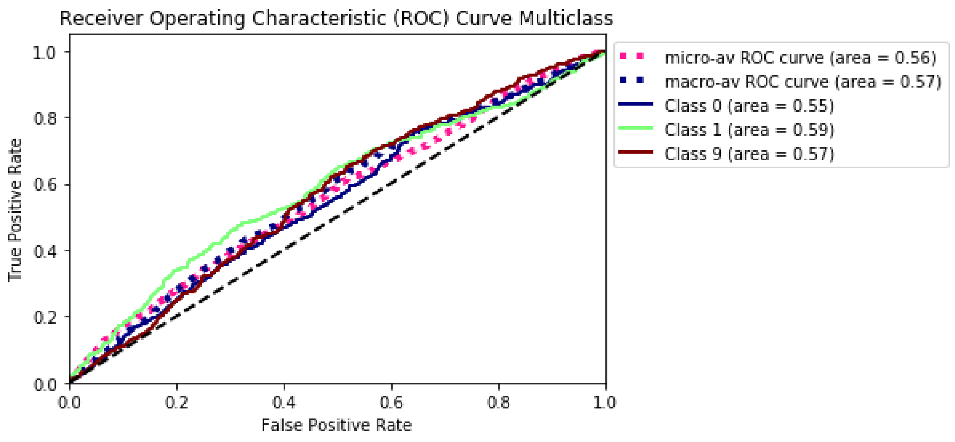


Figure 15: Plot of the ROC curve for the LDA classifier using decades as labels

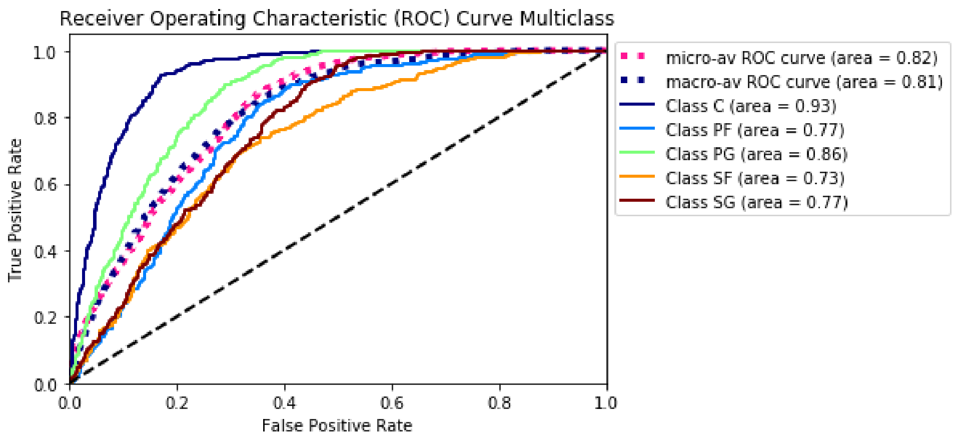


Figure 16: Plot of the ROC curve for the LDA classifier using position as labels

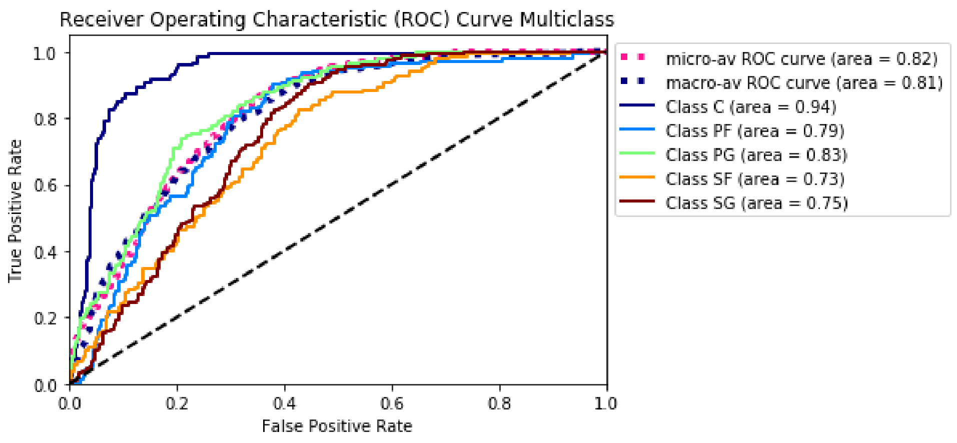


Figure 17: Plot of the ROC curve for the LDA classifier using position as labels for 2010-2019

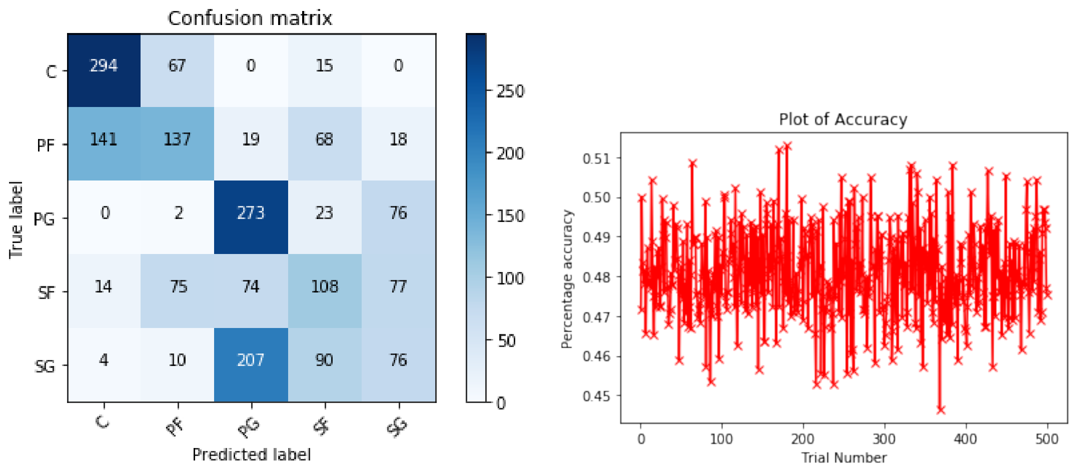


Figure 18: The left panel contains the confusion matrix for a particular simulation using Gaussian Naive Bayes using position as labels for all three decades. The left panel contains the plot of accuracy rate for each trial.

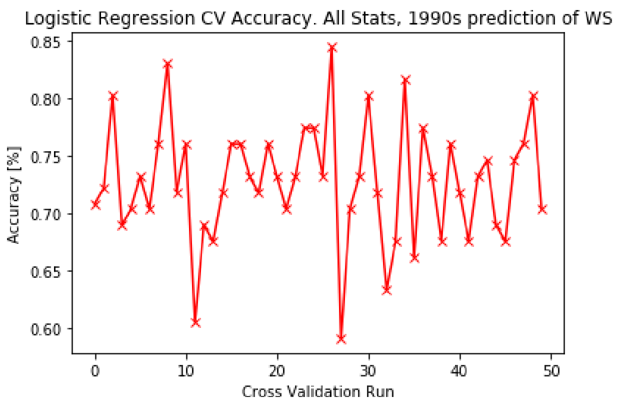


Figure 19: Cross validation accuracy for logistic regression classification of WS quartile for players of the 1990s.

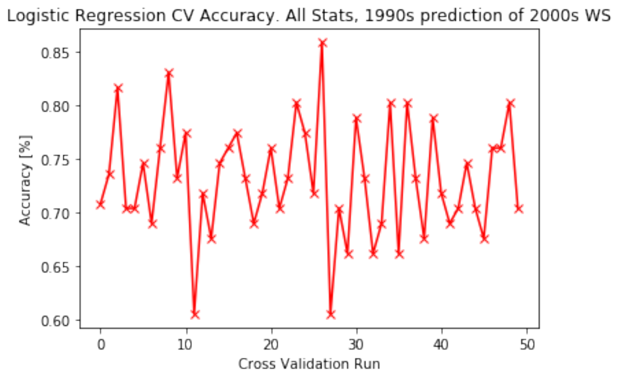


Figure 20: Cross validation accuracy for logistic regression classification of WS quartile for players of the 2000s.

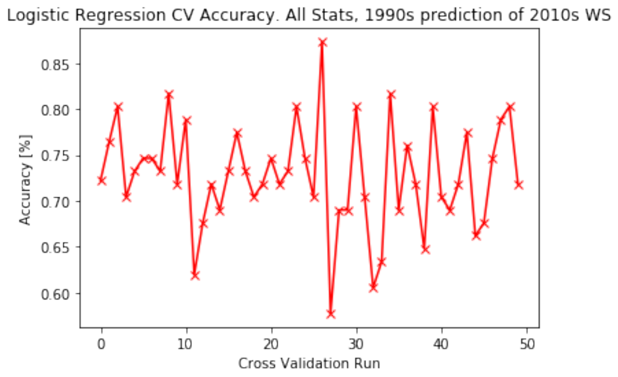


Figure 21: Cross validation accuracy for logistic regression classification of WS quartile for players of the 2010s.

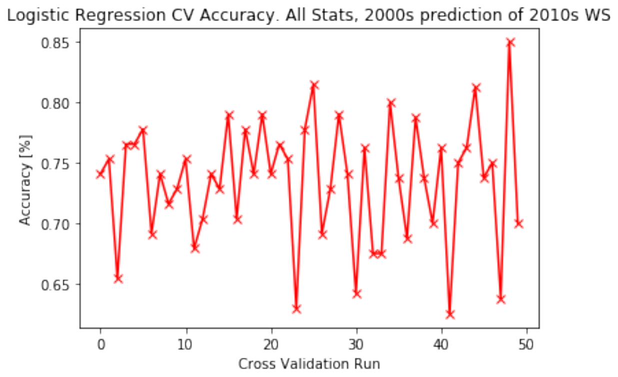


Figure 22: Cross validation accuracy for logistic regression classification of WS quartile for players of the 2010s based on training data of players from the 2000s.

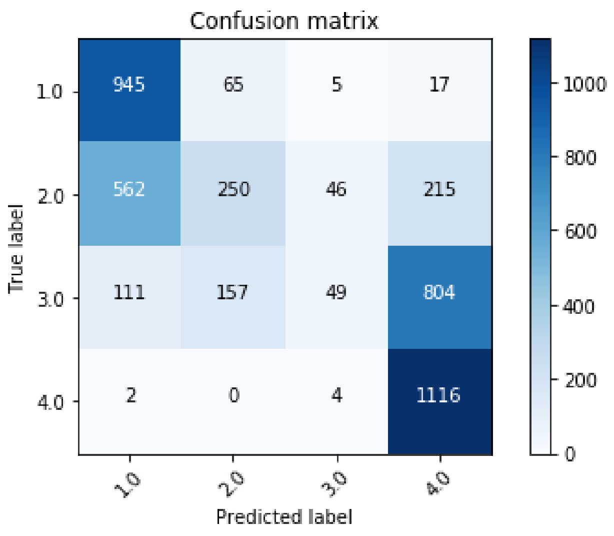


Figure 23: Cross validation accuracy for logistic regression classification of WS quartile for players of the 2010s based on the top 10 principal components of player data from the 1990s.

Appendix D Python Code

```
#!/usr/bin/env python
# coding: utf-8

# In[4]:



from urllib.request import urlopen
from bs4 import BeautifulSoup
import pandas as pd
import numpy as np
from Functions import cleanFunctions as cf
import math
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from scipy import stats as st

import random
import itertools
from scipy import interp
import matplotlib.cm as cm
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import *
from sklearn.preprocessing import label_binarize
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.multiclass import OneVsRestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import GaussianNB as GNB
from sklearn.cluster import KMeans


# In[5]:


'''Function that plots an ROC curve per class of a multiclass classifier case
by using the One Vs Rest technique.  This function was adapted from code
at: http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html
demonstrating the multiclass implementation of ROC curves for classifier
evaluation.

Inputs:
    X_train: dataframe object containing the data to train on.
    y_train: dataframe object containing the label binarized class outputs of
            the training set.
    X_test: dataframe object containing the data for testing the model
    y_test: dataframe object containing the label binarized class outputs of
            the test set.
    model: classifier object delineating the model to fit
    classes: list containing the classes of the data to be used in plotting

Outputs:
    ROC curve plot containing curves for each class, as well as the micro and
    macro average ROC curves.
'''
def multiclassROC(X_train, y_train, X_test, y_test, model, classes):
    classifier = OneVsRestClassifier(model)
    y_score = classifier.fit(X_train, y_train).predict_proba(X_test)
    y_train = label_binarize(y_train, classes=np.unique(y_train))
    y_test = label_binarize(y_test, classes=np.unique(y_test))
    n_classes = y_train.shape[1]
    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    # Compute macro-average ROC curve and ROC area

    # First aggregate all false positive rates
    all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

    # Then interpolate all ROC curves at this points
    mean_tpr = np.zeros_like(all_fpr)
    for i in range(n_classes):
        mean_tpr += interp(all_fpr, fpr[i], tpr[i])

    # Finally average it and compute AUC
    mean_tpr /= n_classes

    fpr["macro"] = all_fpr
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

    # Plot all ROC curves
    plt.figure()
    lw=2
    plt.plot(fpr["micro"], tpr["micro"],
             label='micro-av ROC curve (area = {0:0.2f})'
                   ''.format(roc_auc["micro"]),
             color='deeppink', linestyle=':', linewidth=4)

    plt.plot(fpr["macro"], tpr["macro"],
             label='macro-av ROC curve (area = {0:0.2f})'
                   ''.format(roc_auc["macro"]),
             color='navy', linestyle=':', linewidth=4)

    start = 0.0
    stop = 1.0
    number_of_lines= n_classes
    cm_subsection = np.linspace(start, stop, number_of_lines)
    colors = [ cm.jet(x) for x in cm_subsection ]

    for i, color in zip(range(n_classes), colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=lw,
                 label='Class {0} (area = {1:0.2f})'
                 ''.format(classes[i], roc_auc[i]))

    plt.plot([0, 1], [0, 1], 'k--', lw=lw)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve Multiclass')
    plt.legend(bbox_to_anchor=(1.00, 1.00))
    plt.show()
```

```python
"""
This function prints and plots the confusion matrix.
Normalization can be applied by setting `normalize=True`.
Adapted from code at:
    http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
"""
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.figure()
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    bottom, top = plt.ylim()
    plt.ylim(bottom + 0.5, top - 0.5)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()


# In[6]:


# Grabbing per game data for players in the 90s
year = np.arange(1990, 2020, 1)
url1 = "https://www.basketball-reference.com/leagues/NBA_{}_per_game.html"
url2 = "https://www.basketball-reference.com/leagues/NBA_{}_per_minute.html"
url3 = "https://www.basketball-reference.com/leagues/NBA_{}_per_poss.html"
url4 = "https://www.basketball-reference.com/leagues/NBA_{}_advanced.html"
urls = np.array([url1, url2, url3, url4])
stats_year = pd.DataFrame()
stats = pd.DataFrame()
count = 0
for y in year:
    for u in urls:
        print("Progress {:2.1%}".format(count / (len(year)*len(urls))), end="\r")#print progress of paint can position acquisition
        # URL page we will scraping (see image above)
        url = u.format(y)

        # this is the HTML from the given URL
        html = urlopen(url)
        soup = BeautifulSoup(html)

        # use findALL() to get the column headers
        soup.findAll('tr', limit=2)
        # use getText()to extract the text we need into a list
        headers = [th.getText() for th in soup.findAll('tr', limit=2)[0].findAll('th')]
        # exclude the first column as we will not need the ranking order from Basketball Reference for the analysis
        headers = headers[1:]
        headers

        #when concatenating data from each BR page, some duplicate columns are kept, and some columns need
        #to have additional info attached to their labels in order to know that they refer to per 36 or per poss
        #statistics and not per game statistics. The following conditional fix these flaws in the header labels
        if "per_game" in u:
            mystring = "_pg"
            dup = ['Player','Pos','Age','Tm','Year','G','GS']
            for i in range(0, len(headers)):
                if headers[i] not in dup:
                    headers[i] = headers[i] + mystring
        if "per_minute" in u:
            mystring = "_pm"
            dup = ['Player','Pos','Age','Tm','Year','G']
            for i in range(0, len(headers)):
                if headers[i] not in dup:
                    headers[i] = headers[i] + mystring
        if "per_poss" in u:
            mystring = "_pp"
            dup = ['Player','Pos','Age','Tm','Year','G']
            for i in range(0, len(headers)):
                if headers[i] not in dup:
                    headers[i] = headers[i] + mystring

        # avoid the first header row
        rows = soup.findAll('tr')[1:]
        player_stats = [[td.getText() for td in rows[i].findAll('td')]
                    for i in range(len(rows))]


        stats_new = pd.DataFrame(player_stats, columns = headers)
        if "per_game" not in u:
            stats_new.drop(['Pos', 'Age', 'Tm', 'G'], axis=1)
        [p, s] = stats_new.shape

        stats_year = pd.concat([stats_year, stats_new], axis=1, sort=False)
        count = count + 1
    stats_year.insert (4, "Year", (y*np.ones((p, 1))).astype(int))
    stats = stats.append(stats_year, ignore_index=True)
    stats_year = pd.DataFrame()


stats = stats.drop(['_pp', '\xa0', '\xa0', 'MP_pp', 'MP_pm', 'GS_pp', 'GS_pm'], axis=1)
stats = stats.loc[:,~stats.columns.duplicated()]
stats = stats[~stats['Player'].isnull()]#drop instances where player name is null
stats.head(10)


# In[7]:


stats.loc[stats['Player'] == 'Kobe Bryant']#example player data

#95 stats and 16489 players


# In[8]:


#data preprocessing
stats.insert(5, 'Decade', stats.loc[:,'Year']%100 // 10)

#for columns known to be numeric, cast to numeric quantities
for c in stats.loc[:, 'Year':].columns.tolist():
    stats.loc[:, c] = pd.to_numeric(stats.loc[:, c])

stats_filt = stats[stats["G"] >= 20]#filter out players who played less than 20 games

#get list of categories in the filtered data set that have null entries after casting to numeric
nulls = [i for i, e in enumerate(stats_filt.isnull().sum().tolist()) if e != 0]
cols = stats_filt.columns.tolist()
null_cats = [cols[i] for i in nulls]

#entries of the filtered dataframe may be null after casting to numeric. check what these specific
#categories describe, in this case the null cats are shooting percentages which likely
#means that a given player did not attempt shots of that type and therfore had a non-numeric (i.e empty)
#string value for their shooting percentage. These instances can be safely imputed as 0.
stats_filt = stats_filt.fillna(0)

#standardize position categories, in the case of multi position players... assign them as their primary position
pos_col = stats_filt.loc[:, 'Pos'].tolist()
for i in range(0, len(pos_col)):
    pos = pos_col[i].split('-')
    pos_col[i] = pos[0]

stats_filt.loc[:, 'Pos'] = pos_col

print(stats.shape)#shape of original data set
print(stats_filt.shape)#shape after filtering on games played


# In[9]:


#standardize data before PCA
X = StandardScaler().fit_transform(stats_filt.loc[:, 'MP_pg':])
#perform PCA
```

```python
pca = PCA()
principalComponents = pca.fit_transform(X)
pDf_1 = pd.DataFrame(data = principalComponents[:,:2], columns = ['PC1', 'PC2'])
pDf_1['Decade']  = stats_filt['Decade'].tolist()
pDf_1['Pos'] = stats_filt['Pos'].tolist()

percent_explained_var = 100*pca.explained_variance_/sum(pca.explained_variance_)
cut_off = 10
cumulative = 0
cumulative_exp = [0]
required_modes = 0
first_time = True
for i in range(0, len(percent_explained_var)):
    cumulative = cumulative + percent_explained_var[i]
    cumulative_exp.append(cumulative)
    if cumulative > (100 - cut_off) and first_time:
        required_modes = i
        first_time = False


f, ax1 = plt.subplots(1, 1, figsize=(20, 8))
ax1.plot(np.arange(0,10,1), cumulative_exp[required_modes]*np.ones(10), 'k--')
ax1.plot(np.arange(1,len(pca.explained_variance_) + 1,1).astype(int), percent_explained_var, 'bo', fillStyle='none')
ax1.plot(np.arange(0,len(pca.explained_variance_) + 1,1).astype(int), cumulative_exp, 'r--')
ax1.axvline(x=required_modes, ymin=0, ymax=cumulative_exp[required_modes-1]/100., c='k', ls = '--')
ax1.set_xticks(np.arange(0,len(pca.explained_variance_) + 1,1))
ax1.set_title('Test 1 Explained Variance by Principal Component')
ax1.set_ylabel('Percent Explained Variance [%]')
ax1.set_xlabel('Principal Component')
ax1.set_xlim([0.5, 55])


# In[7]:


#Top Mode PCA Plots
#first plot colored by decade
fig, ax = plt.subplots(figsize=(12, 12))
ax.scatter(pDf_1.loc[:,"PC1"], pDf_1.loc[:,"PC2"], alpha=0.5, c=pDf_1.loc[:,'Decade'])
ax.set_title("Top 2 Principal Components, Colored by Decade")

#second plot of principal components colored by position
fig, ax = plt.subplots(figsize=(12, 12))
colors = {'PG':'red', 'SG':'blue', 'SF':'green', 'PF':'black', 'C':'yellow'}
labels = list(colors.keys())
for g in labels:
    ix = pDf_1.index[pDf_1['Pos'] == g]
    ax.scatter(pDf_1.loc[ix, "PC1"], pDf_1.loc[ix, "PC2"], alpha=0.25, c = colors[g], label = g)
ax.set_title("Top 2 Principal Components, Colored by Position")
ax.legend()
#ax.scatter(pDf_1.loc[:,"PC1"], pDf_1.loc[:,"PC2"], alpha=0.5, label=labels ,c=pDf_1.loc[:,'Pos'].apply(lambda x: colors[x]))
ax.legend(labels)



# In[21]:


#Splitting the data matrix into three decades
d1 = 3885;  # number of samples from 1990-1999
d2 = 4385; # number of samples from 2000-2009
d3 = 4806; # number of samples 2010-2019
stats_d1 = stats_filt[0:d1]
stats_d2 = stats_filt[d1:d1+d2]
stats_d3 = stats_filt[d1+d2:d1+d2+d3]



# In[9]:


X_1 = StandardScaler().fit_transform(stats_d1.loc[:, 'MP_pg':])
#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X_1)
pDf_11 = pd.DataFrame(data = principalComponents[:,:2], columns = ['PC1', 'PC2'])
pDf_11['Decade']  = stats_d1['Decade'].tolist()
pDf_11['Pos'] = stats_d1['Pos'].tolist()
```

```
#plot of principal components colored by position
fig, ax = plt.subplots(figsize=(12, 12))
colors = {'PG':'red', 'SG':'blue', 'SF':'green', 'PF':'black', 'C':'yellow'}
labels = list(colors.keys())
for g in labels:
    ix = pDf_11.index[pDf_11['Pos'] == g]
    ax.scatter(pDf_11.loc[ix, "PC1"], pDf_11.loc[ix, "PC2"], alpha=0.25, c = colors[g], label = g)
ax.set_title("Top 2 Principal Components, Colored by Position (1990-1999)")
ax.legend()
ax.legend(labels)


# In[10]:


# 2000-2009

X_2 = StandardScaler().fit_transform(stats_d2.loc[:, 'MP_pg':])
#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X_2)
pDf_12 = pd.DataFrame(data = principalComponents[:,:2], columns = ['PC1', 'PC2'])
pDf_12['Decade']  = stats_d2['Decade'].tolist()
pDf_12['Pos'] = stats_d2['Pos'].tolist()


#plot of principal components colored by position
fig, ax = plt.subplots(figsize=(12, 12))
colors = {'PG':'red', 'SG':'blue', 'SF':'green', 'PF':'black', 'C':'yellow'}
labels = list(colors.keys())
for g in labels:
    ix = pDf_12.index[pDf_12['Pos'] == g]
    ax.scatter(pDf_12.loc[ix, "PC1"], pDf_12.loc[ix, "PC2"], alpha=0.25, c = colors[g], label = g)
ax.set_title("Top 2 Principal Components, Colored by Position (2000-2009)")
ax.legend()
ax.legend(labels)


# In[11]:


# 2010-2019

X_3 = StandardScaler().fit_transform(stats_d3.loc[:, 'MP_pg':])
#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X_3)
pDf_13 = pd.DataFrame(data = principalComponents[:,:2], columns = ['PC1', 'PC2'])
pDf_13['Decade']  = stats_d3['Decade'].tolist()
pDf_13['Pos'] = stats_d3['Pos'].tolist()


#plot of principal components colored by position
fig, ax = plt.subplots(figsize=(12, 12))
colors = {'PG':'red', 'SG':'blue', 'SF':'green', 'PF':'black', 'C':'yellow'}
labels = list(colors.keys())
for g in labels:
    ix = pDf_13.index[pDf_13['Pos'] == g]
    ax.scatter(pDf_13.loc[ix, "PC1"], pDf_13.loc[ix, "PC2"], alpha=0.25, c = colors[g], label = g)
ax.set_title("Top 2 Principal Components, Colored by Position (2010-2019)")
ax.legend()
ax.legend(labels)


# In[47]:


# Linear discrimination analysis by decade

# standardize data before PCA
X_d= StandardScaler().fit_transform(stats_filt.loc[:, 'MP_pg':])
#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X_d)

#Using first two components to do LDA


X_dp= principalComponents[:,:2]
labels = stats_filt.loc[:,'Decade']
rs = random.seed(15)#set random seed
```

```
classes, y = np.unique(labels, return_inverse=True)#get unique class names and translate to integer values

accuracy = [];

for i in range(1,501):
    #test_size: what proportion of original data is used for test set
    Xtrain, Xtest, ytrain, ytest = train_test_split( X_dp, y, test_size=1/7.0, random_state=rs)
    clf = LDA(n_components = 1,tol=0.01)
    #function returns multiclass ROC plot with AUC scores per class
    clf.fit(Xtrain, ytrain)
    preds_logr = clf.predict(Xtest)#apply the model
    aScore = accuracy_score(ytest, preds_logr)#get accuracy score
    accuracy.append(aScore)
multiclassROC(Xtrain, ytrain, Xtest, ytest, clf, classes)
CM = confusion_matrix(ytest, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
avg_accuracy = sum(accuracy)/501
print('Average accuracy:', avg_accuracy*100)
x = np.linspace(1,500, 500)
plt.plot(x, accuracy, 'rx-')
plt.xlabel('Trial Number')
plt.ylabel('Percentage accuracy')
plt.title('Plot of Accuracy')


# In[48]:


# Linear discrimination analysis by position


# standardize data before PCA
X_p= StandardScaler().fit_transform(stats_filt.loc[:, 'MP_pg':])
#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X_p)

#Using first two components to do LDA


X_pca= principalComponents[:,:2]
labels = stats_filt.loc[:,'Pos']
rs = random.seed(15)#set random seed

# creating labels
classes, y = np.unique(labels, return_inverse=True)#get unique class names and translate to integer values

accuracy = [];
for i in range(1,501):
    #test_size: what proportion of original data is used for test set
    Xtrain, Xtest, ytrain, ytest = train_test_split( X_pca, y, test_size=1/7.0, random_state=rs)
    clf = LDA(n_components = 1,tol=0.01)
    clf.fit(Xtrain, ytrain)
    preds_logr = clf.predict(Xtest)#apply the model
    aScore = accuracy_score(ytest, preds_logr)#get accuracy score
    accuracy.append(aScore)

multiclassROC(Xtrain, ytrain, Xtest, ytest, clf, classes)
CM = confusion_matrix(ytest, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
avg_accuracy = sum(accuracy)/501
print('Average accuracy:', avg_accuracy*100)
x = np.linspace(1,500, 500)
plt.plot(x, accuracy, 'rx-')
plt.xlabel('Trial Number')
plt.ylabel('Percentage accuracy')
plt.title('Plot of Accuracy')


# In[46]:


# Using Naive Bayes

# standardize data before PCA
X_d= StandardScaler().fit_transform(stats_filt.loc[:, 'MP_pg':])
#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X_d)

#Using first two components to do LDA
```

```
X_dp= principalComponents[:,:2]
labels = stats_filt.loc[:,'Pos']
rs = random.seed(15)#set random seed
classes, y = np.unique(labels, return_inverse=True)#get unique class names and translate to integer values

accuracy = [];

for i in range(1,501):
    #test_size: what proportion of original data is used for test set
    X_train, X_test, y_train, y_test = train_test_split( X_dp, y, test_size=1/7.0, random_state=rs)
    gnb = GNB()
    #function returns multiclass ROC plot with AUC scores per class
    preds_logr = gnb.fit(X_train, y_train).predict(X_test)#apply the model
    aScore = accuracy_score(y_test, preds_logr)#get accuracy score
    accuracy.append(aScore)

CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
avg_accuracy = sum(accuracy)/501
print('Average accuracy:', avg_accuracy*100)
x = np.linspace(1,500, 500)
plt.plot(x, accuracy, 'rx-')
plt.xlabel('Trial Number')
plt.ylabel('Percentage accuracy')
plt.title('Plot of Accuracy')


# In[22]:



# standardize data before PCA
X_p3= StandardScaler().fit_transform(stats_d3.loc[:, 'MP_pg':])
#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X_p3)

#Using first two components to do LDA


X_pca3= principalComponents[:,:2]
labels = stats_d3.loc[:,'Pos']
rs = random.seed(15)#set random seed

# creating labels
classes, y = np.unique(labels, return_inverse=True)#get unique class names and translate to integer values

accuracy = [];
for i in range(1,501):
    #test_size: what proportion of original data is used for test set
    Xtrain, Xtest, ytrain, ytest = train_test_split( X_pca3, y, test_size=1/7.0, random_state=rs)
    clf = LDA(n_components = 1,tol=0.01)
    clf.fit(Xtrain, ytrain)
    preds_logr = clf.predict(Xtest)#apply the model
    aScore = accuracy_score(ytest, preds_logr)#get accuracy score
    accuracy.append(aScore)


multiclassROC(Xtrain, ytrain, Xtest, ytest, clf, classes)
CM = confusion_matrix(ytest, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
avg_accuracy = sum(accuracy)/501
print('Average accuracy:', avg_accuracy*100)
x = np.linspace(1,500, 500)
plt.plot(x, accuracy, 'rx-')
plt.xlabel('Trial Number')
plt.ylabel('Percentage accuracy')
plt.title('Plot of Accuracy')


# In[ ]:

#!/usr/bin/env python
# coding: utf-8

# In[1]:


from urllib.request import urlopen
```

```python
from bs4 import BeautifulSoup
import pandas as pd
import numpy as np
from Functions import cleanFunctions as cf
import math
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from scipy import stats

import random
import itertools
from scipy import interp
import matplotlib.cm as cm
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import *
from sklearn.preprocessing import label_binarize
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.multiclass import OneVsRestClassifier
from sklearn.model_selection import cross_validate


# In[2]:


'''Function that plots an ROC curve per class of a multiclass classifier case
by using the One Vs Rest technique.  This function was adapted from code
at: http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html
demonstrating the multiclass implementation of ROC curves for classifier
evaluation.

Inputs:
    X_train: dataframe object containing the data to train on.
    y_train: dataframe object containing the label binarized class outputs of
            the training set.
    X_test: dataframe object containing the data for testing the model
    y_test: dataframe object containing the label binarized class outputs of
            the test set.
    model: classifier object delineating the model to fit
    classes: list containing the classes of the data to be used in plotting

Outputs:
    ROC curve plot containing curves for each class, as well as the micro and
    macro average ROC curves.
'''
def multiclassROC(X_train, y_train, X_test, y_test, model, classes):
    classifier = OneVsRestClassifier(model)
    y_score = classifier.fit(X_train, y_train).predict_proba(X_test)
    y_train = label_binarize(y_train, classes=np.unique(y_train))
    y_test = label_binarize(y_test, classes=np.unique(y_test))
    n_classes = y_train.shape[1]
    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    # Compute macro-average ROC curve and ROC area

    # First aggregate all false positive rates
    all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

    # Then interpolate all ROC curves at this points
    mean_tpr = np.zeros_like(all_fpr)
    for i in range(n_classes):
        mean_tpr += interp(all_fpr, fpr[i], tpr[i])

    # Finally average it and compute AUC
    mean_tpr /= n_classes

    fpr["macro"] = all_fpr
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])
    # Plot all ROC curves
    plt.figure()
    lw=2
    plt.plot(fpr["micro"], tpr["micro"],
             label='micro-av ROC curve (area = {0:0.2f})'
                   ''.format(roc_auc["micro"]),
             color='deeppink', linestyle=':', linewidth=4)

    plt.plot(fpr["macro"], tpr["macro"],
             label='macro-av ROC curve (area = {0:0.2f})'
                   ''.format(roc_auc["macro"]),
             color='navy', linestyle=':', linewidth=4)

    start = 0.0
    stop = 1.0
    number_of_lines= n_classes
    cm_subsection = np.linspace(start, stop, number_of_lines)
    colors = [ cm.jet(x) for x in cm_subsection ]

    for i, color in zip(range(n_classes), colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=lw,
                 label='Class {0} (area = {1:0.2f})'
                 ''.format(classes[i], roc_auc[i]))

    plt.plot([0, 1], [0, 1], 'k--', lw=lw)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve Multiclass')
    plt.legend(bbox_to_anchor=(1.00, 1.00))
    plt.show()

"""
This function prints and plots the confusion matrix.
Normalization can be applied by setting `normalize=True`.
Adapted from code at:
    http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
"""
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.figure()
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    #bottom, top = plt.ylim()
    #plt.ylim(bottom + 0.5, top - 0.5)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()


# In[3]:


# Grabbing per game data for players in the 90s
year = np.arange(1990, 2020, 1)
url1 = "https://www.basketball-reference.com/leagues/NBA_{}_per_game.html"
url2 = "https://www.basketball-reference.com/leagues/NBA_{}_per_minute.html"
url3 = "https://www.basketball-reference.com/leagues/NBA_{}_per_poss.html"
url4 = "https://www.basketball-reference.com/leagues/NBA_{}_advanced.html"
urls = np.array([url1, url2, url3, url4])
stats_year = pd.DataFrame()
```

```
stats_raw = pd.DataFrame()
count = 0
for y in year:
    for u in urls:
        print("Progress {:2.1%}".format(count / (len(year)*len(urls))), end="\r")#print progress of paint can position acquisition
        # URL page we will scraping (see image above)
        url = u.format(y)

        # this is the HTML from the given URL
        html = urlopen(url)
        soup = BeautifulSoup(html)

        # use findALL() to get the column headers
        soup.findAll('tr', limit=2)
        # use getText()to extract the text we need into a list
        headers = [th.getText() for th in soup.findAll('tr', limit=2)[0].findAll('th')]
        # exclude the first column as we will not need the ranking order from Basketball Reference for the analysis
        headers = headers[1:]
        headers

        #when concatenating data from each BR page, some duplicate columns are kept, and some columns need
        #to have additional info attached to their labels in order to know that they refer to per 36 or per poss
        #statistics and not per game statistics. The following conditional fix these flaws in the header labels
        if "per_game" in u:
            mystring = "_pg"
            dup = ['Player','Pos','Age','Tm','Year','G','GS']
            for i in range(0, len(headers)):
                if headers[i] not in dup:
                    headers[i] = headers[i] + mystring
        if "per_minute" in u:
            mystring = "_pm"
            dup = ['Player','Pos','Age','Tm','Year','G']
            for i in range(0, len(headers)):
                if headers[i] not in dup:
                    headers[i] = headers[i] + mystring
        if "per_poss" in u:
            mystring = "_pp"
            dup = ['Player','Pos','Age','Tm','Year','G']
            for i in range(0, len(headers)):
                if headers[i] not in dup:
                    headers[i] = headers[i] + mystring

        # avoid the first header row
        rows = soup.findAll('tr')[1:]
        player_stats = [[td.getText() for td in rows[i].findAll('td')]
                    for i in range(len(rows))]


        stats_new = pd.DataFrame(player_stats, columns = headers)
        if "per_game" not in u:
            stats_new.drop(['Pos', 'Age', 'Tm', 'G'], axis=1, inplace=True)
        [p, s] = stats_new.shape

        stats_year = pd.concat([stats_year, stats_new], axis=1, sort=False)
        count = count + 1
    stats_year.insert (4, "Year", (y*np.ones((p, 1))).astype(int))
    stats_raw = stats_raw.append(stats_year, ignore_index=True)
    stats_year = pd.DataFrame()


stats_raw = stats_raw.drop(['_pp', '\xa0', '\xa0', 'MP_pp', 'MP_pm', 'GS_pp', 'GS_pm'], axis=1)
stats_raw = stats_raw.loc[:,~stats_raw.columns.duplicated()]
stats_raw = stats_raw[~stats_raw['Player'].isnull()]#drop instances where player name is null
stats_raw.head(10)


# In[4]:


#data preprocessing
stats_raw.insert(5, 'Decade', stats_raw.loc[:,'Year']%100 // 10)
#for columns known to be numeric, cast to numeric quantities
for c in stats_raw.loc[:, 'Year':].columns.tolist():
    stats_raw.loc[:, c] = pd.to_numeric(stats_raw.loc[:, c])

stats_filt = stats_raw[stats_raw["G"] >= 20]#filter out players who played less than 20 games

#get list of categories in the filtered data set that have null entries after casting to numeric
nulls = [i for i, e in enumerate(stats_filt.isnull().sum().tolist()) if e != 0]
cols = stats_filt.columns.tolist()
null_cats = [cols[i] for i in nulls]
```

```
#entries of the filtered dataframe may be null after casting to numeric. check what these specific
#categories describe, in this case the null cats are shooting percentages which likely
#means that a given player did not attempt shots of that type and therfore had a non-numeric (i.e empty)
#string value for their shooting percentage. These instances can be safely imputed as 0.
stats_filt = stats_filt.fillna(0)

#standardize position categories, in the case of multi position players... assign them as their primary position
pos_col = stats_filt.loc[:, 'Pos'].tolist()
for i in range(0, len(pos_col)):
    pos = pos_col[i].split('-')
    pos_col[i] = pos[0]

stats_filt.loc[:, 'Pos'] = pos_col
#move MP to front of data frame (it is a total statistic, not one that can be averaged)
cols = list(stats_filt)
cols.insert(7, cols.pop(cols.index('MP')))#pop the MP column out of its current spot, and reinsert at 7 spot
stats_filt = stats_filt[cols]# use cols list to reorder dataframe columns

#Also, if a player moved team, lets take the averages of the stats from each team, and change team name
#to a concatenated string of the teams played on.
years = stats_filt.loc[:, 'Year'].unique().tolist()#list of years in data set
bp_list = []
count = 0
for y in years:
    print("Progress {:2.1%}".format(count / (len(years))), end="\r")#print progress of paint can position acquisition
    temp = stats_filt[(stats_filt["Year"] == y) & (stats_filt["Tm"] == 'TOT')]#grab instances where a player has a "total" team played for, meaning
    for p in temp['Player']:#for each player that has played for multiple teams
        pdata = stats_filt[(stats_filt["Year"] == y) & (stats_filt["Player"] == p)]#find the rest of their data from the individual teams for that
        teams = pdata['Tm'].values.tolist()#get list of teams played for
        if 'TOT' in teams: teams.remove('TOT')#remove "total" team name
        newTm = '/'.join(teams)#join all teams played for seperated by /

        indexNames = pdata[(pdata['Player'] == p) & (pdata['Tm'] != 'TOT')].index# Get indexes for player on multiple other teams
        stats_filt.drop(indexNames, axis=0, inplace=True)# Delete these row indexes from dataFrame
        pidx = stats_filt[stats_filt['Player'] == p].index
        stats_filt.loc[pidx, 'Tm'] = newTm#set Tm name to the concatenated string.
    count = count + 1


print("\n")
print(stats_raw.shape)#shape of original data set
print(stats_filt.shape)#shape after filtering on games played


# In[5]:


#standardize data before PCA
X = StandardScaler().fit_transform(stats_filt.loc[:, 'MP_pg':'BPM'])

#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X)
pDf_1 = pd.DataFrame(data = principalComponents[:,:2], columns = ['PC1', 'PC2'])
pDf_1['Decade']  = stats_filt['Decade'].tolist()
pDf_1['Pos'] = stats_filt['Pos'].tolist()

percent_explained_var = 100*pca.explained_variance_/sum(pca.explained_variance_)
cut_off = 10
cumulative = 0
cumulative_exp = [0]
required_modes = 0
first_time = True
for i in range(0, len(percent_explained_var)):
    cumulative = cumulative + percent_explained_var[i]
    cumulative_exp.append(cumulative)
    if cumulative > (100 - cut_off) and first_time:
        required_modes = i
        first_time = False


f, ax1 = plt.subplots(1, 1, figsize=(20, 8))
ax1.plot(np.arange(0,required_modes+1,1), cumulative_exp[required_modes]*np.ones(required_modes+1), 'k--')
ax1.plot(np.arange(1,len(pca.explained_variance_) + 1,1).astype(int), percent_explained_var, 'bo', fillStyle='none')
ax1.plot(np.arange(0,len(pca.explained_variance_) + 1,1).astype(int), cumulative_exp, 'r--')
ax1.axvline(x=required_modes, ymin=0, ymax=cumulative_exp[required_modes-1]/100., c='k', ls = '--')
ax1.set_xticks(np.arange(0,len(pca.explained_variance_) + 1,1))
ax1.set_title('Test 1 Explained Variance by Principal Component')
ax1.set_ylabel('Percent Explained Variance [%]')
ax1.set_xlabel('Principal Component')
```

```
ax1.set_xlim([0.5, 55])


# In[6]:


#Top Mode PCA Plots
#first plot colored by decade
fig, ax = plt.subplots(figsize=(12, 12))
ax.scatter(pDf_1.loc[:,"PC1"], pDf_1.loc[:,"PC2"], alpha=0.5, c=pDf_1.loc[:,'Decade'])
ax.set_title("Top 2 Principal Components, Colored by Decade")

#second plot of principal components colored by position
fig, ax = plt.subplots(figsize=(12, 12))
colors = {'PG':'red', 'SG':'blue', 'SF':'green', 'PF':'black', 'C':'yellow'}
labels = list(colors.keys())
for g in labels:
    ix = pDf_1.index[pDf_1['Pos'] == g]
    ax.scatter(pDf_1.loc[ix, "PC1"], pDf_1.loc[ix, "PC2"], alpha=0.25, c = colors[g], label = g)
ax.set_title("Top 2 Principal Components, Colored by Position")
ax.legend()
ax.legend(labels)


# In[7]:


'''Our plan is to split the data set into each decade and first apply PCA to reduce the dimensions of the data
of each decade. Then we intend to apply ICA to search for statistically independent signals within the data.
Our hope is to compare the ICA extracted features between eras to demonstrate the dynamics of the sport over the
last 30 years. If successful, we are also interested in using these feature dynamics to predict the trajectory of
the sport in the coming decade, and potentially predict new player success.
'''
#We are at some point going to need a metric for predicting success... I propose "years in top 25% of league BPM"
#Lets look at BPM distributions of each decade...
decades = stats_filt.loc[:, 'Decade'].unique().tolist()
bp_list = []
for d in decades:
    temp = stats_filt[stats_filt["Decade"] == d]
    bp_list.append(temp.loc[:, 'BPM'])

#filter out players who played less than 20 games
fig, ax = plt.subplots()
ax.set_title('BPM for Last 3 Decades')
ax.boxplot(bp_list)
ax.set_xticklabels(decades, fontsize=8)
plt.show()

#Let's calculate years in top 25% BPM for each player...
#start by calculating percentile of each player's BPM for each year
years = stats_filt.loc[:, 'Year'].unique().tolist()#list of years in data set
percentile_cutoff = 75.0
bpm_list = []
stats_filt['BPM_annual_percentile'] = np.nan #initialize empty column for holding annual BPM percentile
stats_filt['BPM_class'] = np.nan #initialize empty column for holding BPM class (is top 25% or not?)
count = 0
for y in years:
    print("Progress {:2.1%}".format(count / (len(years))), end="\r")#print progress
    temp = stats_filt[stats_filt["Year"] == y].loc[:, ['Player', 'BPM']]#grab out Player name and BPM data for each year
    for p in temp['Player']:
        playerBPM = temp.loc[temp[temp['Player'] == p].index, 'BPM']#get the player's BPM for the year
        percentile = stats.percentileofscore(temp['BPM'], playerBPM.iloc[0])#calculate what percentile the player's BPM is
        pidx = stats_filt[(stats_filt['Player'] == p) & (stats_filt["Year"] == y)].index#get index of player for the year in full dataset
        stats_filt.loc[pidx,'BPM_annual_percentile'] = percentile#fill in player's BPM percentile value for the year
        if percentile > percentile_cutoff:#check if the percentile is above the cutoff
            stats_filt.loc[pidx,'BPM_class'] = 1#if the player has a BPM percentile higher than cutoff, class 1
        else:
            stats_filt.loc[pidx,'BPM_class'] = 0#otherwise, class 0
    count = count + 1

stats_filt.BPM_class.value_counts().plot(kind = 'barh')#counts of BPM classes
plt.show()

#determine years of top 25% BPM performance
stats_filt['BPM_years'] = np.nan
players = stats_filt['Player'].unique().tolist()
count = 0
for p in players:
    print("Progress {:2.1%}".format(count / (len(players))), end="\r")#print progress
    pidx = stats_filt[stats_filt['Player'] == p].index
    p_BPM_count = stats_filt.loc[pidx,'BPM_class'].sum()
    stats_filt.loc[pidx, 'BPM_years'] = p_BPM_count
    count = count + 1

fig, ax = plt.subplots()
ax.bar(stats_filt.BPM_years.unique().tolist(), stats_filt.BPM_years.value_counts())
ax.set_title("Number of Players with 75th Percentile BPM Years")
plt.show()


# In[8]:


topBPM = stats_filt[stats_filt['BPM_years'] > 15]
s = topBPM['Player'].unique().tolist()
listToStr = ', '.join([str(elem) for elem in s])
print("Players with > 15 years of top 75% BPM: " + listToStr)

topBPM = stats_filt[stats_filt['BPM_years'] == stats_filt['BPM_years'].max()]
s = topBPM['Player'].unique().tolist()
listToStr = ''.join([str(elem) for elem in s])
print("\nPlayer with most years in top 75% of league BPM: " + listToStr + ' (' + str(stats_filt['BPM_years'].max())+' years)')



# In[9]:


#Instead of BPM.. let's calculate years in top 25% WS for each player...
#start by calculating percentile of each player's WS for each year
years = stats_filt.loc[:, 'Year'].unique().tolist()#list of years in data set
percentile_cutoff = 75.0
bpm_list = []
stats_filt['WS_annual_percentile'] = np.nan #initialize empty column for holding annual BPM percentile
stats_filt['WS_class'] = np.nan #initialize empty column for holding BPM class (is top 25% or not?)
count = 0
for y in years:
    print("Progress {:2.1%}".format(count / (len(years))), end="\r")#print progress
    temp = stats_filt[stats_filt["Year"] == y].loc[:, ['Player', 'WS']]#grab out Player name and WS data for each year
    for p in temp['Player']:
        playerWS = temp.loc[temp[temp['Player'] == p].index, 'WS']#get the player's WS for the year
        percentile = stats.percentileofscore(temp['WS'], playerWS.iloc[0])#calculate what percentile the player's WS is
        pidx = stats_filt[(stats_filt['Player'] == p) & (stats_filt["Year"] == y)].index#get index of player for the year in full dataset
        stats_filt.loc[pidx,'WS_annual_percentile'] = percentile#fill in player's WS percentile value for the year
        if percentile > 75.0:#check if the percentile is above the cutoff
            stats_filt.loc[pidx,'WS_class'] = 4#if the player has a WS percentile higher than cutoff, class 4
        elif percentile > 50.0:
            stats_filt.loc[pidx,'WS_class'] = 3#if the player has a WS percentile higher than cutoff, class 3
        elif percentile > 25.0:
            stats_filt.loc[pidx,'WS_class'] = 2#if the player has a WS percentile higher than cutoff, class 2
        else:
            stats_filt.loc[pidx,'WS_class'] = 1#otherwise, class 1
    count = count + 1

#determine years of top 25% BPM performance
stats_filt['WS_years'] = np.nan
players = stats_filt['Player'].unique().tolist()
count = 0
for p in players:
    print("Progress {:2.1%}".format(count / (len(players))), end="\r")#print progress
    pidx = stats_filt[stats_filt['Player'] == p].index
    p_BPM_count = stats_filt.loc[pidx,'WS_class'].sum() // 4
    stats_filt.loc[pidx, 'WS_years'] = p_BPM_count
    count = count + 1

fig, ax = plt.subplots()
ax.bar(stats_filt.WS_years.unique().tolist(), stats_filt.BPM_years.value_counts())
ax.set_title("Number of Players with 75th Percentile WS Years")
plt.show()


# In[10]:


topBPM = stats_filt[stats_filt['WS_years'] > 16]
s = topBPM['Player'].unique().tolist()
listToStr = ', '.join([str(elem) for elem in s])
print("Players with > 16 years of top 75% WS: " + listToStr)

topBPM = stats_filt[stats_filt['WS_years'] == stats_filt['WS_years'].max()]
s = topBPM['Player'].unique().tolist()
listToStr = ''.join([str(elem) for elem in s])
```

```
print("\nPlayer with most years in top 75% of league WS: " + listToStr + ' (' + str(stats_filt['BPM_years'].max())+' years)')


# In[17]:


#Data sets for classification
#everything, no advanced
basic_drop = ['Player', 'Pos','Age','Tm','Year','G','MP','GS','OWS','DWS','WS','WS/48',
              'BPM_annual_percentile','BPM_class','BPM_years','WS_annual_percentile','WS_class','WS_years']
X_e = stats_filt.drop(basic_drop, axis=1)
#per game only
keep = X_e.columns.tolist()[0:24]
X_pgo = X_e[keep]
#per36
keep = X_e.columns.tolist()[24:45]
keep.append('Decade')
X_pm = X_e[keep]
#per100 pos
keep = X_e.columns.tolist()[45:68]
keep.append('Decade')
X_pp = X_e[keep]
#only advanced stats (no win shares)
keep = X_e.columns.tolist()[68:80]
keep.append('Decade')
X_as = X_e[keep]


# In[18]:


#run PCA again... to use as data for classification.
#standardize data before PCA
X = StandardScaler().fit_transform(X_e)

#perform PCA
pca = PCA()
principalComponents = pca.fit_transform(X)
pDf_2 = pd.DataFrame(data = principalComponents[:,:2], columns = ['PC1', 'PC2'])
pDf_2['Decade']  = X_e['Decade'].tolist()


percent_explained_var = 100*pca.explained_variance_/sum(pca.explained_variance_)
cut_off = 10
cumulative = 0
cumulative_exp = [0]
required_modes = 0
first_time = True
for i in range(0, len(percent_explained_var)):
    cumulative = cumulative + percent_explained_var[i]
    cumulative_exp.append(cumulative)
    if cumulative > (100 - cut_off) and first_time:
        required_modes = i
        first_time = False


f, ax1 = plt.subplots(1, 1, figsize=(20, 8))
ax1.plot(np.arange(0,required_modes+1,1), cumulative_exp[required_modes]*np.ones(required_modes+1), 'k--')
ax1.plot(np.arange(1,len(pca.explained_variance_) + 1,1).astype(int), percent_explained_var, 'bo', fillStyle='none')
ax1.plot(np.arange(0,len(pca.explained_variance_) + 1,1).astype(int), cumulative_exp, 'r--')
ax1.axvline(x=required_modes, ymin=0, ymax=cumulative_exp[required_modes-1]/100., c='k', ls = '--')
ax1.set_xticks(np.arange(0,len(pca.explained_variance_) + 1,1))
ax1.set_title('Test 1 Explained Variance by Principal Component')
ax1.set_ylabel('Percent Explained Variance [%]')
ax1.set_xlabel('Principal Component')
ax1.set_xlim([0.5, 55])


# In[19]:


#Implement classification method, all stats, decade 9
decade = 9
X = X_e[X_e['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']
rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
```

```python
print ('\n\n\nLogistic regression classifier: All Stats\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr_e = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr_e)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr_e)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr_e, average='weighted')#get precision
R = recall_score(y_test, preds_logr_e, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr_e, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
scores_allstats = cross_validate(clf, X, y, cv=50, scoring=scoring)
f, ax = plt.subplots(1, 1)
ax.plot(np.arange(0,len(scores_allstats['test_accuracy']),1).astype(int), scores_allstats['test_accuracy'], 'r-x')
ax.set_title('Logistic Regression CV Accuracy. All Stats, 1990s prediction of WS')
ax.set_ylabel('Accuracy [%]')
ax.set_xlabel('Cross Validation Run')


# In[20]:


#Implement classification method, per game stats, decade 9
decade = 9
X = X_pgo[X_pgo['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']
rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: Per Game\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scores_pg = cross_validate(clf, X, y, cv=50, scoring=scoring)
```

```
# In[21]:


#Implement classification method, per possession stats, decade 9
decade = 9
X = X_pm[X_pm['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']
rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: Per Minute\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scores_pm = cross_validate(clf, X, y, cv=50, scoring=scoring)


# In[22]:


#Implement classification method, per possession stats, decade 9
decade = 9
X = X_pp[X_pp['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']
rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: Per Possession\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
```

```
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scores_pp = cross_validate(clf, X, y, cv=50, scoring=scoring)


# In[23]:


#Implement classification method, per possession stats, decade 9
decade = 9
X = X_as[X_as['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']
rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: Advanced Stats\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scores_as = cross_validate(clf, X, y, cv=50, scoring=scoring)


# In[24]:


#Apply the best performing model.. to the next decade, and see how well it predicts the WS.
#Implement classification method, all stats, decade 9
decade = 9
X = X_e[X_e['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']

next_decade = 0
X_nd = X_e[X_e['Decade'] == next_decade]#pull out data for the 90s
cats_nd = stats_filt[stats_filt['Decade'] == next_decade].loc[:,'WS_class']

rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
classes_nd, y_nd = np.unique(cats_nd, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: All Stats\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_nd, y_nd, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr_e = clf.predict(X_nd)#apply the model
aScore = accuracy_score(y_nd, preds_logr_e)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_nd, preds_logr_e)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_nd, preds_logr_e, average='weighted')#get precision
R = recall_score(y_nd, preds_logr_e, average='weighted')#get recall
F1 = f1_score(y_nd, preds_logr_e, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
scores_9_0 = cross_validate(clf, X, y, cv=50, scoring=scoring)
f, ax = plt.subplots(1, 1)
ax.plot(np.arange(0,len(scores_9_0['test_accuracy']),1).astype(int), scores_9_0['test_accuracy'], 'r-x')
ax.set_title('Logistic Regression CV Accuracy. All Stats, 1990s prediction of 2000s WS')
ax.set_ylabel('Accuracy [%]')
ax.set_xlabel('Cross Validation Run')


# In[25]:


#Apply the best performing model.. to the next decade, and see how well it predicts the WS.
#Implement classification method, all stats, decade 9
decade = 9
X = X_e[X_e['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']

next_decade = 1
X_nd = X_e[X_e['Decade'] == next_decade]#pull out data for the 90s
cats_nd = stats_filt[stats_filt['Decade'] == next_decade].loc[:,'WS_class']

rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
classes_nd, y_nd = np.unique(cats_nd, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: All Stats\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_nd, y_nd, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr_e = clf.predict(X_nd)#apply the model
aScore = accuracy_score(y_nd, preds_logr_e)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_nd, preds_logr_e)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_nd, preds_logr_e, average='weighted')#get precision
R = recall_score(y_nd, preds_logr_e, average='weighted')#get recall
F1 = f1_score(y_nd, preds_logr_e, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
scores_9_1 = cross_validate(clf, X, y, cv=50, scoring=scoring)
f, ax = plt.subplots(1, 1)
ax.plot(np.arange(0,len(scores_9_1['test_accuracy']),1).astype(int), scores_9_1['test_accuracy'], 'r-x')
ax.set_title('Logistic Regression CV Accuracy. All Stats, 1990s prediction of 2010s WS')
ax.set_ylabel('Accuracy [%]')
ax.set_xlabel('Cross Validation Run')


# In[26]:


#Apply a logistic regression model built from 2000s data.. to the 2010s decade, and see how well it predicts the WS.
#Implement classification method, all stats
decade = 0
X = X_e[X_e['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']

next_decade = 1
X_nd = X_e[X_e['Decade'] == next_decade]#pull out data for the 90s
cats_nd = stats_filt[stats_filt['Decade'] == next_decade].loc[:,'WS_class']

rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
classes_nd, y_nd = np.unique(cats_nd, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: All Stats\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_nd, y_nd, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr_e = clf.predict(X_nd)#apply the model
aScore = accuracy_score(y_nd, preds_logr_e)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_nd, preds_logr_e)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_nd, preds_logr_e, average='weighted')#get precision
R = recall_score(y_nd, preds_logr_e, average='weighted')#get recall
F1 = f1_score(y_nd, preds_logr_e, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
#Cross Validation
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
scores_0_1 = cross_validate(clf, X, y, cv=50, scoring=scoring)
f, ax = plt.subplots(1, 1)
ax.plot(np.arange(0,len(scores_0_1['test_accuracy']),1).astype(int), scores_0_1['test_accuracy'], 'r-x')
ax.set_title('Logistic Regression CV Accuracy. All Stats, 2000s prediction of 2010s WS')
ax.set_ylabel('Accuracy [%]')
ax.set_xlabel('Cross Validation Run')


# In[27]:


#Try PCA first and then run classification
pDf_2 = pd.DataFrame(data = principalComponents[:,:2], columns = ['PC1', 'PC2'])
pDf_2['Decade']  = X_e['Decade'].tolist()

#Apply the best performing model.. to the next decade, and see how well it predicts the WS.
#Implement classification method, all stats, decade 9
decade = 9
X = pDf_2[pDf_2['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']

next_decade = 1
X_nd = pDf_2[pDf_2['Decade'] == next_decade]#pull out data for the 90s
cats_nd = stats_filt[stats_filt['Decade'] == next_decade].loc[:,'WS_class']

rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
classes_nd, y_nd = np.unique(cats_nd, return_inverse=True)#get unique class names and translate to integer values
```

```
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: All Stats\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_nd, y_nd, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr_e = clf.predict(X_nd)#apply the model
aScore = accuracy_score(y_nd, preds_logr_e)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_nd, preds_logr_e)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_nd, preds_logr_e, average='weighted')#get precision
R = recall_score(y_nd, preds_logr_e, average='weighted')#get recall
F1 = f1_score(y_nd, preds_logr_e, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
scores_9_1 = cross_validate(clf, X, y, cv=50, scoring=scoring)
f, ax = plt.subplots(1, 1)
ax.plot(np.arange(0,len(scores_9_1['test_accuracy']),1).astype(int), scores_9_1['test_accuracy'], 'r-x')
ax.set_title('Logistic Regression CV Accuracy. All Stats, 1990s prediction of 2010s WS')
ax.set_ylabel('Accuracy [%]')
ax.set_xlabel('Cross Validation Run')


# In[28]:


#Try PCA first and then run classification
pDf_2 = pd.DataFrame(data = principalComponents[:,:5], columns = ['PC1', 'PC2', 'PC3', 'PC4', 'PC5'])
pDf_2['Decade']  = X_e['Decade'].tolist()

#Apply the best performing model.. to the next decade, and see how well it predicts the WS.
#Implement classification method, all stats, decade 9
decade = 9
X = pDf_2[pDf_2['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']

next_decade = 1
X_nd = pDf_2[pDf_2['Decade'] == next_decade]#pull out data for the 90s
cats_nd = stats_filt[stats_filt['Decade'] == next_decade].loc[:,'WS_class']

rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
classes_nd, y_nd = np.unique(cats_nd, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: All Stats\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_nd, y_nd, clf, classes)
clf.fit(X_train, y_train)#training the model
```

```
preds_logr_e = clf.predict(X_nd)#apply the model
aScore = accuracy_score(y_nd, preds_logr_e)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_nd, preds_logr_e)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_nd, preds_logr_e, average='weighted')#get precision
R = recall_score(y_nd, preds_logr_e, average='weighted')#get recall
F1 = f1_score(y_nd, preds_logr_e, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
scores_9_1 = cross_validate(clf, X, y, cv=50, scoring=scoring)
f, ax = plt.subplots(1, 1)
ax.plot(np.arange(0,len(scores_9_1['test_accuracy']),1).astype(int), scores_9_1['test_accuracy'], 'r-x')
ax.set_title('Logistic Regression CV Accuracy. All Stats, 1990s prediction of 2010s WS')
ax.set_ylabel('Accuracy [%]')
ax.set_xlabel('Cross Validation Run')


# In[29]:


#Try PCA first and then run classification
pDf_2 = pd.DataFrame(data = principalComponents[:,:10], columns = ['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7', 'PC8', 'PC9', 'PC10'])
pDf_2['Decade']  = X_e['Decade'].tolist()

#Apply the best performing model.. to the next decade, and see how well it predicts the WS.
#Implement classification method, all stats, decade 9
decade = 9
X = pDf_2[pDf_2['Decade'] == decade]#pull out data for the 90s
cats = stats_filt[stats_filt['Decade'] == decade].loc[:,'WS_class']

next_decade = 1
X_nd = pDf_2[pDf_2['Decade'] == next_decade]#pull out data for the 90s
cats_nd = stats_filt[stats_filt['Decade'] == next_decade].loc[:,'WS_class']

rs = random.seed(15)#set random seed
classes, y = np.unique(cats, return_inverse=True)#get unique class names and translate to integer values
classes_nd, y_nd = np.unique(cats_nd, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.33, random_state=rs)

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier: All Stats\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_nd, y_nd, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr_e = clf.predict(X_nd)#apply the model
aScore = accuracy_score(y_nd, preds_logr_e)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_nd, preds_logr_e)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_nd, preds_logr_e, average='weighted')#get precision
R = recall_score(y_nd, preds_logr_e, average='weighted')#get recall
F1 = f1_score(y_nd, preds_logr_e, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
scores_9_1 = cross_validate(clf, X, y, cv=50, scoring=scoring)
f, ax = plt.subplots(1, 1)
ax.plot(np.arange(0,len(scores_9_1['test_accuracy']),1).astype(int), scores_9_1['test_accuracy'], 'r-x')
ax.set_title('Logistic Regression CV Accuracy. All Stats, 1990s prediction of 2010s WS')
ax.set_ylabel('Accuracy [%]')
ax.set_xlabel('Cross Validation Run')
```