Major League Baseball Pitch Classification

Project for AMATH 582

Allyson Tom

16 March 2018

Abstract

This is a survey of several classification algorithms for the purpose of identifying pitch type in Major League Baseball. The dataset, from Pitchf/x, includes all pitches thrown in the 2015 regular season. The algorithms used are kNN, LDA, decision trees, random forests, SVM, and Naïve Bayes. Random forests ultimately give the best performance in terms of accuracy score on the test data, at around 77%.

SECTION I. INDTRODUCTION AND OVERVIEW

The best Major League Baseball pitchers each have several different types of pitches in their arsenal. Baseball is a situational and strategic sport, so the idea is that having more pitches available keeps batters guessing and keeps them off-balance, which leads to more unsuccessful at-bats. Every single pitch is decided before the throw, by the coach, catcher, and pitcher, as they consider facts such as the game situation, previous pitches thrown, batter history, and pitcher-hitter matchup, among others. Some of the most well-known pitch types are fastball, curveball, and changeup. Outside of the baseball community, some of the lesser-known pitch types are slider, splitter, and cutter. But how are these pitch types defined? What makes a splitter a splitter, a curveball a curveball? Speed, location, and movement are the top three variables used to classify a pitch. However, none of these variables alone can completely determine a given pitch. Not to mention, different pitchers can make the same pitch type look different. That, in fact, is part of the pitcher's advantage.

The purpose of this project will be to classify pitches based on several features. Although it may be easy for someone familiar with the sport to physically see and identify which pitch has been thrown, it is still important for this task to be automated. The MLB records lots of data – just about every statistic imaginable, even down to player height differential for a given day of the year at a given ballpark. It simply is not practical to have someone sitting behind home plate at every game, recording pitch location, break, speed, type, outcome, etc. For this reason, it is useful to have a pitch classifier that automates the process as measurements are taken electronically.

The Data

The data I am studying to solve the problem of pitch classification come from Pitchf/x. Pitchf/x is a technology created by *Sportvision*® and used by the MLB. Pitchf/x uses specialized cameras installed in all Major League ballparks to record highly specified measurements about

Figure 1. This is how a Pitchf/x screen might look to a fan watching the game on TV. Pitchf/x can map out the pitch's trajectory and its location as it crosses the plate, among other things. This makes the experience more informative and interactive for fans.

Public image provided by Sportvision®

Type	Abbreviation
Changeup	CH
Curveball	CB
Cutter	FC
Eephus	EP
Fastball, Four-Seam	FF
Fastball, Two-Seam	FT
Knuckleball	KN
Knuckle-Curve	KC
Pitch Out	PO
Sinker	SI
Slider	SL
Splitter	FS

Figure 2. List of pitch types used in this study and their abbreviations as identified byPitchf/x. The four most common pitch types are four-seam fastball, two-seam fastball, changeup, and curveball.

every pitch. This is useful to players and coaches especially, but also provides fans with a more interactive experience. For example, Pitchf/x allows fans to see mapped-out trajectories of pitches on the TV screen (see *Figure 1*). It is quickly revolutionizing the capture and use of pitching statistics in the big leagues. The Pitchf/x dataset I am using here captures essentially all variables related to pitching from the 2015 MLB regular season. I chose this dataset because the MLB's data comes from it as well, and MLB broadcasters trust it. MLB.com holds this data and it is also used in the MLB apps, suggesting that Major League Baseball trusts it.

The Pitchf/x dataset uses 18 different unique pitch classifications. I decided to simplify this list a bit; the clasification problem here deals with classifying 12 main pitch types. See *Figure* 2 for a list of these pitches and their associated abbreviations. Several of the variables in the Pitchf/x dataset are defined in terms of the *x*, *y*, and *z*-axes. Pitchf/x has established a universal frame of reference, which allows results from different ballparks to be compared. The graphic in *Figure 3,* taken from the Pitchf/x data glossary, depicts the frame of reference in which measurements are taken. The *initial_speed* variable describes the speed of the ball (miles per hour) as it leaves the pitcher's hand. Similarly, *plate_speed* describes the speed of the ball (miles per hour) as it crosses home plate. In feet per second, as describes from the catcher's perspective, *init_vel_x*, *init_vel_y,* and *init_vel_z* describe the velocities along the *x*, *y*, and *z*-axes, respectively, at 50 feet from home plate. The variables *bat_side* and *throws* identify handedness in the batter and pitcher, respectively. Consider that an inside pitch thrown to a lefty would be an outside pitch to a righty. Thus, two pitches with the same location can be very different in terms of what the batter sees, and will often result in different classifications. When the ball is about to be released by the pitcher, *init_pos_x* gives the horizontal location of the ball at the mound, and *init_pos_z* gives the ball height at the mound. These are both measured in feet. The variable *plate_x* describes the horizontal location of the ball from the center of home plate, and *plate_z* describes the height of the ball above the ground at home plate. These are again

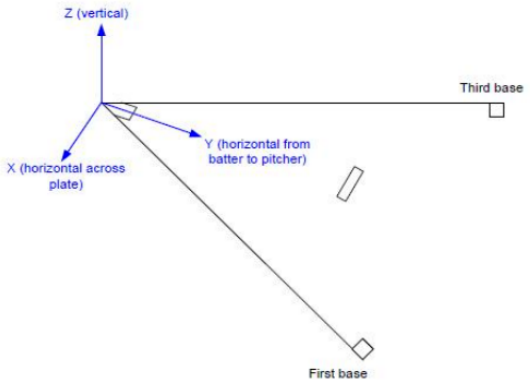

Figure 3. This graphic explains the Pitchf/x universal frame of reference. If Pitchf/x had not established a perspctive for the measurements taken, it would be imposibble to compare pitches thrown at different ballparks. Thus, this frame of reference is absolutely necessary for the measurements taken to be meaningful.

determined frm the catcher's perspective. Finally, movement describes how the ball travels while it is in the air. For example, curveballs typically move more than fastballs. A pitch that comes in a straight line from the mound to the plate can be describes as having no movement. These next few variables describe the movement of a pitch. Acceleration along the *x, y,* and *z* axes is given respectively by *init_accel_x, init_accel_y,* and *init_accel_z*. These are measured in feet per second from the catcher's perspective. The variables *break_x* and *break_z* give the horizontal and vertical movement of tha ball, respectively, and are measured in inches.

SECTION II. THEORETICAL BACKGROUND

Before handing the data over to a classification algorithm, we must prepare the data for training. One step in doing so is to compute the singular value decomposition (SVD) of the data matrix. The SVD is a matrix factorization. Given any matrix $\boldsymbol{A}$ with dimension m x n, the (reduced) SVD decomposes A into a product of three matrices: $\boldsymbol{U}$, an m x n matrix whose orthonormal columns form a basis for the column space of $\boldsymbol{A}$; $\boldsymbol{\Sigma}$, a diagonal n x n matrix; and $\boldsymbol{V^*}$, an n x n unitary matrix whose rows form a basis for the row space of $\boldsymbol{A}$.

$$\boldsymbol{A} = \boldsymbol{U\Sigma V}^* \tag{1}$$

If $\boldsymbol{A}$ has rank r, then exactly r of the diagonal elements σ_i of $\boldsymbol{\Sigma}$ will be nonzero; they appear along the diagonal of $\boldsymbol{\Sigma}$ in descending order. The σ_i's are ordered so that σ_1 corresponds to the mode accounting for the most variation, σ_2 the second most, and so on. In other words, they tell us specifically where quantities of the variation occur. Their sizes indicate where variation in the data occurs, and in which corresponding orthogonal directions $\boldsymbol{v_i}$ and $\boldsymbol{u}_i$, where the vectors $\boldsymbol{v}_i$ are the columns of V, and the vectors $\mathbf{u_i}$ are the columns of U. This leads to the following fact, which will prove especially useful:

> *For any N satisfying* $0 \leq r \leq N$*, the partial sum given below will yield the best rank N approximation to* $\boldsymbol{A}$ *in the sense of the* L^2*-norm:*
>
> $$\boldsymbol{A_N} = \sum_{j=1}^{N} \sigma_j \boldsymbol{u}_j \boldsymbol{v}_j^* \tag{2}$$

This fact can be used to reconstruct low rank approximations. By taking less than the full amount, which corresponds to rank r, we may take the above sum with $N \leq r$ terms, corresponding to the best rank N approximation to the data matrix. In essence, we can reconstruct the data to a desired accuracy by creating low-rank approximations, taking as many modes as we deem necessary. This property of the SVD allows us to extract low rank structure of the matrix $\boldsymbol{A}$, and determine the projection of our data onto its principal modes. It can also be used to determine which features are the most important in the dataset.

Now, I will briefly cover the theory behind each of the classification algorithms used here. These are all considered supervised (as opposed to "unsupervised") learning algorithms, as the labels are provided to each learning algorithm ahead of time.

- *k-Nearest Neighbors (KNN):* The kNN algorithm is one of the simplest and most intuitive of all classification methods. Put simply, the kNN algorithm labels a new point based on its k closest neighbors in feature space. A new data point is labeled by a vote among those neighbors. The votes can either be equally weighted, or weighted by distance to the point in question. Here, distance will be measured by the Euclidean metric. The user specifies the parameter k; different values of k will likely produce different results. Important considerations regarding this parameter include the following: a small k will be especially sensitive to noise, while k too large will defeat the purpose of considering the closest neighbors. Also, k should be chosen to be odd, so as to avoid ties in the voting process. See *Figure 4* for an example.

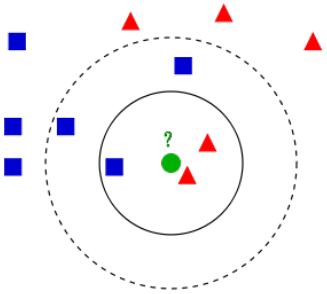

Figure 4. There are two classes: blue squares and red triangles. The green circle represents the data point we wish to classify. If k = 3, points considered lie within the solid black ring. The triangles win out in a 2 to 1 vote, so the green circle would be classified as a red triangle. If k = 5, points considered lie within the dotted black ring. Now, the squares win out in a 3 to 2 vote.

Source: From Wikipedia - By Antti Ajanki AnAj - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2170282

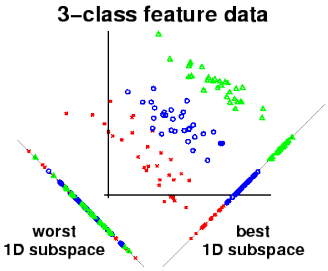

Figure 5. There are 3 classes: red, blue, and green. Projecting down to the line on the left provides no useful information for classification. However, projecting down to the line on the right clearly separates the 3 classes. LDA searches for such a line.

Source: https://www.quora.com/What-is-an-intuitive-explanation-forlinear-discriminant-analysis-LDA

- *Linear Discriminant Analysis (LDA):* LDA attempts to find linear separation between data points with different labels. If you were able to project your data onto a line, choosing the 'correct' line could give a lot of information about classifying new data points. The optimization involved in finding the 'right' line is exactly the problem that LDA solves. See *Figure 5* for an example of how this works.
- *Naïve Bayes:* A key assumption made by this type of classifier is that features are independent. In other words, if the break of a pitch in the x-direction is indicative of the fact that it is a curveball, this classifier will independently consider the speed in its contribution, even though the two features are likely correlated. This method uses probability to make its predictions. Despite being relatively easy and quick to train, it performs surprisingly well (in general) against more complex classification algorithms.
- *Decision Trees:* A decision tree moves from input data to labels via a series of decisions, based on feature values. This type of model assigns a label by creating a sequence of splitting criteria, like the branches of the tree. The aspects of the data on which to split are what the algorithm is learning from the training data. Say we were trying to classify apples and bananas. The tree might first split on color: red vs. yellow. Some apples that are greener in color might fall in the yellow category. So, the tree splits again: round shape vs. non-round shape, etc. And thus, the apple is classified differently from the banana. One clear

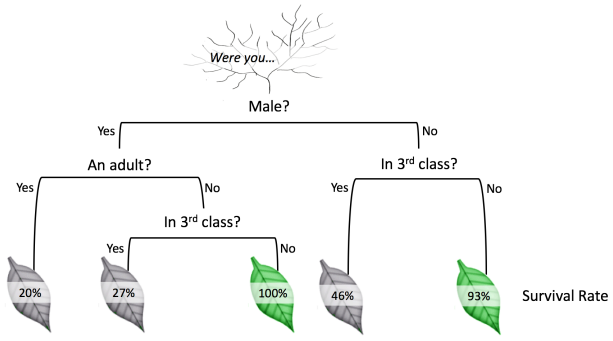

Figure 6. This is an example of a decision tree, whose goal is to determine survival of passengers on the Titanic. The first decision is a split on male-female. If the subject in question was male, the tree follows up by asking if the subject was an adult. Continuing in this manner, the tree gives each subject a survival rate.

Source: "Decision Trees: A Disastrous Tutorial", by Annalyn Ng. Obtained from https://www.kdnuggets.com/2016/09/decision-trees-disastrous-overview.html

advantage of decision trees is that it enables the user to see exactly how and why certain classifications are made. See *Figure 6* for an example.

- *Random Forests:* Random forests are an ensemble method, meaning that it uses several algorithms to make a prediction that is better than any single one of the algorithms could make on its own. The algorithms used in the ensemble are decision trees. The forest is made up of the decision trees, and the mode of the classes predicted by the trees is the forest's prediction. Random forests are widely applicable and are one of the most popular classification techniques.
- *Support Vector Machine (SVM):* SVM is a linear classifier that sets out to find the support vectors (lines) that provide maximum separation between data points, as they appear mapped in space. The algorithm penalizes points lying on the incorrect side of the support vectors, and in that sense it seeks to find the support vectors with the minimal number of misrepresented points. To classify a new point, the algorithm maps it in the same space as the training data, and looks at which side of the support vectors it lies. SVM is also one of the most popular and widely applied classification algorithms.

As a last but extremely important point, we discuss cross-validation. This is a necessary step, regardless of which classification technique is used. Because a test-train split introduces a random element to the results of our classification, we must guard against 'random' results. For instance, if we are trying to classify items in two classes, perhaps we just so happen to get good results the first time we train and test. To report accuracy scores based on this on attempt would be foolish, for another random split of the data could produce abysmal results based on the same random factor that gave us good results the first time. Thus, averaging scores over several different test-train splits allows us to get a better feel for the algorithm's true power on the given dataset.

SECTION III. ALGORITHM IMPLEMENTATION AND DEVELOPMENT

Using Python's Pandas library, I created a data structure called a DataFrame to hold all the observations and features for the purposes of cleaning the data. I will not go into the details of cleaning the data in this report, as the main purpose here is classification. In total, there are 699,773 observations (pitches), 17 features (measurements taken by Pitchf/x) per observation,

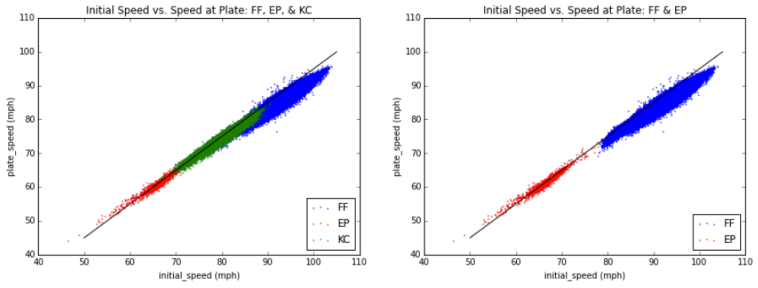

Figure 7a
A plot of initial speed vs. speed at the plate for four-seam fastballs, eephus pitches, and knuckle-curves.

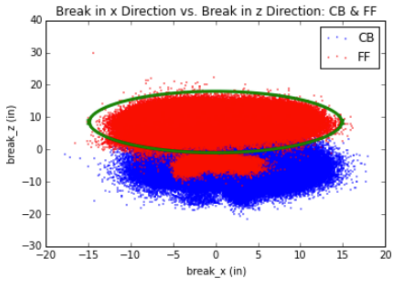

Figure 7b
A plot of break in the x- vs. z-directions.

The graphs in *Figure 7a* suggest that these variables will be particularly useful in distinguishing between FF and EP, while KC's have some overlap with the other two pitches. As another example, I plotted the break in the x-direction vs. break in z-direction for CB and FF. Some clustering became apparent. See *Figure 7b*. While there is some overlap, there appear to be two main ellipses of data points in the scatter plot.

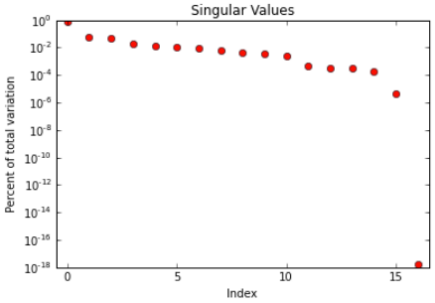

Figure 8. A plot of the principal modes of the data matrix, taken in feature space. The first singular value accounts for an incredibly large portion of the data, while the following 15 trail off. Finally, the last singular value contributes essentially no information as far as variation in the dataset. This information was used for feature selection and to reconstruct a lower-rank approximation of the data matrix.

and 18 unique classifications in the original dataset for the 2015 season. I simplified the dataset a little bit so as to exclude pitches that are thrown very rarely: if only 15 eephus pitches are thrown per season, this is only about .002% of all pitches thrown in a season, and I wasn't really concerned with picking that out. Again, see *Figure 2* for a table of every pitch type that is included in the classification task along with its identifying abbreviation in the Pitchf/x dataset; there are 12 in the cleaned dataset. I also dealt with duplicate variables and missing data.

Because of the large size of the dataset and the limited power of my laptop, I decided to look at a few statistics before I dove into the classification. I did this with the intention of determining which variables provide the most information. For instance, I plotted initial speed vs. speed at the plate for fastballs, eephus pitches, and knuckle-curves. See *Figure 7a* and *Figures 7b*. To simplify the computations, I decided to compute the SVD of my data matrix in search of some intuition as far as feature selection. See *Figure 8* for a plot of the principal components. This suggests that the last mode is contributing essentially nothing to the overall picture painted by the features; the first mode accounts for a very large portion of the total variation (82%, in fact). I found that the first 8 modes accounted for nearly 99% of the total variation, so I reconstructed the data matrix using only the first 8 prinicpal modes (a rank-8 truncation).

I tried several different classification algorithms for this task, but before running each one I split the data into a test set and a training set. This guards against overfitting, random results, and allows me to evaluate each classifier based on its performance classifying "new" data. I ran several algorithms on the data, but the process for each was essentially the same. Based on my intention for running the algorithm and the training time, I decided to vary some of the parameters from the default SciKit-Learn models. Finding optimal parameters is a difficult task, and oftentimes their values will vary wildly depending on the data. In each case, I would vary a single parameter at a time in a certain direction until the accuracy score stopped improving. Then, I would move to a second parameter and see how it affected the previously maximized accuracy score. Continuing in this manner, I attempted to get the best results I could. It should be noted that Python has functions for automating this process (such as GridSearchCV, which finds the best parameters from a given dictionary), but doing so that way took much more computing time on my computer, and several attempts to do it that was usually ended in me terminating the process manually before it completed. So, for those reasons, I went about parameter adjustment in a manual, but deliberate, manner. Once the data was split, trained and tested, I obtained an accuracy score for that particular split. Repeating this process and averaging the results gave me the final computational results that I will discuss in the next section.

SECTION IV. COMPUTATIONAL RESULTS

As mentioned above, I tried six different classification algorithms on the Pitchf/x dataset, in search of one that would perform the desired task with reasonable success. Throughout the process, I tuned some parameters, tried different test-train splits, and cross-validated my results. As expected, certain algorithms performed better than others. The worst performing algorithm in terms of accuracy score on the test data was naïve Bayes. The best performing algorithm in that same respect was random forests. Here, I will detail the results of handing my data to each algorithm.

- *kNN:* The accuracy score on test data for the k-nearest neighbors algorithm was 74.8%. To achieve these results, I set the number of neighbors to look at, k, to 100 data points. I also chose to weight votes by distance from the point in question. (Uniform weighting resulted in a 74.1% accuracy score.) The accuracy score seemed to increase as k increased up to (approximately) 100, and then on the other side, decreased as k increased past 100. For instance, using 1,000 neighbors only resulted in 71.1% accuracy; that is a lot more work for worse results. However, results were very close: choosing k in the range of 5 to 1,000, the accuracy score varied only by about 5%.
- *Naïve Bayes:* The accuracy score on the test set using Naïve Bayes was 58%. I simply took this algorithm straight out of the box from Python's SciKit-Learn module. Presumably, by using priors on the classes this performance could be improved, but I used Naïve Bayes as a baseline for results because of its simplicity and training speed. However, considering that there are 12 classes to choose from in classifying each new point, 58% is not bad; choosing a class at random would result (probabilistically) in an 8.3% accuracy score.
- *LDA*: The accuracy score on the test set using LDA was 60%. Similar to my approach with the Naïve Bayes algorithm, I again took this algorithm straight out of the box. I considered using Quadratic Discriminant Analysis as well, but did not achieve any improvement in results. In addition, some of the data in the Pitchf/x set is collinear, which most likely had a negative effect on the results obtained using discriminant analysis classifiers.
- *Decision Trees*: By constructing a decision tree to classify pitches based on the Pitchf/x data, I was able to achieve an accuracy score of 72.4%. Python allows the user to specify several parameters in the creation of a decision tree model, so I had a few decisions of my own to make. To measure the quality of a proposed split, I specified that the algorithm should use the information gain ("entropy"). I restricted the depth of the tree to 25. In order to split at a node, I required the tree to have at least 10 samples falling to that node. Similarly, I specified the number of samples at each leaf to be at least 10, so that the tree would not fit too specifically to the test data. These numbers are not large, given the sample size. The default parameters resulted in 61% accuracy, so tuning them to the given values gave quite an improvement.
- *Random Forests:* As has been the case in most of my experience comparing and contrasting these types of classification algorithms, random forests outperformed the rest. By tuning a few parameters from the default SciKit-Learn random forest implementation, I was able to achieve an accuracy score of 77% on the test data. This was attainable using 40 random trees per forest, splitting on information gain, and requiring a minimum of 8 samples to be at each leaf node.
- *SVM:* The SVM algorithm, as supplied by SciKit-Learn, yielded 73% accuracy. This was achieved using the radial basis function kernel, which is supplied by default. The SVM algorithm was by far the most time-consuming to train, and for this reason, I was not able to

tune parameters to the dataset as I was with most of the other classification algorithms. I assume that given the chance to try different parameters, at least a small improvement in accuracy stands to be made; that said, 73% accuracy is still pretty good.

SECTION V. SUMMARY AND CONCLUSION

The Pitchf/x dataset is an incredible one. There is so much more than what was described here, and it could be used for doing valuable research and tackling many interesting problems in baseball. Classification itself is an interesting problem, and I think the pitch classification problem specifically is an intriguing one that is important for the MLB. Pitchers need to have several pitches to be successful, but it is not feasible to have a human record all pitches. Especially with the capabilities of today's technology, having a human record each pitch is probably a "dumb" way to go about things. The results obtained here are not enough to perform the task at hand without human intervention to the level of accuracy that is necessary for sports statistics. Random forests were able to distinguish the 12 most common pitches with 77% accuracy. Although this isn't up to the desired accuracy level, it is a good start, and the results provide enough intuition about the problem to suggest that the desired level of accuracy is attainable with the variables that Pitchf/x measures. While location, speed, and trajectory are all important features, none of them alone can completely determine pitch type. It is the combination of the three that makes it possible to classify a pitch, as each variable and measurement tells us more about the pitch, and combining them gives us a nearly complete picture.

Appendix A. PYTHON FUNCTIONS

- *as_matrix* – converts Pandas DataFrame to matrix form.
- *astype* – converts data type as specified.
- *diag* – returns the diagonal elements of an array. Used to extract modes.
- *dot* - computes matrix products. Used in reconstructing data to lower rank.
- *dropna* – drops rows with missing data. Used to clean data.
- *fit* - fits a model to given data. Used to create classification models.
- *flatten* – flattens array along single dimension. Used to prepare labels.
- *from_csv* – reads data in from *.csv* file. Used to move data into Pandas DataFrame.
- *len* – returns size of largest dimension. Used in loops and to determine array size.
- *linspace* – returns array of equally spaced values.
- *mean* – returns the mean of an array. Used in mean centering of data.
- *plot* – creates a plot of specified arrays. Used to create graphs for report and analysis.
- *replace* – replaces values in an array as specified. Used to clean data.
- *scale* - scales the data to 0 mean and unit variance. Used in data preprocessing.
- *score* – returns cross-validation score of given model on test data.
- *shape* – returns the shape of an array. Used in loops and to determine sample size.
- *sqrt* – returns the square root of given value.
- *sum* – returns the sum of elements in an array. Used to scale singular values.
- *svd* – returns the SVD of a matrix. Used to determine principal modes.
- *train_test_split* – Splits given data into training and test sets of specified proportions.

Appendix B. PYTHON CODES

```
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
import numpy as np
from sklearn.decomposition import PCA
from sklearn import model_selection
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn import preprocessing
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

#load data from csv to pandas dataframe
pitchDF = pd.DataFrame.from_csv('drive-download-20161204T080739Z/Pitchfx.csv')

#drop unnecessary columns from the dataframe
pitchDF.drop(['game_id', 'game_date', 'sv_pitch_id', 'at_bat_number', 'pitch_number', \
             'inning', 'top_inning_sw', 'event_type', 'event_result', 'pre_balls', 'pre_strikes',
'pre_outs', \
              'batter_id', 'pitcher_id', 'HitTrajectory', 'PlaybyPlay', 'Runneron1st_ID', \
              'Runneron2nd_ID', 'Runneron3rd_ID'], axis=1, inplace=True)

#drop rows where any data is missing
pitchDF = pitchDF.dropna(axis=0)
#replace first list variables with second list to avoid duplicates and correct misclassifications
pitchDF = pitchDF.replace(to_replace = ['CU','XX','FO', 'IN', 'FA'], value = ['CB','UN','PO',
'UN', 'SI'])

#drop all pitches classified as UN, AB, or SC
pitchDF = pitchDF[(pitchDF['pitch_type'] != 'UN') & (pitchDF['pitch_type'] != 'AB') &
(pitchDF['pitch_type'] != 'SC')]

# drop pitches not from 2015
pitchDF = pitchDF[(pitchDF['year'] == 2015)]

#scatter plot of speeds

#first subplot grahps FF, EP, & KC
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
plt.scatter(pitchDF[pitchDF['pitch_type'] == 'FF']['initial_speed'].values,
pitchDF[pitchDF['pitch_type'] == 'FF']\
            ['plate_speed'].values, s=0.3, color='blue', label = 'FF')
plt.scatter(pitchDF[pitchDF['pitch_type'] == 'EP']['initial_speed'].values,
pitchDF[pitchDF['pitch_type'] == 'EP']\
            ['plate_speed'].values, s=0.3, color='red', label = 'EP')
plt.scatter(pitchDF[pitchDF['pitch_type'] == 'KC']['initial_speed'].values,
pitchDF[pitchDF['pitch_type'] == 'KC']\
            ['plate_speed'].values, s=0.3, color='green', label = 'KC')
#graph the approximate line of best fit
x_vals_speed = np.linspace(50, 105, 200)
y_vals_speed = [x-5 for x in x_vals_speed]
plt.plot(x_vals_speed,y_vals_speed,color='black')
plt.title('Initial Speed vs. Speed at Plate: FF, EP, & KC')
plt.xlabel('initial_speed (mph)')
plt.ylabel('plate_speed (mph)')
plt.legend(loc=4)

#second subplot grahps FF & EP
plt.subplot(1,2,2)
plt.scatter(pitchDF[pitchDF['pitch_type'] == 'FF']['initial_speed'].values,
pitchDF[pitchDF['pitch_type'] == 'FF']\
            ['plate_speed'].values, s=0.3, color='blue', label = 'FF')
plt.scatter(pitchDF[pitchDF['pitch_type'] == 'EP']['initial_speed'].values,
pitchDF[pitchDF['pitch_type'] == 'EP']\
            ['plate_speed'].values, s=0.3, color='red', label = 'EP')
plt.plot(x_vals_speed,y_vals_speed,color='black')
```

```
plt.title('Initial Speed vs. Speed at Plate: FF & EP')
plt.xlabel('initial_speed (mph)')
plt.ylabel('plate_speed (mph)')
plt.legend(loc=4)
plt.show()

#create scatter plot of movement

#plot CB and FF
plt.scatter(pitchDF[pitchDF['pitch_type'] == 'CB']['break_x'].values,
pitchDF[pitchDF['pitch_type'] == 'CB']\
            ['break_z'].values, s=0.3, color='blue', label = 'CB')
plt.scatter(pitchDF[pitchDF['pitch_type'] == 'FF']['break_x'].values,
pitchDF[pitchDF['pitch_type'] == 'FF']\
            ['break_z'].values, s=0.3, color='red', label = 'FF')

#plot the ellipse
x_vals = np.linspace(-15,15,100)
def pos_ellipseFF(x):
    return 8 + np.sqrt((10**2)*(1-((x**2)/15.**2)))
def neg_ellipseFF(x):
    return -np.sqrt(10**2*(1-(x**2/15.**2))) + 9
y_vals1 = [pos_ellipseFF(pt)  for pt in x_vals]
y_vals2 = [neg_ellipseFF(pt) for pt in x_vals]
plt.plot(x_vals,y_vals1, color='green', linewidth=3)
plt.plot(x_vals, y_vals2, color='green', linewidth=3)
plt.title('Break in x Direction vs. Break in z Direction: CB & FF')
plt.xlabel('break_x (in)')
plt.ylabel('break_z (in)')
plt.legend()
plt.show()

pitchDF.columns

# create data matrix and column of labels
data_mat = pitchDF.as_matrix(columns=['bat_side', 'throws', 'initial_speed', 'plate_speed',
'break_x', 'break_z',\
                                      'plate_x', 'plate_z', 'init_pos_x', 'init_pos_y',
'init_pos_z', 'init_vel_x', \
                                      'init_vel_y', 'init_vel_z', 'init_accel_x', 'init_accel_y',
'init_accel_z'])
labels = pitchDF.as_matrix(columns = ['pitch_type'])

print data_mat.shape
print labels.shape

# assign R -> 0, L -> 1 for handedness
for i in range(len(data_mat)):
    if data_mat[i,0] == 'R':
        data_mat[i,0] = 0.0
    else:
        data_mat[i,0] = 1.0
    if data_mat[i,1] == 'R':
        data_mat[i,1] = 0.0
    else:
        data_mat[i,1] = 1.0

data_mat = data_mat.astype(float)
data_mat = preprocessing.scale(data_mat)

u, s, vh = np.linalg.svd(data_mat, full_matrices=False)
print u.shape, 'u'
print np.diag(s).shape, 's'
print vh.shape, 'vh'

# find number of modes
plt.semilogy(np.diag(s)/np.sum(np.diag(s)),'ro')
plt.xlim([-.5,16.5])
plt.title('Singular Values')
plt.xlabel('Index')
plt.ylabel('Percent of total variation')

np.cumsum(np.diag(np.diag(s)/np.sum(np.diag(s))))
```

```
# truncate
data_mat = np.dot(u[:,:7],np.dot(np.diag(s)[:7,:7],vh[:7,:]))

# split the data into test and train sets
x_train, x_test, y_train, y_test = model_selection.train_test_split\
            (data_mat, labels, train_size = .7, test_size = .3)


labels = labels.flatten()
y_train = y_train.flatten()

'''#dictionary of parameters to try
params_knn = {'n_neighbors': [5, 100, 500, 1000, 10000], 'weights': ['uniform', 'distance']}

#cv = 3: 3-fold cross validation
grid_search = model_selection.GridSearchCV(estimator = KNeighborsClassifier(), param_grid =
params_knn)
grid_search.fit(data_mat, labels)'''

'''print "These are the best parameters and the best score for KNN."
print "Best parameters: ", grid_search.best_params_
print "Best CV score: ", grid_search.best_score_'''

mdl = KNeighborsClassifier(n_neighbors=100, weights='distance')
mdl.fit(x_train,y_train)

mdl.score(x_test,y_test)

mdl = SVC(kernel='rbf')
mdl.fit(x_train,y_train)

mdl.score(x_test,y_test)

clf = GaussianNB()
clf.fit(x_train,y_train)

clf.score(x_test,y_test)

clf = DecisionTreeClassifier(criterion='entropy', max_depth=20, min_samples_split=10,
min_samples_leaf=8)
clf.fit(x_train,y_train)

clf.score(x_test,y_test)

rf = RandomForestClassifier(n_estimators=40,criterion='entropy',min_samples_leaf=8)
rf.fit(x_train,y_train)

rf.score(x_test,y_test)

lda = LinearDiscriminantAnalysis()
lda.fit(x_train,y_train)

lda.score(x_test,y_test)
```