|  |
| --- |
| [Company name] |
| [Document title] |
| [Document subtitle] |

|  |
| --- |
| Ian Cheng  [Date] |

Contents

1 Analysis 3

1.1 Overview 3

1.1.1 Useful Knowledge 3

1.2 Research 4

1.2.1 Available Systems 4

1.2.2 Methods to the problem 4

1.2.2.1 Linear Regression 5

1.2.2.2 Limitations of Linear Regression 5

1.3 Proceeding method 6

1.3.1 Retrieving statistical data 6

1.3.1.1 Pandas 7

1.3.2 Data Storage 8

1.3.2.1 CSV 8

1.3.2.2 MySQL 8

1.3.2.3 SQLAlchemy 9

1.3.3 Generating team score and record 9

1.3.3.1 Scoring efficiently 9

1.3.3.2 ‘Free points’ 10

1.3.3.3 Offensive Rebounds 11

1.3.3.4 Protecting the ball 11

1.3.4 Optimization 12

1.3.4.1 Genetic Algorithms 12

1.4 Limitations 15

1.5 Objectives 16

2 Documented Design 18

2.1 Program Flowchart 18

2.2 Retrieving Statistical Data 18

2.2.1 Web Scrape algorithm 19

2.2.2 Manipulating data 19

2.2.3 External Libraries Used 21

2.3 Database connection and storage 21

2.3.1 Python-SQL connection 21

2.3.2 Database Storage 23

2.3.3 External Libraries Used 24

2.4 Pseudocode for data retrieval and storage 24

2.5 Querying 25

2.5.1 How to Query 25

2.5.2 Pseudocode for querying 26

2.5.3 External Libraries Used 27

2.6 Generating Team Score 27

2.6.1 Team score formula 27

2.6.2 Pseudocode for generating team score 28

2.7 Determining Record 28

2.7.1 Method 28

2.7.2 Getting the teams 28

2.7.3 Simulating games 29

2.7.4 Simulate Season 29

2.7.5 Pseudocode for determining record 30

2.7.6 External Libraries Used 32

2.8 Genetic Algorithm 32

2.8.1 Initialization 32

2.8.2 Fitness Function 32

2.8.3 Evolution – Selection 32

2.8.4 Evolution – Crossover 33

2.8.5 Evolution – Mutations 33

2.8.6 Pseudocode for genetic algorithm 33

2.8.7 External Libraries Used 34

2.9 Simulating next season 35

2.9.1 Simulating next season 35

3 Technical Solution 35

4 Testing 36

4.1 Retrieving statistical data + storage 36

4.1.1 Test Evidence 36

4.1.2 Retrieval Results Table 40

4.2 Querying 40

4.2.1 Roster 40

4.2.2 Stats list 41

4.2.3 Query Results Table 42

4.3 Generating Team Score 42

4.3.1 Neutral Team Score 42

4.3.2 Weighted Team Score 43

4.3.3 Generating Team Score Results Table 44

4.4 Determining Record 45

4.4.1 45

# Analysis

## Overview

In recent years, the usage of statistics in sports has grown in popularity. Statistical analysis has been used by sports teams to make major decisions such as which players to draft or trade for, how to play defense on specific teams and how much effect a player has on the team’s performance. This was first popularized by Oakland Athletics’ General Manager Billy Beane of Major League Baseball (MLB) where he prioritized the use of data analysis to make personnel decisions in the late 1990s. His success with the team, despite being at a heavy disadvantage in terms of the money they had to pay players, led to other teams in the MLB and more importantly franchises from other sports such as basketball to adopt the approach of using statistics.

There are a lot of research into statistical analysis in baseball but significantly less for basketball and with the growing popularity of basketball around the world (also being a big basketball fan myself), I decided to make a program that predicts the record (wins/ losses) for future seasons of any team in the National Basketball Association (NBA).

The NBA is a professional men’s basketball league based in North America with a total of 30 teams. It is regarded as the best professional basketball league with the best players in the world and in turn have many statistics that are recorded of players and teams. The goal for this program is to take these past statistics, process them and make a prediction on a team’s future record.

### Useful Knowledge

This section contains a list of knowledge / explanation of jargon that should be noted to better understand this report.

* There are two conferences in the NBA. The Western and the Eastern conference and each conference has 3 divisions which contain 5 teams each totaling to 30 teams in the NBA
  + Different divisions play differing number of games with different opponents
* But each team in the NBA plays 82 games total in a season
* Each game is 48 minutes split into four 12-minute quarters
* There are **no ties** in a game. If the scores are equal at the end of the regulation time, overtime (5 minutes) is played. If the scores are still equal, they play another overtime and keep on doing this until a winner is decided
* Record, presented like ‘wins-loss’ (49-33) are a way of showing how many games a team won and lost in a season. Adds up to 82
* Eight teams with the best record in each conference make the Playoffs

## Research

### Available Systems

I first searched on the internet to look if there are any systems that do something similar to this. I found that there are some systems that predict the result of individual NBA games and some systems which predict a team’s record as a whole.

Many of these systems are developed by betting websites. For example, oddsshark.com (a betting website) has a page on upcoming games and the odds of a team winning coupled with the odds of a bet. However, only stats that are of interest to bettors are displayed and “irrelevant” ones aren’t. Another example is FiveThirtyEight (538). 538 is a website that is focused on analysis of politics, economics and sports. They have a projected record displayed on their website predicted from their in-house system CARMELO (Career-Arc Regression Model Estimator with Local Optimization). CARMELO forecasts a player’s future performance based on trajectories of similar players and uses this to generate a team rating to develop of record projection.

Other projection systems are simply expert’s polls combined with non-expert voters. These are used to give teams a power ranking (A ranking of teams based on a derived rating) and determine a record this way. This is generally the opinion of a few individuals (in the case of NBA’s official power ranking, one person’s opinion). Though objective, still opinion based.

Upon my research, I also found some undergraduate and graduate thesis papers which tackles this problem.

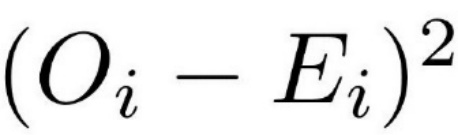
These are systems and people who also set out to tackle some variation of essentially the same problem. However, none of which is readily available for a user to use.

### Methods to the problem

During my research into available systems, I found that there are a few methods that can be used to tackle this problem

#### Linear Regression

Many of the aforementioned thesis papers used linear regression as a model to predict some result. This uses some variables to find the combination of coefficients that give the lowest sum of the squared differences between the observed dataset and the actual dataset.





*Where O is the predicted value and E is the true value.*

Team, opponent and location can be used as variables to predict point differentials or the teams can be used as a variable and scores as values or using a PER (a metric that measures a player’s effectiveness) as input to output a team’s win ratio.

#### Limitations of Linear Regression

A major limitation of using a linear regression model of past data is that it doesn’t account for players leaving or joining the team when predicting future records. Linear regression will find the relationship between variables and will base it upon the input data. i.e. the roster of the input team (id a player leaves the team, it will not affect the result).

A team’s roster is a very big factor in basketball. Even one player (especially if they are All Star caliber) joining or leaving a team can have a very big impact on the team’s performance. For instance, LeBron James and the Cleveland Cavaliers had a record of 61-21 in the 2009-2010 season, topping the eastern conference. However, next season when LeBron James left the team to join the Miami Heat, Cavaliers’ record plummeted to just 19-63. The following three seasons were also disastrous with their best record being 33-49 until LeBron James came back in the 2014-2015 season. Instantly their record increased to 53-29 becoming the second in the eastern conference and went to the NBA Finals, compared to them not even making the Playoffs the previous four years without James. Furthermore, LeBron James left again before the 2018-2019 season and (as of 4th March 2019) Cavaliers have a record of 19-48. With 15 games left to play this season, even if they win all of their remaining games (which is very unlikely) they would still have a losing record of 34-48.

This shows how important accounting for the players on a team is when predicting a team’s record.

## Proceeding method

I decided to come up with a solution that takes into account the roster of a team every year. This method first obtains all the data for every player in the NBA and in a form that can be easily requested. Secondly, the data will be combined in a way that would give every team a power score based on the stats of the players on the team. This will account for players changing teams as their value is added to their new team. With a score for every team, the program can compare the scores to each other. However, not every team will put the same emphasis on parts of the game. Some teams may aim to only take high percentage shots making their Field goal percentage higher than other teams or teams with very good Big men will try to get more offensive rebounds so the importance of each stat to each team is different and in turn the weighting of each stat is unique to the team.

So, this method will mainly rely on optimizing the weights put on each factor of the score. It will find the most representable weights for each team.

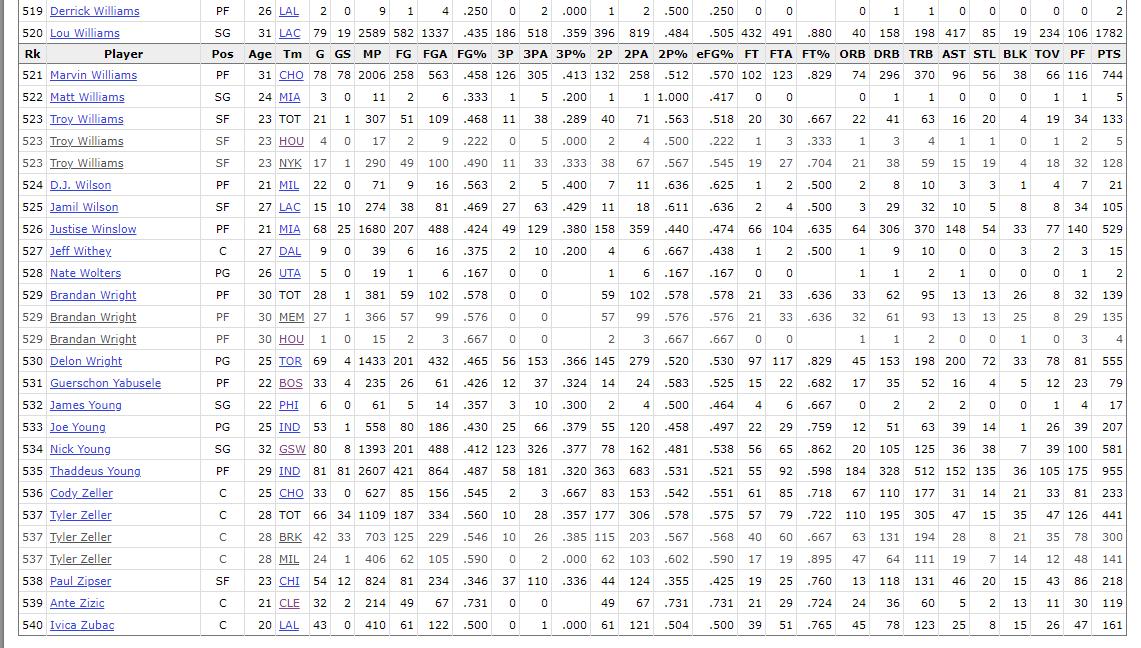
### Retrieving statistical data

First thing I need is to get ahold of the past data. The ideal situation would be an NBA statistics API where requests on the needed statistics can be made and the server sends that data back. This would be good as I will be able to get real time stats in the easiest way possible. However, stats.nba.com (Official NBA stats website) limits the number of APIs that are available to the public and other available APIs are paid subscription based making it very hard to retrieve stats this way. There are also no public databases that can be queried from.

Another valid option is to utilize web scraping. Even though there aren’t available databases, there are a lot of websites that display the required statistics. Basketball-reference.com is a particularly good website to web scrape from because they have every stat of every player per season displayed on one single web page. This means that everything in a season can be obtained with one single web scrape.



**Figure 1: Top of table on basketball-reference**

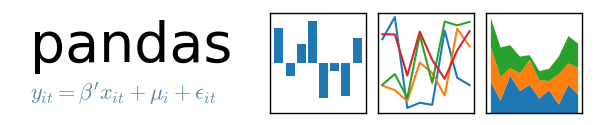


**Figure 2: Bottom of table on basketball-reference**

Another advantage is that the data is displayed in the form of a HTML table which will make it convenient for us to web scrape. Additionally, the difference between URL from season to season is just the year number so it’s easy to create a loop to web scrape from different seasons in the future.

#### Pandas

Pandas (Python data analysis library) is an external library for Python which is designed for data manipulation and analysis. There are a few inbuilt functions like read\_html() and to\_sql which are useful to what I want to do.



**Figure 3: Pandas logo**

### Data Storage

Web scraped data in Pandas is returned to a Pandas DataFrame which is an array that holds data with labels. This is no good because this is essentially a variable array which disappears when the program closes. There should be only one initial web scrape and store that data somewhere so that we don’t have to web scrape every time we run the program as this can take time to do. We also want to store it in a way that we can query players by team efficiently.

#### CSV

A way of storage can be putting the data into a comma separated values file. This allows us to only web scrape once and get the data from the CSV file when we need it. However, a disadvantage of storing in a CSV file is that even though we will not need to web scrape every time, we will still need to load the file into a DataFrame and store it at the start of the program then get the necessary stats. Considering the seasons, players and the number stats/ fields there are, this can take up more than necessary memory.

Considering these factors, I decided to store the stats in a database management system. Because of the nature of the data, it makes sense to store it in a relational database management system following the relational database model. Storing in a database solves all our problems. We only need one initial web scrape and the data does not need to be loaded into a variable at the start of the program. Instead we can query only the necessary data that we wish for. Avoiding unnecessary usage of memory to store the DataFrame.

#### MySQL

MySQL is a good database to use as it is open-source relational database management system. It uses tables as the main management system which works well with our database model.



**Figure 4: MySQL Logo**

#### SQLAlchemy

SQLAlchemy is a third-party toolkit that allows for database interactions with Python. It can create a connection between Python and the database whereby data can be queried or put in the database from a Python program.



**Figure 5: SQLAlchemy Logo**

### Generating team score and record

A team score will be generated by combining every team’s player statistics the same way. The method I will use is Dean Oliver’s “Four Factors of Basketball Success”. Dean Oliver is a statistician who contributed significantly to the use of APBRmetrics (usage of statistics in the analysis of basketball). These four factors are: to score efficiently, get as many ‘free’ points as possible, grab as many rebounds as possible and to protect the basketball. I am going to use these factors because they are closely related to the team’s performance. They are most related to the team getting as many points as possible (which is ultimately the decider for any basketball game.)

#### Scoring efficiently

The most efficient team in terms of scoring will be to make shots on every possession but it’s practically impossible that every possession will result in points added for the team. However, the more efficiently a team does this, undoubtedly, will be better off.

A general statistic to measure this is by Field Goal Percentage (FG%). This is essentially the number of shots a player makes divided by the number of attempted shots. However, this is not the best representation of scoring efficiency because on every possession, a player can either score a 2-point field goal or a 3-point field goal.

Consider Player A taking 10 shots and making 2 2-pointers and 2 3-pointers. His FG% would be 40% and he added 10 points to his team. Now consider Player B taking 10 shots but making 5 2-pointers and 0 3s. His FG% is 50% which is higher than Player A’s but he has added the same number of points to his team. He’s scoring ‘more efficiently’ even though he’s added the same value.

A solution to this is to use a metric called **effective Field Goal Percentage (eFG%).** eFG% accounts for the fact that the three point is worth 1 more than the two point.

**Figure 6: eFG% Formula**

**Figure 6** shows the formula for calculating effective field goal percentage. Considering the scenario above, both of the players now have eFG% of 50%. This gives us the best relative measure for points per field goal attempted. In other words, how efficient the player is scoring.

#### ‘Free points’

Another way of scoring points is at the free throw line. Every free throw is worth 1 point and are at a set range where 1 player shoots without anyone contesting which makes it a guaranteed attempt at scoring as oppose to possessions where teams are not guaranteed to have an attempt at scoring (turnovers).

**Figure 7: Free throw rate formula**

The free throw rate formula measures how often a team gets to the free throw line and how often they make them. The number of free throws made can be divided by the number Field Goal Attempts instead of the number of possessions is because usually teams average under one FGA per possession.

#### Offensive Rebounds

If a team is unable to score every possession, the best option is to grab an offensive rebound so that the team has a second chance of scoring. Offensive rebounds extend a team’s possession and gives them an extra attempt at a field goal.

**Figure 8: Offensive rebound formula**

**Figure 8** shows our formula for calculating offensive rebounds for a team. We divide the total number of offensive rebounds by 82 (number of games) and then divide by 100.

#### Protecting the ball

This is measured with turnover percentage (TOV%). A turn over means that a team does not get an attempt at field goal so the lower the TOV% the better. TOV% is calculated using the following formula:

**Figure 9: Turnover percentage formula**

Players who are on the court more, have the ball more and make more plays will more often than not will have more turnovers than other players. This formula takes that into account by including FTA and FGA.

By generating a power score for every team, a team’s record can be determined by comparing the score to other teams’ score. Every team plays each opponent different times but the general formula is that a team plays:

* 4 games against the 4 other teams in the **same division**
* 4 games against 6 teams that are in the **same conference** but **different division**
* 3 games against 4 remaining teams in the **same conference** but **different division** \*
* 2 games against the 15 teams in the **opposing conference**

\* five-year rotation determines which out of division team is played 3 times and which are played 4

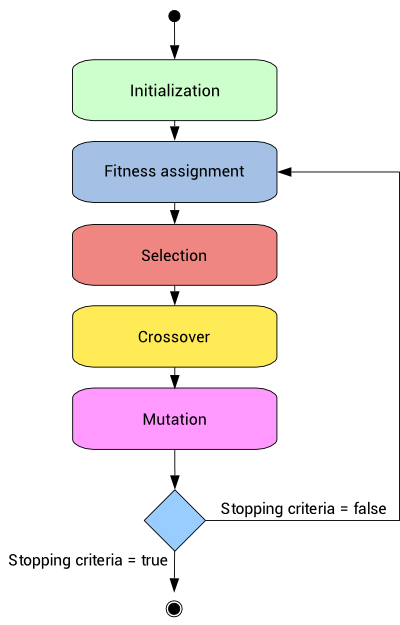
### Optimization

Dean Oliver assigned different weights to each of the four factor that we use to determine our power score. He says that scoring is worth 40%, turnovers are worth 25%, rebounding 20% and free throws are worth 15%. As I previously mentioned, different teams put different emphasis on parts of the game so it would be more representable to find the unique weights for each team.

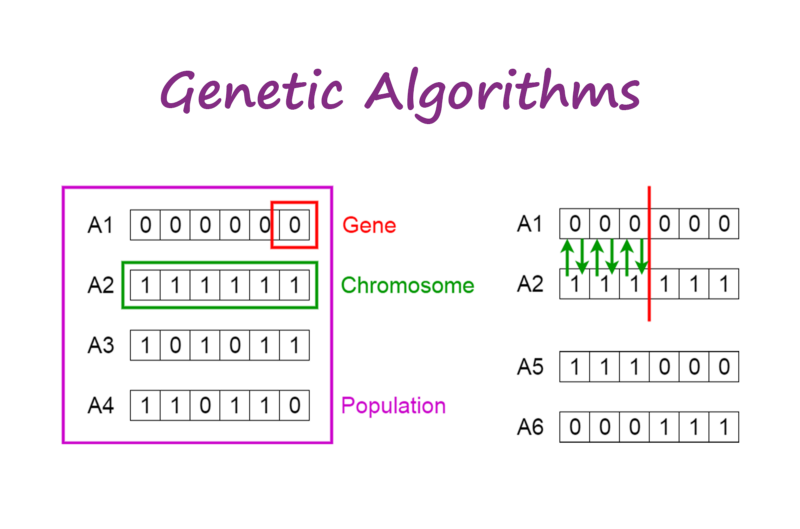
Because of the nature of basketball games, we cannot 100% predict the outcome of any given match therefore we can model the problem using a metaheuristic optimization algorithm that finds a sufficiently good solution.

#### Genetic Algorithms

Genetic Algorithm is a metaheuristic inspired and based on Charles Darwin’s Theory of Evolution and Natural Selection.



**Figure 10: Genetic Algorithm Stages**



**Figure 11: Genetic Algorithm model**

In a genetic algorithm, a population of individuals is evolved towards better solutions. **Figure 10** shows the stages in a genetic algorithm (GA).

**Initialization**

This is the first stage of a GA whereby randomly generated individuals create a population. This is to allow for the entire range of possible solutions to be assessed. However, sometimes the solutions can be ‘seeded’ which means that the range is restricted to an area where the optimal solution is most likely to be found. All the individuals are proposed solutions to the problem. A fitness function is also proposed. Fitness function is an objective that is to be reached with an individual, an optimal solution to the problem. Usually this is a single figure of merit.

**Fitness assessment**

The fitness of all the individuals in the population is assessed using the fitness function. This is usually evaluated on the basis of how close it is to the final solution.

**Evolution – Selection**

From selection onwards is where the process of evolution comes in. Selection from the existing population is made to breed new generations. Every individual in the population has a fitness score based on the fitness assessment. In the selection process, a portion of the top performers are selected to go to the next generation (Natural Selection). Similarly, lesser performers are also selected to promote genetic diversity. This is done to avoid getting stuck at a local solution and hence not being able to find the real optimal solution.

**Evolution – Crossover**

In the crossover stage, ‘parents’ (selected individuals) are bred together to form the new population. The ‘children’ are bred until the population returns to its original size. Crossover algorithms used are problem dependent but the most common ones are single point crossover: a point on the parents’ chromosome is randomly chosen and any genes behind that point is swapped between the two parents to form new off springs.



**Figure 12: Single Point crossover**

Multi-point crossover: two points on the parents’ chromosomes are randomly chosen and the genes in the segments are swapped



**Figure 13: Multi-point crossover**

Uniform crossover: Chromosomes are not divided into segments, rather each gene has a 50/50 change of getting swapped.



**Figure 14: Uniform crossover**

**Evolution – Mutation**

Mutation is a process where there is random chance that a random gene is randomly modified. This is also a process to promote genetic diversity.

**Stopping Criteria**

After every generation, the stopping criteria is assessed based on the fitness function to see if an individual has reached the optimal solution yet. If not, the algorithm repeats from fitness assessment and every stage is carried out again until the stopping criteria is reached.

## Limitations

Although the program doesn’t need to web scrape every time it runs, it does however, need to web scrape when statistics for the most recent season comes out. Basketball-reference updates the website every day but if we want to predict the record for next season, we have to wait until the current season is over to have all the data to process. This is also a second limitation as we don’t know a team’s roster for the season before hand, we can only make predictions when they come out which is usually close to the start of the season. For this reason, we are limited to predicting one season at a time. During the season, players can also be traded from team to team. Sometimes these can be very impactful players so a prediction at the start of the season is not always representable for the entire season, however, a trade deadline in the middle of the season (usually the start of February) means that there are rarely any big moves after this. Predictions after this date can be representable.

There are also uncontrollable factors that affect a player’s performance. This includes the frequency of games played in a given period of time and how much they need to travel. This can have an effect on a team’s performance due to players’ fatigue which will become more apparent later into the season. The difference in schedule between the seasons will mean that using past data as part of the model will not be as accurate.

## Objectives

**Setup**

1. Set up a database
2. Web scrape all relevant data from basketball-reference
3. Manipulate the data so that it is the correct data types
4. Manipulate the data so that it is compatible with SQL
5. Create a connection between MySQL database and Python using SQLAlchemy
6. Store all the data so that it can be called so the program doesn’t have to web scrape every time it runs

**Program**

1. Query current season players for the team
2. Query previous season relevant stats for the players
3. Combine stats of every player to produce a team score for every team
4. Simulate every game of the season for a team many times using randomly generated coefficients (weights on stats) to determine a record. Taking into account home court advantage and randomness of each game.
5. Using a Genetic Algorithm, evolve the coefficients by using the actual record as fitness function
6. Find the coefficients that determine the correct record for that team
7. Using these coefficients, simulate games of the next season
8. Do this 5 times and average the records to decrease the randomness (uncertainty)
9. Display the average predicted record for next season

# Documented Design

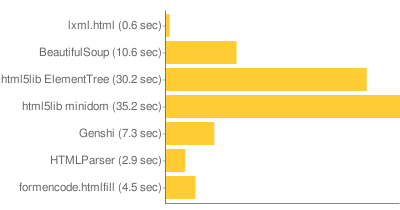
## Program Flowchart

**Figure 15: Program functions flowchart**

## Retrieving Statistical Data

The tools for web scraping I will use is the Pandas library for Python along with lxml, BeautifulSoup4 and html5lib libraries to parse HTML tables.

Pandas is used because they have an inbuilt function ‘read\_html()’ which makes it very easy to obtain data by automating the HTML parsing, resolving some issues that may occur. One of the issues is that the lxml library may fail to parse the table due to it having strict Markup Validations for HTML tables. So, if this happens the function switches to html5lib and BeautifulSoup4 automatically to guarantee a valid return. Using lxml as the primary library instead of html5lib and BeautifulSoup4 despite the issue is because lxml is much faster.



**Figure 16: HTML parsing library time comparisons**

### Web Scrape algorithm

<https://www.basketball-reference.com/leagues/NBA_2018_totals.html>

**Basketball-reference URL to total stats for season 17-18**

<https://www.basketball-reference.com/leagues/NBA_2017_totals.html>

**Basketball-reference URL to total stats for season 16-17**

Because the URL for pages on basketball-reference vary only by year number, it is possible to use a for loop to web scrape all the data.

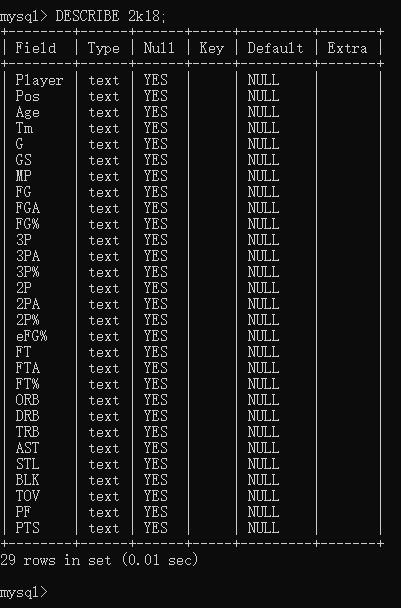
### Manipulating data

Due to basketball-reference’s data layout and data type representation, some modifications are needed after web scraping. The modifications will be done on every web scraped DataFrame to return a desired DataFrame which will then be stored.

1. On **Figure 1** and **Figure 2** (tables on basketball-reference) it can be seen that there are rows of headings in the middle of the table (occurs every 20 rows). This is for easy viewing on the website so that the viewer knows which numbers correspond to which stats when looking at the middle or bottom of the table. As this is no use to us, we need to delete all of these rows except for the top headings as this will be used as the heading in the database. Pandas has a drop() function which allows us to drop rows or columns. However, this only drops one row or a column at a time and because there are many of these redundant rows, we will need to us a different method.
2. stats = stats[stats['Player'] != 'Player']
3. # This get rids of all the rows which are headings in the middle of the table
4. # From 'column\_name' select all the rows that value != values (player)

\*\*\*from DataFrame ‘stats’, get all the rows in column ‘Player’ (header) which doesn’t have the value ‘Player’ and store it in new DataFrame called stats.

1. The headings on basketball-reference uses special characters like ‘%’ and ‘/’. These are invalid characters to use for headings in MySQL which means that if we keep the headings, we will not be able to query anything from the database as it will return an error. The heading will need to be changed to something that is accepted by MySQL. We do this by using the rename() function.
2. stats.rename(columns={'FG%':'FGp', '3P%':'3Pp', '2P%':'2Pp', 'eFG%':'eFGp', 'FT%':'FTp'} , inplace = True )
3. #Headings violate mysql rules with special characters. Changes headings to fit those rules.
4. The retrieved data are all in the text data type as shown in **Figure 17**.



**Figure 17: Data types before modification**

As most of everything stored is a number and that we are working with numbers, we need to change all the variables to the correct data type i.e. changing all the numbers to data type double and keeping all the strings as text. The function to\_numeric() changes all domains which contain a numeric number in other data types to doubles.

1. stats = stats.apply(pd.to\_numeric, errors = 'ignore')
2. #Changes all datatypes from string to double instead of numbers in text
3. #errors = ignore, keeps all the columns with just text as string
4. Some players’ name has an apostrophe in them. E.g. “D’Angelo Russell”. The apostrophe can cause problems when querying. We can remove apostrophe in the names. This won’t cause querying differences as long as remove it from all the names. This can be done by replacing every instance of apostrophe with nothing. This will join the name together. E.g. “D’Angelo Russell” becomes “DAngelo Russell”.
5. In 2014, there was a team name change. The Charlotte Bobcats changed names to Charlotte Hornets. With this, the team’s identifier also changed from “CHA” to “CHO”. To ensure that identifier matching still works with older seasons, we should change all instances of the old identifier, “CHA” to “CHO”.

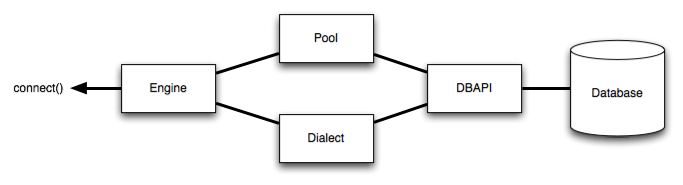
### External Libraries Used

1. **Pandas** **–** Web Scrape function
2. **lxml –** Main HTML table parsing
3. **BeautifulSoup4 / html5lib –** Backup HTML table parsing

## Database connection and storage

### Python-SQL connection

To create the connection between Python and MySQL, we use the external library SQLAlchemy and the connector PyMySQL. SQLAlchemy function create\_engine(), creates an Engine which connects to the DBAPI. The DBAPI (Python Database API Specification) is a commonly used specification for Python to define patterns for different connection packages. It is used so that the Python application can talk to the database. The Engine connects to the DBAPI through a connection Pool and a Dialect. These are instructions on how to talk to specific kinds of DBAPI.



**Figure 18: SQLAlchemy Engine Structure**

**Figure 18** shows a diagram of the Engine structure for connection. The Engine references both the Pool and the Dialect interpret the behavior of that specific database.

General form to create an Engine:

sqlalchemy.create\_engine(‘dialect+driver://username:password@host:port/database’)

Dialect is the identifying name for the dialect. In the case of SQLAlchemy it is the name of the database all in lowercase. E.g. ’sqlite’, ‘oracle’, ‘mysql’, ‘posgresql’ etc. For our Engine, it would be ‘mysql’. The driver is the name of the DBAPI used to connect to the database (also in lowercase). The driver I will be using is PyMySQL as this is a driver for MySQL. The rest of the parameter are for details of the database like the username and password, the host and the port and the name of the database within the system.

Our Engine:

1. **def** sqlconnection():
2. engine = sqlalchemy.create\_engine('mysql+pymysql://root:qazwsxedcrfv@localhost:3306/stats')
3. #dialect+driver://username:password@host:port/database
4. **return** engine

### Database Storage

The data will be stored in the database with each season occupying 1 relation. The attributes will be each player’s name and the stats names (Field goal, point per game, games played etc.). The tuple will the all the players’ numbers.

Relation variable

Attributes

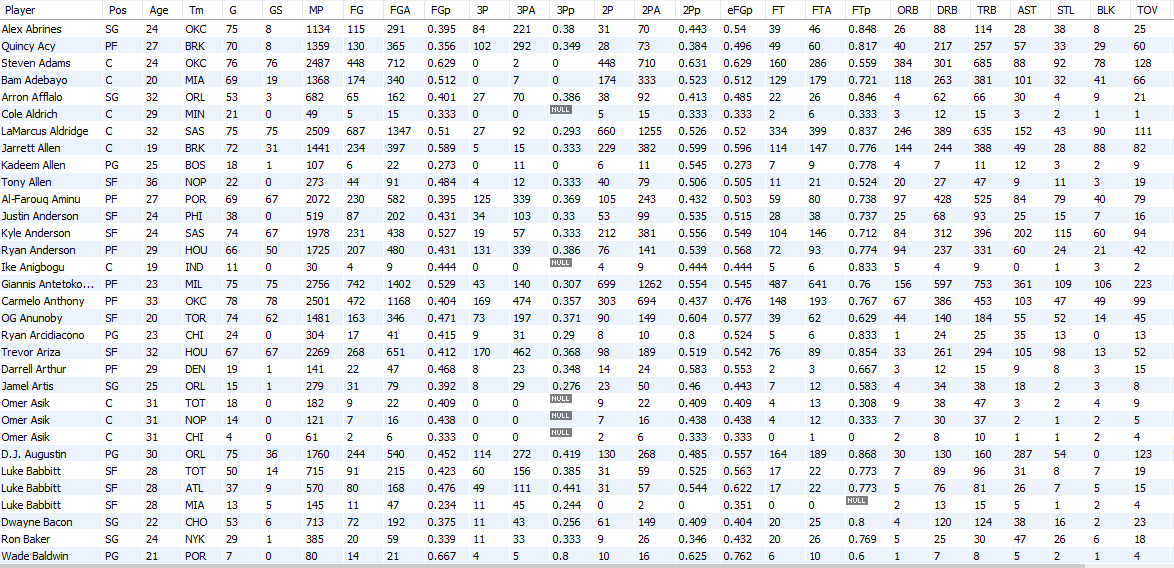
**Team Name**

Tuples



|  |  |  |
| --- | --- | --- |
| Player Name | … | … |
|  |  |  |
|  |  |  |
|  |  |  |

**Figure 19: Database model**



**Figure 20: MySQL 2018 Table**

**Figure 20** shows how a table looks like after all the stats have been stored. The tables in the database are kept like this and are not normalized. Normalizing the database serves to reduce data redundancy so that problems like data loss and insertion/ deletion anomalies doesn’t happen when the database is updated.

However, the reason why we don’t need to normalize it is because we will not need to update our database. The stats stored in the database will never change because it is what they got in that year. They can’t go back to 2015 and score 2 more points for example. For this reason, the aforementioned problems will not happen and the efficiency of querying will be improved as it will not need to call to different tables.

The stats are stored in the database after the modifications in the loop for web scrape.

1. **def** savetosql(df, num):
2. #This function puts the called dataframe into the database using the connection
3. engine = sqlconnection()
4. df.to\_sql(name = '2k' + num , con = engine, index = False,  if\_exists = 'replace')
5. # name = table name, con = connection if\_exists (replace everything in it)

This function is called which calls the connection function to create a connection in the variable engine. Pandas function to\_sql() takes in the parameters including the connection and puts the DataFrame into MySQL. This is done for each season.

### External Libraries Used

1. **SQLAlchemy** – To create a connection between Python program and MySQL
2. **PyMySQL –** Driver for SQLAlchemy connection
3. **Pandas –** Save DataFrame to MySQL

## Pseudocode for data retrieval and storage

Import libraries

def sqlconnection():

engine 🡨 create engine using SQLAlchemy

return engine

def savetosql(dataframe):

engine 🡨 sqlconnection()

dataframe to sql using engine

y 🡨 user input year

y2 🡨 y – 5

for range years y2 to y:

table 🡨 read\_html( front of URL + year number + end of URL)

Manipulation of the DataFrame

Savetosql(table)

## Querying

### How to Query

To query from Python code, SQLAlchemy is again used. SQLAlchemy can execute SQL commands written in the Python program in MySQL. The connection*.execute(query)* function takes in a parameter which is the query statement in SQL and executes this in the database linked to connection. The return result can be stored in a variable. The return result is an object as a result of the query. To get the actual data asked for, object*.fetchall()* function is used on the returned result.

When querying, we want to get the roster of the year we are predicting but the previous year stats for those players. This is because when we make the predictions, the season has not played yet so the players won’t have any stats.

The first step is to query for a return of all the players on the team for this year. Store all the players in a names list and use the names list to query all the stats associated with the player from the year before.

1. **def** roster(year, team):
2. #This function gets the roster from season 'year'
3. query = 'SELECT Player FROM 2k'+ str(year) + ' WHERE Tm = '+ '\'' + team + '\''
4. resultProxy = connection.execute(query)
5. #resultProxy is the object returned by .execute() method
6. resultSet = resultProxy.fetchall()
7. #Actual data requested when using fetch method on resultProxy
9. x = []
10. **for** row **in** resultSet:
11. x.append (row[0])
12. #fetchall() return all rows and all fields
13. #Need to iterate over the rows to access the fields and get the data
14. **return** x
15. #Roster for the year is in a 1D list and returned

This function gets the current roster. A for loop is used because the return we get is everything albeit these fields would be NULL values, we still only want the names. So, we only append the zeroth element of every row (this is the index where the names are) to the names list.

1. **def** stats(year, team):
2. #This function gets the previous season stats of players on roster
3. players = roster(year, team)
4. #players is a list with players from predict year
5. playerStats = []
6. **for** i **in** players:
7. query = 'SELECT \* FROM 2k'+ str(year - 1) + ' WHERE Player = ' + '\''+ i + '\' and Tm = \'TOT\' '
8. #Players who played for multiple teams will have multiple records in the table
9. #This is to Query for just the reccord of their totals as oppose to a specific  team
10. resultProxy = connection.execute(query)
11. resultSet = resultProxy.fetchall()
12. **if** **not** resultSet:
13. #If list is empty
14. query = 'SELECT \* FROM 2k'+ str(year - 1) + ' WHERE Player = ' + '\''+ i +  '\' '
15. resultProxy = connection.execute(query)
16. resultSet = resultProxy.fetchall()
17. **if** resultSet:
18. **for** row **in** resultSet:
19. playerStats.append( {'eFG%': row [16], 'FGA': row[8], 'FTA': row[18], 'TOV' : row[26], 'FT': row[17], 'ORB': row[20] })
20. **return** playerStats

This is the function for getting the stats. PlayerStats is a list of dictionaries. Each dictionary is the stats for each player. For example, playerStats = [ {Stephen Curry’s stats}, {Kevin Durant’s stats}, {Klay Thompson’s Stats} … ]. Doing this, the list is the stats for a single team however, the dictionaries will not have a player’s name tied to it because from this point onwards, it doesn’t matter which stats belongs to who as long as it’s a player on the same team.

### Pseudocode for querying

Import libraries

Engine 🡨 create engine using SQLAlchemy

Connection 🡨 engine.connect()

def roster(year, team):

SQL query statement 🡨 select players from ‘year’ where team == team

resultProxy 🡨 connection.execute()

resultSet 🡨 resultProxy.fetchall()

namesList 🡨 list

for row in resultSet:

append index of row in resultSet to names

return namesList

def stats(year, team):

names = roster(year, team)

for I in names:

SQL query statement 🡨 select stats from ‘year -1’ where player == names (i)

resultProxy 🡨 connection.execute()

resultSet 🡨 resultProxy.fetchall()

playerStats 🡨 append necessary stats to this list of dictionaries

return playerStats

### External Libraries Used

1. **SQLAlchemy** – To create a connection between Python and MySQL and to execute query statements in MySQL from Python
2. **PyMySQL –** Driver for SQLAlchemy connection

## Generating Team Score

### Team score formula

The Four Factors are going to be used to determine a team score. They are going to be combined with positive stat adding to the score and negative stat taking away from the score. Each factor is going to have a weight on it.

**Figure 21: General team score formula**

**Figure 21** shows the formula for calculating the team score. eFG%, ORB% and FTRate are all positive multipliers for a team. They should all add to the power score for the team whereas turnover percentage measures a team’s carelessness. The higher the TOV%, the worst off the team will be therefore this should be taken off of the power score. In the above formula x, y, z and w represent the weights for each team that is going to be optimized.

The factors in **Figure 21** are **team** factors. It is the sum of individual players’ factors

### Pseudocode for generating team score

Import functions from querying

def teamScore():

call function from querying to get list of dictionaries

loop through the items in list (dictionaries)

add each players’ factor score to the team’s factor score

apply formula for each factor

calculate team score

## Determining Record

### Method

For every team, we call the team score function to generate a score for every team.

To determine the record, we simulate every game 1000 times. Taking into account randomness and home court advantage. We simulate a total of 82,000 games per season for one team and count the number of wins. We don’t need to count the number of losses because the number of games per season is always 82. We just need to subtract the number of wins from that to determine the losses.

### Getting the teams

Every team is stored in a text file. One for teams in the East and one for teams in the West. We read the files into Python and create two 2d arrays. Each array inside the two 2d arrays is an array containing the five teams in the same division.

Example:

West = [[DEN,OKC,POR,UTA,MIN], [GSW,LAC,SAC,LAL,PHO], [HOU,SAS,DAL,NOP,MEM]]

East = [[TOR,PHI,BOS,BRK,NYK], [MIL,IND,DET,CHI,CLE], [CHO,ORL,MIA,WAS,ATL]]

The reason why we do this is because a team plays a different amount of games against opponents from different divisions and different conference. So, a 2d array allows for a call to the same division to play the same number of games and different division and conference for different number of games.

### Simulating games

When simulating a game, we randomly add a score from 0 to 0.25 for the home team and 0 to 0.15 for the away team. This gives a slight advantage to the home team as there is a chance of a higher number added to their score. The added-on score is random so that a team doesn’t have a 100% chance of winning from their calculated score previously as this is not the case in NBA games. Majority of NBA games have a score difference at the end of the game within 10 points and many games are decided on a last second shot so no team has a definitive chance of winning when playing against a specific opponent.

There will be a function to simulate a game when the team we are simulating is at home and when the team is away. The coefficients will only modify the main team we are simulating so the coefficients are passed as a parameter to modify either the home or the away team in each of the functions.

After the 1000 simulations, we calculate which team won more of the 1000 games and that team is determined to be the winner of that one game. If they win the same number of games, then it is counted as a win for the away team.

### Simulate Season

The function to simulate the season calls upon the simulate games function to simulate every game of the season to determine a record. Four games will be simulated against the four other teams in the same division. Two will be at home and two will be away. For the 15 teams in the other conference, two games will be simulated for each team with one at home and one away. As with the remaining 10 teams in the same conference but different division, 4 games will be played against 6 of these teams and the remaining 4 teams will be played 3 times. Which teams gets played three times or four times is determined by a five-year rotation in the NBA.

As I do not know the NBA’s five-year rotation and even if I did, it will be different for every team, I decided to random which 6 of the 10 teams gets played four times and which 4 gets played three times. For the teams that gets played four times, two will be at home and two will be away as with the teams that get played three times, once at home and once away. The last game will be random as to who will be home or away.

### Pseudocode for determining record

Import libraries

Import functions

def westTeams():

read text file for west teams

for line in file

append teams for first division

append teams for second division

append teams third division

close file

def eastTeams():

read text file for east teams

for line in file

append teams for first division

append teams for second division

append teams third division

close file

def simulateGameHome():

call function to get team scores

For 1000 times:

home team score add number between 0 - 0.25

away team score add number between 0 – 0.15

if homeScore > awayScore:

homeWin + 1

else

awayWin + 1

If homeWin > awayWin:

return won

else

return lost

def simulateGameAway():

call function to get team scores

For 1000 times:

home team score add number between 0 - 0.25

away team score add number between 0 – 0.15

if homeScore > awayScore:

homeWin + 1

else

awayWin + 1

If homeWin > awayWin:

return lost

else

return won

def SimulateWest():

same division – four teams

Simulate 2 games at home

Simulate 2 games away

Other conference

Simulate 1 game at home

Simulate 1 game away

Same conference different division

Random 6 teams to play 4 times

Simulate 2 at home

Simulate 2 away

Random 4 teams to play 3 times

Simulate 1 at home

Simulate 1 away

Random either simulate at home or away

def Simulateeast():

same division – four teams

Simulate 2 games at home

Simulate 2 games away

Other conference

Simulate 1 game at home

Simulate 1 game away

Same conference different division

Random 6 teams to play 4 times

Simulate 2 at home

Simulate 2 away

Random 4 teams to play 3 times

Simulate 1 at home

Simulate 1 away

Random either simulate at home or away

### External Libraries Used

**random** – generate random values to be added on to the teams score

## Genetic Algorithm

### Initialization

In this stage, a function is created to generate four random values to act as the coefficients. The range in which we generated our coefficients is seeded, meaning we choose it from a particular area where the solution is most likely instead of just any number. This range is from 0.75 to 1.25 inclusive. This range is chosen because the lowest score and the highest score these weights produce results in the team either losing or winning all of their games in a season. For example, the lowest score a team can have is when the first three factors have the weight of 0.75 and the TOV% factor have the weight of 1.25. This results in the team winning zero games the entire season. On the other hand, the highest score a team can have is when 1.25 is the weight of the first three factors and 0.75 as the weight of TOV%. This will result in any team winning all 82 games of their entire season. With these two extremes, we can conclude that the optimal result for the weights to predict how many games a team wins will lie in this range making it suitable to use this range. Seeding the range can decrease the time of convergence to a result.

The population size will be 12. This is quite small but it is a tradeoff between program run time and how accurate each population is. However, because our initialization is seeded to a range of 0.5 a small population wouldn’t cause a big issue in accuracy so the advantages gained in decreased run time outweighs the disadvantage to accuracy issues.

### Fitness Function

The fitness assessment criteria used is the record for the previous year. The fitness function will call the function to determine a record and compare it to the actual record. The function returns an absolute value of the observed number of wins and the expected number of wins. This essentially is a measure of how far away from the correct solution the individuals are.

### Evolution – Selection

The selection process is simple. We take the top performing one third of the individuals and randomly select one third of the worse performing ones. This is to promote genetic diversity. Two thirds of the population are selected to be parents meaning there will be a low crossover rate as only one third (four) individuals are bred. This is because a low crossover rate will keep more of the strong performing individual’s gene.

### Evolution – Crossover

With two thirds of the population selected previously, we now have to breed the last one third of the population so that it returns to the original size. Different crossover methods have benefits and downsides to them. A single point crossover means that the off-springs will not be that genetically diverse from their parents, keeping much of the traits of the parents. A uniform crossover will create off-springs that are very different from their parents.

Considering this, the breeding algorithm I chose to use is the multipoint crossover method. This is a mix of single-point and uniform crossover. I do not want the off-springs to be too different from the parents (as the seed range is small so there is accuracy already) but I also want it to diversify enough (as population size is small) so that it can actually converge even if initializing population is poor. A multipoint crossover gets the best of both worlds so it’s a suitable crossover technique to use.

### Evolution – Mutations

The chance of mutations is chosen to be 5%. A random individual in the population is chosen and, in the individual, a random gene is chosen to be mutated. In mutation, we replace the gene with a randomly generated number in our seed range.

### Pseudocode for genetic algorithm

Import libraries

Import functions

def individual(n,range):

list 🡨 n numbers in range

return x

def population (count,n,range):

loop in range count

list 🡨 individual (n,range)

return population

def finessFunction():

fitnessScore 🡨 absolute value of difference between expected and observed

return fitnessScore

def evolution(target, pop):

x 🡨 fitness of passed in population

if x:

return pop

while true

append the best soring individuals to a new list

random select left over individuals

while len(newPop) ! = len(oldPop):

multipoint crossover breeding

random father

random mother

random first point

random second point

everything in mother between first and second point is now in father

everything in father between first and second point is now in mother

append to newPop as off-spings

mutations

5% chance

Random individual

Random gene

Random new value

Gene replaced with new value

### External Libraries Used

**Random** – generate random values for: individual initialization,

- Generate random parameters for selecting lesser performers, mutations, parents, crossover positions

## Simulating next season

### Simulating next season

Once the genetic algorithm has ran and determined the coefficients, these coefficients will be used to simulate the games for next season by calling the determine record function. Instead of coefficients from the genetic algorithm, the predicted coefficients are passed in. A predicted record is generated off of this.

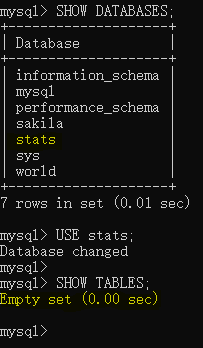
# Technical Solution

# Testing

## Retrieving statistical data + storage

### Test Evidence

This sub section will test the function that **web scrapes**, **changes** and **stores** the statistics.



**Figure 22: Empty Database**

**Figure 22** shows the database ‘stats’ before the Python script is run. As shown, the database is empty.

After the script is run, what **should** be seen is:

* Five tables of increasing year up until (not including) the predicting year.
* Not useful rows not stored
* All datatypes should be changed to be the correct ones

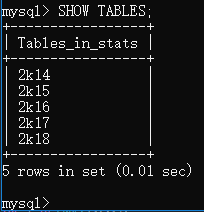
See 2.2.2

* All heading names adjusted to fit SQL rules
* Remove all instances of apostrophe in players’ names
* Adjust the team name



**Figure 23: Stat Retrieve - Test 1**

**Figure 23** shows the test with the year 2019 as input. What should happen is 5 tables should be stored in the database from (year – 5) to (year – 1).



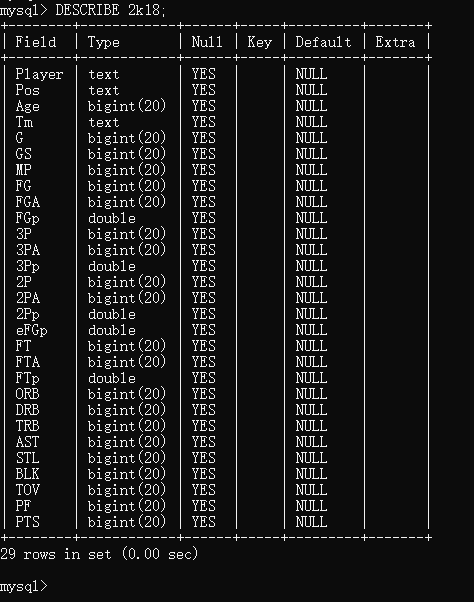
**Figure 24: Stat Retrieve - Result 1.1**

**Figure 24** show the result for our first test above. It is exactly as we expected with 5 tables each for a different year.



**Figure 25: D'Angelo Russell**

**Figure 25** shows D’Angelo Russell’s name as a value in MySQL. It can be seen that the apostrophe in the name has been removed.



**Figure 26: Stat Retrieve - Result 1.2**

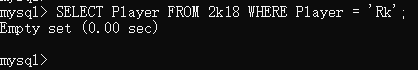
**Figure 26** shows the return when we ask for description of table in SQL. In the field column, all the headings do not contain any special characters and in the type column everything is in the correct data type as opposed to the result before changing (See **Figure 17: Data types before modification**)



**Figure 27: Stat Retrieve - Result 1.3**

**Figure 27** Shows the result when querying for the team names in the year 2014. Because the team name change happened in 2014, it didn’t take effect until 2015. However, from our query, the team name stored is correct meaning that we changed it from “CHA” to “CHO”.

To test if we have removed all the useless rows in the middle of the table, we can query for the value ‘Rk’ in the column headed ‘Player’. This is the value of rows we do not want. If the query returns some rows, then it means that we were unsuccessful in removing these rows. However, if the query returns nothing then we were successful.



**Figure 28: Stat Retrieve - Result 1.4**

From **Figure 28**, we can see that we have removed all of the rows we don’t want as an empty set is returned when we try to query for them.

### Retrieval Results Table

|  |  |  |
| --- | --- | --- |
| **Task** | **Description** | **Result** |
| Store season stats | Each season stored as a table with a total of 5 tables for 5 previous seasons from predicting year | ✔ |
| Remove rows | Some rows from the website are not useful. This should be removed and not stored. | ✔ |
| Change datatypes | Web scraped data are all in the text datatype. Need to changed numbers to correct datatype and keep text as text. | ✔ |
| Adjust headings | Web scraped data headings violate SQL special character rules. Need to change this to abide by the rules. | ✔ |
| Adjust player names | Player with apostrophes in their names can cause problems when querying later. Need to adjust names to remove all apostrophes. | ✔ |
| Adjust team name | There was a team name change in the NBA. Need to adjust this so there is no error when value matching | ✔ |

## Querying

### 4.2.1 Roster

When querying, we need to first get all the players on the roster. The function should query into MySQL for all the names of the player on the team and should put the names into an array. The function should return a list of names.

**def** roster(year, team):

#This function gets the roster from season 'year'

query = 'SELECT Player FROM 2k'+ str(year) + ' WHERE Tm = '+ '\'' + team + '\''

resultProxy = connection.execute(query)

''' This executes the query statement, 'query', from dB in connection and

stores it in resultProxy '''

#resultProxy is the object returned by .execute() method

resultSet = resultProxy.fetchall()

#Actual data requested when using fetch method on resultProxy

x = []

**for** row in resultSet:

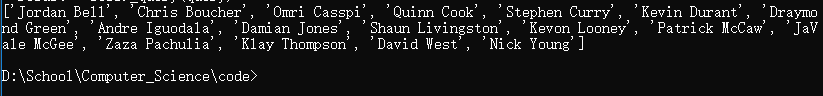
x.append (row[0])

#fetchall() return all rows and all fields

#Need to iterate over the rows to access the fields and get the data

**return** x

#Roster for the year is in a 1D list and returned



**Figure 29: Querying - Result 1**

**Figure 29** shows the result when calling the function for the roster. The function returns a list of names, like it’s supposed to.

### 4.2.2 Stats list

Next is to get the stats for these players. In order for the data queried from the database to be used in the next stage, the functions have to return a list of dictionaries. With each index in the list being a dictionary and each dictionary containing the stats of one player with the key being a particular stat and the value as the value of that stat for that player. The dictionary should have multiple key and value pairs as each pair corresponds to one stat.

**def** stats(year, team):

#This function gets the previous season stats of players on roster

players = roster(year, team)

#players is a list with players from predict year

playerStats = []

**for** i in players:

query = 'SELECT \* FROM 2k'+ str(year - 1) + ' WHERE Player = ' + '\''+ i + '\' and Tm = \'TOT\' '

''' Players who played for multiple teams will have multiple records in the table

This is to Query for just the reccord of their totals as oppose to a specific team '''

resultProxy = connection.execute(query)

resultSet = resultProxy.fetchall()

**if** not resultSet:

#If list is empty

query = 'SELECT \* FROM 2k'+ str(year - 1) + ' WHERE Player = ' + '\''+ i + '\' '

resultProxy = connection.execute(query)

resultSet = resultProxy.fetchall()

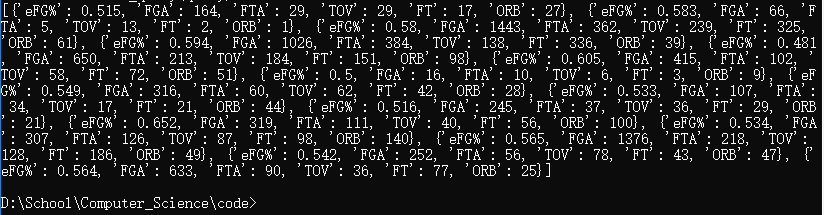
**if** resultSet:

**for** row in resultSet:

playerStats.append( {'eFG%': row [16], 'FGA': row[8], 'FTA': row[18], 'TOV' : row[26], 'FT': row[17], 'ORB': row[20]})

**return** playerStats

#This is a list with dictionaries as elements



**Figure 30: Querying - Result 2**

**Figure 30** shows the result when calling the function to get stats. As seen, it was successful as a list of dictionaries of stats was returned.

### 4.2.3 Query Results Table

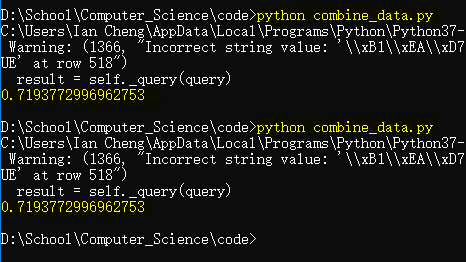
|  |  |  |
| --- | --- | --- |
| **Task** | **Description** | **Result** |
| Query Roster | Query the names of the players from MySQL and return the names in a list | ✔ |
| Query Stats | Using the list of names, query the stats of those player from MySQL and return it in a list of dictionaries | ✔ |

## Generating Team Score

### Neutral Team Score

First, we test the program in generating the neutral team score. This is when the weights on all the factors are 1. Every time this runs, the score should be the same.

**print**( teamScore(18,'GSW', [1,1,1,1]) )



**Figure 31: Team Score - Result 1**

**Figure 31** shows the result when generating Golden State Warriors’ neutral score from 2018. As can be seen, the score is the same every time it is run.

### Weighted Team Score

The returned score should be different with different weights. We will test it with the highest number and the lowest number it can get.

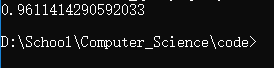
**print**( teamScore(18,'GSW', [0.75,0.75,0.75,1.25]) )



**Figure 32: Team Score - Result 2**

**Figure 32** shows the resulting score when the first three weights is 0.75 and the last weight is 1.25. These are the lowest and the highs number the weights can take on. Meaning this is the lowest team score.

**print**( teamScore(18,'GSW', [1.25,1.25,1.25,0.75]) )

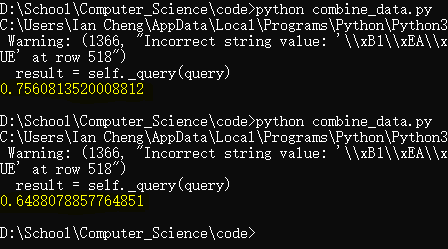


**Figure 33: Team Score - Result 3**

**Figure 33** shows the other end of extreme weights. This is the highest team score.

**print**( teamScore(18,'GSW', [1.02,0.83,1.15,0.91]) )

**print**( teamScore(18,'GSW', [0.98,1.21,0.73,1.20]) )



**Figure 34: Team Score - Result 4**

**Figure 34** shows the team score with two sets of four random weights. The return value lies in between the extreme results. Which makes sense.

**Weighted Score Results table for GSW**

|  |  |  |
| --- | --- | --- |
| Weights | Nature of weights | Score |
| [0.75,0.75,0.75,1.25] | Extreme | 0.4776131703333474 |
| [0.98,1.21,0.73,1.20] | Normal | 0.6488078857764851 |
| [1.00,1.00,1.00,1.00] | Neutral | 0.7193772996962753 |
| [1.02,0.83,1.15,0.91] | Normal | 0.7560813520008812 |
| [1.25,1.25,1.25,0.75] | Extreme | 0.9611414290592033 |

### Generating Team Score Results Table

|  |  |  |
| --- | --- | --- |
| **Task** | **Description** | **Result** |
| Generate Neutral Score | Neutral score with weights 1. This is the base team score and is the basis on how the weights edit the score. | ✔ |
| Generate Weighted score | Weighted team score base on neutral score. Will be used in support of genetic algorithm | ✔ |

## Determining Record

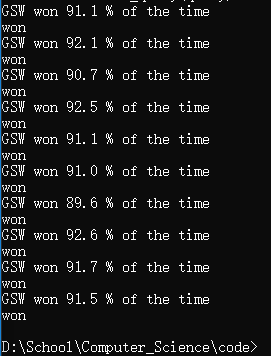
### Simulating home & away games

When simulating games, a random score is added on to both teams. The size of that score depends on if they are home or away. As every game is simulated 1000 times, this means that a team shouldn’t win the same number of those games every simulation.

I can test this by creating a small loop and simulate the same game between two teams

**for** i in range(10):

**print**( simulateMainAtHome(score, 'GSW', 'ORL', 18) )



**Figure 35: Determining Record - Result 1**

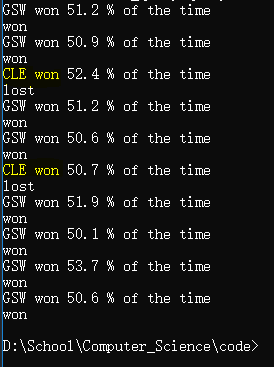
**Figure 35** shows the result when we run the small test above on the function. The test runs the 1000 games simulation 10 times. As shows in the result, even though GSW (Golden State Warriors) won all 10 times against ORL (Orlando Magic), the number of times they won each game differed. This result is reliable because it is consistent with real life. GSW is one of the best teams in the NBA whereas ORL is one of the worst so it makes sense that Golden State has a very high chance of winning a game however that chance is not 100%.

Looking at another test.

**for** i in range(10):

**print**( simulateMainAtHome(score, 'GSW', 'CLE', 18) )

This test pits GSW against CLE (Cleveland Cavaliers).



**Figure 36: Determining Record - Result 2**

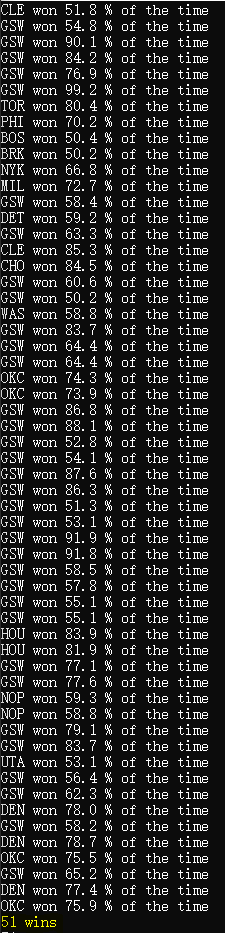
As can be seen in **Figure 36**, this time GSW has won much less of the time and even losing on two games. This test further shows the reliability of the model because the Cavaliers was the team that the Warriors played against in the playoff finals that year. So, it’s natural that they have less of a chance to win and even lose to a much better team than previous.

### Simulating Seasons

Each game of the season should be simulated and given a winner. The record should be determined by this and the number of win outputted.

**print**( simulateWest(west[1][0], 1, 0, 18, [1,1,1,1] ) )

Running the test above, the program returns:



**Figure 37: Determining Record - Result 3**

This is a successful test as the function simulates all the games and determine a record. This record might seem low now but that is because this was tested using neutral coefficients as I am only testing if it’s doing what it should.

### Determining Record Results Table

|  |  |  |
| --- | --- | --- |
| **Task** | **Description** | **Result** |
| Simulate Game | Add random score to create home court advantage and randomness and compare the teams to determine a majority winner | ✔ |
| Simulate Season | Simulate every game of the season by comparing scores to determine a winner. Display the number of wins | ✔ |

## Genetic Algorithm

### Generic Genetic Algorithm

Firstly, I built a generic genetic algorithm that generates four numbers where the sum is equal to 150. A working algorithm will be the basis of my main Genetic Algorithm.

Running this code on the functions…

target = 150

repeat = True

testPop = population(20,4,0,50)

**print**(testPop)

count = -1

**while** repeat == True:

**for** i in testPop:

**if** fitnessFunction(i,target) == 0:

count = count + 1

**print**('Solution found after', count,'evolution(s)')

**print**(i)

repeat = False

**break**

**else**:

count = count + 1

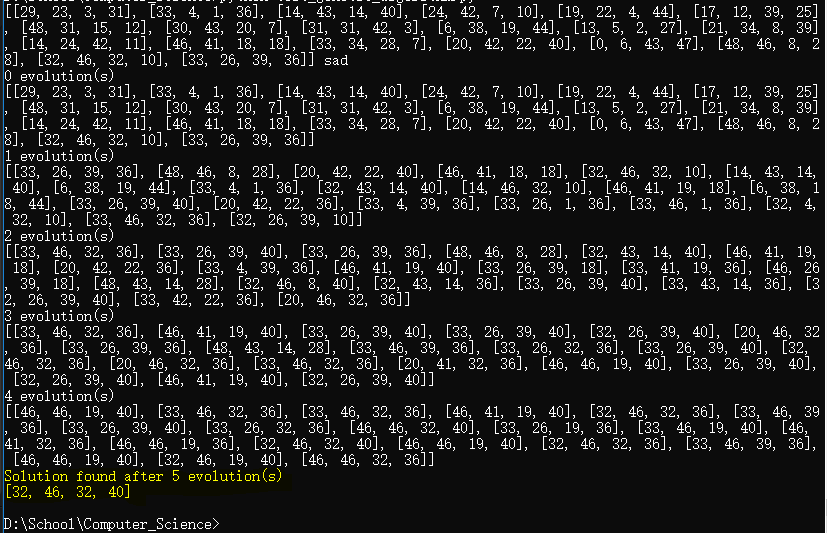
**print**(count,'evolution(s)')

**print**(testPop)

testPop = evolution(testPop,target)

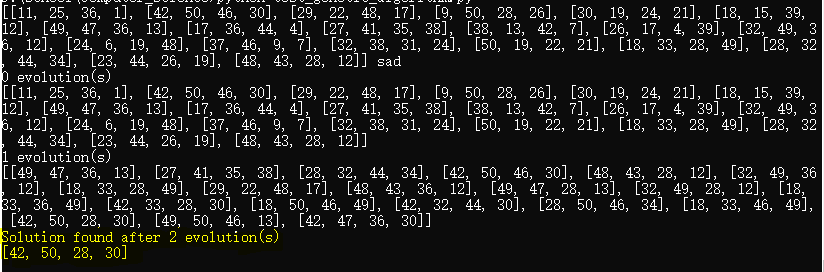
**break**

… Successfully finds the solution to the problem in varying nuber of evolutions.



**Figure 38: Genetic Algorithm - Result 1**

The above result shows the Genetic Algorithm finding the correct solution in 5 evolutions. The sum of 32, 46, 32 and 40 is indeed 150. The best result I had was the solution found after only 2 evolutions.



**Figure 39: Genetic Algorithm - Result 2**

### Problem-Specific Genetic Algorithm

The general idea of the generic genetic algorithm can be used in building our problem specific one. Some changes need to be made to the way it defines a population, evolution, fitness function and the usage code.

After using the Genetic Algorithm to find the coefficients, we can then use those coefficients to determine the record for next year.

**def** runGA(target, conference, cNum, tNum, year ):

size = 12

# not 4

test = population(size ,4,75,125)

repeat = True

count = 0

**while** repeat == True:

count = count + 1

test = evolution(test, target, conference, cNum, tNum, year)

solution = test

**print**(count, 'evolution(s)')

**if** len(solution) != size:

**return** solution

repeat = False

**break**

''' If the returning value is a solution, the len of the 1 d list will be Four

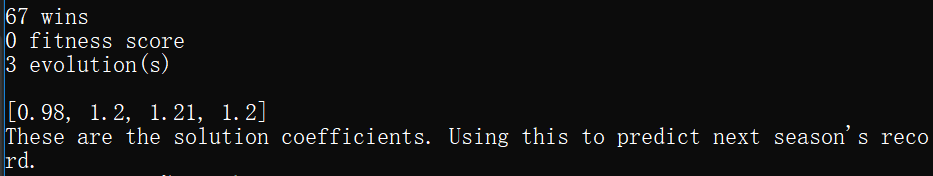
this is not 'size' so we know that what was returned was the solution

if returning value is the new population for next evolution, the length

will be 6 and the same as size so we know this is the pop for next evolution

so these functions are not executed '''

**print**( runGA(67, 'west', 1, 0, 17) )



**Figure 40: Genetic Algorithm - Result 3**

**Figure 40** shows the result when the function is run to predict the record in 2018 using the record from 2017 as fitness function. It successfully converges to a solution and finds the coefficients after three evolutions.

### Genetic Algorithm Results Table

|  |  |  |
| --- | --- | --- |
| **Task** | **Description** | **Result** |
| Generic Genetic Algorithm | This algorithm should find four numbers that sums to 150 | ✔ |
| Specific Genetic Algorithm | This algorithm should find the four coefficients that correctly predict the test record | ✔ |

## Prediction

After the genetic algorithm finds the four coefficients using the previous win-loss as the fitness assessment, the coefficient should be applied to data of the test year to predict that record.

**if** co == 'west':

w = 0

**for** i in range(5):

w = w + simulateWest(west[nums[0]][nums[1]], nums[0], nums[1], year +1, sol )

w = w // 5

# because of the randomness, we can take an average

**elif** co == 'east':

w = 0

**for** i in range(5):

w = w + simulateEast(east[nums[0]][nums[1]], nums[0], nums[1], year +1, sol )

w = w // 5

**print**(t, 'is predicted to win', w, 'games in 20' + str(year + 1), 'Their record:', w , '-' , 82 - w)



**Figure 41: Genetic Algorithm - Result 4**

Using the solutions generated (**Figure 40**), the algorithm predicted that Golden State Warriors will win 58 games. This is a very good prediction as Golden State did actually win 58 games that year

Small pop size somewhat relies on the initialization of individuals to be close to the solution