2018

Sports Analytics With Computer Vision

Colby T. Jeffries
The College of Wooster, cjeffries18@wooster.edu

Sports Analytics With Computer Vision

Independent Study Thesis

Presented in Partial Fulfillment of the
Requirements for the Degree Bachelor of Arts in
the Department of Mathematics and Computer
Science at The College of Wooster

by
Colby Jeffries

The College of Wooster
2018

Advised by:

Dr. R. Drew Pasteur

Dr. Sofia Visa

Abstract

Computer vision in sports analytics is a relatively new development. With multi-million dollar systems like STATS's SportVu, professional basketball teams are able to collect extremely fine-detailed data better than ever before. This concept can be scaled down to provide similar statistics collection to college and high school basketball teams. Here we investigate the creation of such a system using open-source technologies and less expensive hardware. In addition, using a similar technology, we examine basketball free throws to see whether a shooter's form has a specific relationship to a shot's outcome. A system that learns this relationship could be used to provide feedback on a player's shooting form.

Acknowledgements

First and foremost, I would like to thank my advisors, Dr. R. Drew Pasteur and Dr. Sofia Visa. Without their help, I would have been unable to discover and learn what I did. Their advice and guidance helped make all of this possible. I would also like to thank the other members of the College of Wooster faculty and staff that provided advice and technical help: Prof. Nathan Sommer, Dr. Denise Byrnes and Coach P. J. Kavanagh. I would also like to thank Scott Stoudt for his help in generating test data. In addition I would like to thank both the Computer Science and Mathematics departments at the College of Wooster that provided financial support as well as the College at large for the educational opportunities it afforded me, including this project.

I would also like to thank those whose support made this project possible: my mother, Lisa, my fiancée, Katherine, and the rest of my family that supported me through this endeavor. Finally, I would also like to thank my friends at the College of Wooster who provided great feedback and ideas: Scott, Thomas, Tommy, Max, Nick, Todd, Phillip, Gianni, Joe, Dylan, along with many others.

Contents

List of Figures

Chapter 1

Introduction

In sports analytics, data collection is one of the most important pieces of the process, as there would be nothing to analyze without data. Historically, data collection has been simple, usually only consisting of discrete measurements (like how many points were scored by a player). In recent times, data collection has gotten more advanced as coaches and teams have started to appreciate the advantage that good numbers can grant them. However, collecting more advanced and accurate statistics is increasingly labor intensive and difficult. This is where computer vision comes into play.

With recent advances in computer vision algorithms and computational power, using computers to collect statistics has become easier and more effective than ever before. State-of-the-art, feature-complete systems exist, but remain extremely expensive. Thus, these systems are only available to professional teams and only the best college programs. This is where our project comes into play. We aim to create a basketball data collection system that is more accessible to smaller college teams and even high schools. We want our system to be cheap, easy to setup and use, and effective. Along with a data collection system, we also aim to create a narrower system to analyze free throws. This would give insight into whether or not a player's form has a specific impact on the outcome of their shot.

1.1 Problem Outline

This project contains two problems: that of our data collection system and of our shot analysis system. We first outline the data collection system problem. This problem consists of the following objectives:

- Collect the positions of players over the course of the game.
- Collect the position of the ball over the course of the game.
- Compile this data into a usable format.
- Generate statistics reports from this data.
- In meeting these objectives, the created system must be relatively inexpensive and easy to use.

The shot analysis problem is comparatively simpler, consisting of the following objectives:

- Collect the detailed positioning of the shooters body over time.
- Compile the data from a shot into a usable format.
- Analyze multiple shots from the same player and determine if there is a relationship between shot form and shot outcome.

- In meeting these objectives, the created system must be relatively inexpensive and easy to use.

In order to build these systems, we must meet these objectives in some capacity. These objectives are specifically left vague, as there are many ways to accomplish them.

1.2 Goals

In this section we will explain our goals for the project and how these goals would be sufficient to fulfill our objectives.

1.2.1 Data Collection System

The overall goal of our system is to help automate statistics collection and collect temporal data that cannot be collected through other methods. The primary purpose of collecting this data is the same as most other sports analytics projects: aid coaches and players in strategic decision making. This is a broad goal, and does not really capture the specifics of what we plan to do, thus it is wise to break our goals down a bit.

Our data collection system will consist of a physical camera system along with a software system that together will complete our objectives. The camera system will collect the images needed to derive the positions of the ball and players, where the software system will manage the cameras, derive and compile the data, and generate the statistics.

Our goal is to build a camera system that meets the following requirements:

- Within our budget of $1000. This budget is reasonable for even a modest high school facility, so staying under this budget would make this system extremely accessible.
- Easy to set up and use. The end users of this system will not be computer scientists or engineers, so it must be usable by coaching staff.
- Capable of collecting the footage needed for 3D data on the ball position and 2D data on the players' positions. This is necessary to meet our objectives. We are looking for 2D player positions as they are all that is necessary for virtually all important statistics, and much easier to collect.

To meet these requirements we plan to use multiple cameras, set up in strategic positions to cover the needed angles. These cameras will need to meet the requirements of the system itself. We will discuss how we went about building this system in a later chapter.

Our goal on the software side is to build a program meeting the following requirements:

- Capable of recording an entire game with minimal user interaction.
- Capable of analyzing recorded games to generate the 2D player position data and 3D ball position data. There will be an unavoidable amount of user interaction necessary, but we aim to minimize it.
- Capable of generating statistics reports from the collected data.

- Easy to use. This is key for any interface that is created.

To meet these requirements we plan to build a three part software suite, with separate interfaces for recording, analysis, and report generating. This software will contain all of the needed computer vision elements, and will be closely integrated with the camera system. We will discuss how we went about building the software system in a later chapter.

1.2.2 Shot Analysis System

The shot analysis system is entirely made up of a software system. This system will be capable of using any camera, with better camera yielding better results. The system has the following requirements:

- Capable of capturing the body position of the shooter for the duration of the shot and saving it.
- Capable of taking multiple saved shots and analyze them to determine the relationship between form and shot outcome.

To meet these requirements we plan to build a system using existing software to estimate body positions from images. We hope to be able to ascertain whether or not shot form has a clear relationship with shot outcome using a clustering algorithm or some other machine learning tool. If such a relationship exists, we aim to use it to provide feedback to a shooter to help improve their shot.

Chapter 2

Background: Sports Analytics

A sport, at its heart, is numbers. Points, goals, time, yards: measurable values. Some sports are more numeric than others: baseball generates an enormous amount of information where soccer really only keeps track of time and goals. In general, a sport is all about numbers, and where there are numbers, there are people there to analyze them.

The sports piece of "Sports Analytics" is pretty obvious, but what exactly does analytics mean? Analytics comes from the business world, and refers to a powerful tool set that includes advanced statistics, data science, machine learning, and other related methods. In business, these tools are used to gain a competitive advantage over other businesses. This objective clearly translates to the sports world as well [12].

Sports analytics and statistics have had a huge boom in popularity and utility in the last few decades due to increases in computational power and advances in data collection. Teams that embrace sports analytics, like the early 2000s Oakland A's under Billy Beane, the late 2000s Tampa Bay Rays under

Stuart Sternberg, and the current Houston Rockets under Daryl Morey, have been wildly successful, and have caused sports analytics to be important piece of virtually every teams' toolbox [12] [26]. In this section, we briefly describe the history of sports analytics at large, sports analytics in basketball, along with the future direction of the field.

2.1 Origins

Every modern day sport started in roughly the same place in terms of recording data and analyzing it: discrete data was collected (sometimes very little, in the case of soccer, or quite a lot, like in the case of baseball), but analyzed and manipulated very little. This data consists of information that someone watching the game could record, like points scored, errors in baseball, yards in football, and other things of that nature. As these sports developed, their statistics got more complicated. These advances usually corresponded with improvements in computers [32] [34] [15]. We are going to discuss a brief history of statistics in a few different sports with active statistics and analytics communities: baseball, American football, and basketball.

Baseball is likely the sport that is most known for its statistics and analysis. Due to the "box score", invented by sportswriter Henry Chadwick, spectators could get an idea of what happened in the game through statistics. This box score contained all of the events of game and all of the statistics attributable to each player. Based on counts from game box scores, summary statistics for individuals in batting, fielding, and pitching were developed. By the the end of the 1800s, most of the standard statistics we know today had been developed [15].

However, despite the term statistics applied to these recorded values, there are not really statistics at all, they are recorded numbers that involve little to no manipulation. It was not until the 1950s that advanced statistics started being applied to the game. These early forays into baseball statistics were investigations into applying statistical models to optimal strategies and player performance. These areas continue to be the hottest areas of research in baseball analysis [15].

With the improvements of computers in the 1960s, baseball had an explosion of new statistics and analysis. New statistics, new applications for old statistics, and large readily available data sets brought an unprecedented level of interest in baseball and baseball statistics. Up to the 1990s, these statistics became more advanced, and the collected data more detailed [15]. This led to the biggest revolution in sports statistics and sports analytics: Moneyball, which we discuss in the next section.

American Football generates has a lot of discrete data, and records of this type of data goes back to the 1930s for professional leagues and even earlier at the college level. This data remained unused for advanced analysis until the 1970s when the first paper detailing the usage of statistical reasoning for football was published. However, football has largely remained outside of academic research. The reason for this is left up to speculation, but it seems to be a combination of the nature of the game and the large gambling market around football. There is not good way to format play-by-play information and thus there is no good way to analyze it. On top of that, football has such a huge variety of possible situations due to the complexity of the game. This means that analysis becomes significantly more complicated. The gambling market does not prevent research from occurring, but rather causes much of the research related to the gambling market to be proprietary, and thus unavailable to the public [32].

Basketball is fairly new onto the scene with complicated statistics and academic research. The NBA has stats from back around the time of its origins in the 1940s [2]. However, due to the fact that basketball was less popular than football and baseball for much of the 1900s, basketball was not researched nearly as much. However, a resurgence in popularity in the past few decades has led to a significant increase in research. Much of the research in the past dealt with modeling shooting and the NCAA tournament [34]. Modern research still investigates these areas but is also now looking at many more facets of the game, as we will discuss in a later section.

There is an important thing to notice when looking at these three examples. Baseball had much more complicated statistics, much earlier. The reason for this is rooted in the the way these sports work and the technology available. Basketball and football are examples of complex invasion team sports. Essentially, these sports see the team attacking their opponents territory to score, and defending their own territory. These sports do not yield themselves well to the generation of linear discrete data like baseball does. Because of this, powerful technology (which we will discuss later) is needed to track players' movements [20].

2.2 The Moneyball Revolution

The Moneyball story is the subject of a movie, book, and numerous articles; it is a household name. Moneyball is the story of the Oakland Athletics in the late 1990s and early 2000s, and how they, one of the poorest teams in the MLB, performed as well as the richest teams. The Athletics made the most of their small budget, but how? How did a team with one of the smallest budgets, in a game where a large budget is an enormous advantage, perform so well? The answer is simple: they re-evaluated the way a baseball team is managed and how players are selected [28]. As mentioned above, an entire book was written on this story, so we will only give a brief overview.

The primary subject of this story, aside from the Oakland A's, is the A's general manager Billy Beane. Beane essentially proved that the collective wisdom of the last century of baseball was wrong, and that it has not kept up with the changes of the game. Up until this point in the history of baseball, scouts were largely responsible for determining the value of a player, and reporting to management who to attempt to draft, trade for, etc. Beane looked into statistical alternatives to determine a players value. The Athletics attempted to find statistical indicators of player performance, especially ones that no one else used. This allowed them to draft players than other teams overlooked, and get them for a reasonable price. This, in turn, allowed them to stretch the small budget further, and compete with times with double the amount of money [28].

Beane and the A's discovered a few things through statistics and analytics that flew in the face of traditional baseball wisdom: that college players with more experience are better draft picks than highly talented prodigy high school players, that instead of tradition measures of offensive capability (like batting average) statistics like on-base and slugging percentages were much more indicative of players' offensive capability, along with other things. Essentially, the A's were redefining what numbers made a player valuable. Initially, the baseball community dismissed the A's, until they went to the playoffs in 2002 and 2003. This was initially described as an aberration, a fluke. However, the numbers backed the A's up, and soon, other teams started to adopt the A's strategies [28].

Once other teams started copying the A's strategies, the edge the A's had slowly eroded. Over time, baseball broadly adopted Beane's methods and integrated it with their current methodologies. Now both statistics and scouts were used to evaluate players. Most front offices now have statisticians to look at the stats, and scouts to look at things that statistics cannot capture like a players maturity and personality [28].

The A's used statistics to do the job of what was previously human judgment. They discovered previously unknown relationships that proved useful. This was far more complicated then anything that had been done in relation to sports outside of academia. From this point forward, statistics became a huge part of baseball. However, the effect of the A's success was not limited to baseball. Interest in more complicated statistics spread to other sports, including basketball, where data collection and analysis was taken even higher.

2.3 Basketball and SportVu

Sports analytics in basketball has seen a rise since Moneyball. Until recently, scouts evaluated basketball players by watching games. They determined a players value through instinct. Nowadays, statistics are starting to heavily influence strategy and player evaluation. The most obvious result of this is the shift toward three-point shots. The numbers show that if shooting from more than a few feet away, the shot might as well be a three-pointer in order to maximize points. Teams that embrace the three-point shot have consistently outperformed those who have not. This is become even more abundantly clear with the recent domination of the Golden State Warriors, who are consistently one of the best three-point shooting teams in the league and have been practically unbeatable for the past three years.

An even better example is the current Houston Rockets. They have been one of the best teams in the league for the past few years, due many statistics driven changes they have made. Like the Warriors, the Rockets have heavily favored the three-pointer, so much so that they have attempt, on average, more three-pointers than two-pointers. Despite this, they remain competitive (in the top ten) in three-pointer field goal percentage, meaning this strategy is paying dividends. Also contributing to their success are strategies at player level: James Harden, a favorite for MVP this year, takes a higher proportion of shots for his team than any other player in the NBA. Generally, it is expected that the success rate of strategies decrease with frequency of use, but as of right now he seems to defy this rule, so much so that he is practically always guarded by two opposing players [2]. The strategies that the Rockets have been employing rely heavily on their usage of available statistics and their high-end analytics. All of these strategies employed by the Rockets are driven by their Ivy League educated analytics-savvy general manager Daryl Morey.

Player evaluation has also changed with statistics. Statistics like +/-, player impact, net rating, and effective field goal providing new insight into how a player performs [2]. These statistics show the value of versatile players like Draymond Green that previously would be underrated. These numbers also validate the strength of well known superstars like LeBron James, while simultaneously giving a more accurate pictures of high scoring players that lack defensively who were previously overrated.

All of these changes are the result of better statistics and a recent innovation in sports analytics. This innovation is likely the biggest change in the history of sports analytics: optical tracking. Optical tracking allows a huge amount of new, finely detailed data to be collected, causing an explosion of data and new analytic possibilities. The STATS company is responsible for the largest and most popular optical tracking statistics systems. Their basketball system, SportVu, is the most well known and complete system of its type. Every team in the NBA has the SportVu system implemented in their facilities to collect information from every game [7].

The technical details of SportVu are sparse, seemingly to protect the system from competition, however, some information is available. SportVu uses six cameras (Figure 2.1), and is capable of tracking the (x, y) positions of the players along with the (x, y, z) position of the ball 25 times every second. This data is combined with data from the scorers table, including game events like fouls and turnovers, the game and shot clock, the current score, and other

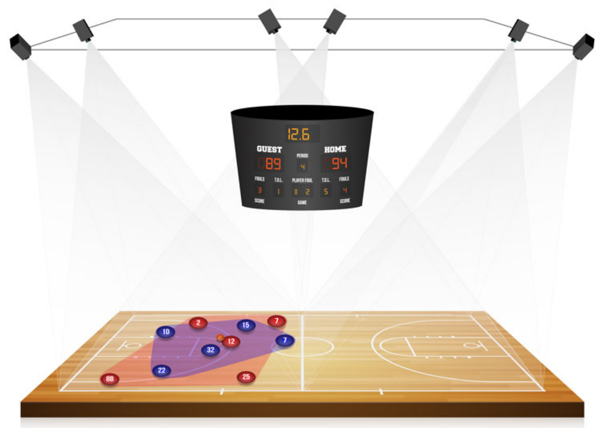


Figure 2.1: A depiction of how the SportVu cameras are laid out [7].

game elements. The exact method that this data is captured is not specified. This raw data is used to calculate numerous statistics with the aim of improving team performance and providing more information to coaches to improve decision making [7].

According to the NBA, SportVu automatically collects a huge amount of statistics. These stats include various pieces about shooting location and efficacy, player ball time and possession, defensive stats like blocks and steals, along with other more complicated stats like player impact [2]. These stats are combined into other hybrid player effectiveness stats as well. This information is provided to teams as reports and data visualizations [7].

SportVu was not originally in every NBA arena. Initially, it was used by a few teams starting in 2010, who made great use of the system and payed for it themselves. Over time, more teams adopted the system, until Adam Silver, the NBA commissioner, eventually mandated that all teams implement the system in 2013 to get better statistics throughout the league. Because of this, we now

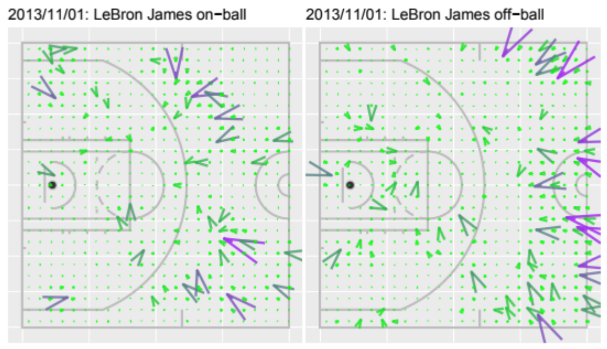


Figure 2.2: Chart of the "smoothed empirical acceleration vectors" of LeBron James during the Heat vs. Nets game [36].

get a complete picture of every game, allowing the league to generate accurate and complete full season statistics [7].

Aside from STATS internal work on the SportVu system and the NBA teams stats collection, little work has been done with SportVu data. There is a sample SportVu data set available from a game on November 1st, 2013, between the Miami Heat and the Brooklyn Nets. Essentially, this data set contains raw SportVu data. Each row contains the time, the quarter, the shot clock, the game clock, the (x, y, z) of the ball, the (x, y) of every player on both teams, the number of the current possession, and an event ID for each player that indicates if an event happened involving that player. Each row is collected 25 times a second, resulting in almost a million rows from a single game. Steven Wu and Luke Bornn explored this data in an article of theirs, and showed what can be done with it. They specifically looked at modeling the movement of players on offense, and were able to come up with a few interesting statistics and visualizations (Figure 2.2) [36].

From this, it should be easy to see where our project fits in. We aim to build a system that has some of the capabilities of SportVu, but without the price tag. We want this technology to be available to colleges and high schools, not just the NBA.

2.4 Moving to Other Sports

High end sports analytics is moving to other sports as well. Baseball, post Moneyball, does not have much more that can be done. Due to the linear nature of the game, we are able to generate a complete picture of the game. Other faster pace games like basketball are starting to move toward systems like SportVu. Soccer, hockey, American football, and a few other sports are have systems from STATS akin to SportVu [6]. These other systems, will allow the collection of much more detailed data. Other sports present an increased challenge over baseball and basketball. American football, soccer, hockey, and other fast sports have many things happening at the same time, and a lot of important pieces of the game occur "away from the ball". New and novel statistics will have to be created to encapsulate everything that happens and evaluate players at a similar efficiency as the statistics that have been found for basketball and baseball. All of these sports are active areas of research, and there is a lot of money involved with these sports, so it is likely we will see advances soon, spurred on by the success of SportVu and the importance of sports analytics in basketball and baseball.

Chapter 3

Background: Computer Science

In this section, we give background on the areas of computer science that this project deals with. It is important to understand the previous research and advances in these fields that make our goals possible.

3.1 Machine Learning

Machine learning is the subfield of artificial intelligence that deals with constructing computer programs and algorithms that improve themselves. These programs "learn" through experience. The exact process varies, as these algorithms are diverse and rely on a large variety of concepts from math and computer science [30]. In this section we give a brief overview of machine learning, including some history and modern techniques.

3.1.1 History

In this section we go through some of the important milestones in machine learning, and their importance. The history of machine learning is not well documented, so creating a comprehensive history is difficult.

The first program that could be considered part of machine learning was written by Arthur Samuel in 1952. This program played checkers, and improved itself by studying the moves that previously resulted in victories [29]. This program demonstrated that self-improving programs are possible, and that they are effective.

In 1957, the first neural network was designed by Frank Rosenblatt. This machine, known as a perceptron, simulated the thought process of the human brain [29]. This machine specifically simulates the neuron, and is quite simple. However, when multiple perceptrons are combined together, they form a neural network, a much more complicated machine learning technique that is still used today [30].

In 1967, the nearest neighbor algorithm was written [29]. This algorithm amounted to basic pattern recognition, and is foundational in many computer vision algorithms. This algorithm also played a part in the creation of clustering and instance based learning algorithms [30].

In 1981, Gerald DeJong introduced Explanation Based Learning. This learning mechanism involves the analysis of training data and the creation rules to discard unimportant data [29]. The concept has been extended to numerous algorithms and techniques [30].

The next major point in machine learning is not a single discovery, but rather a sort of shift in paradigm of the field. Starting in the 1990s, scientists started to focus on data driven approaches over knowledge based approaches. Essentially, scientists started building programs to analyze large data sets in hopes of learning from the data [29]. The machine learning field has continued in this direction, and most modern approaches to machine learning are data driven.

The late 1990s and 2000s saw more major improvements in machine learning, and further milestones in performance. The 1990s saw IBM's Deep Blue beat a world champion at chess. In 2011, IBM's Watson won a game of Jeopardy. In the last decade, we have seen complicated machine learning algorithms out of Google, Facebook, and Microsoft that have tackled complicated problems. Google's Brain learned to categorize objects in the same way a cat does. Facebook's DeepFace has been able to recognize individuals in photos at roughly the same level that a human can. In 2016, one of Google's machine learning systems, AlphaGo, beat a professional at Go, one of the worlds most complicated board games [29]. As we can see, machine learning has accomplished a lot and worked towards solving numerous complicated and difficult problems. In all likelihood, machine learning will continue solving more difficult problems as computers and algorithms improve.

3.1.2 Modern Techniques

Modern machine learning systems use a wide variety of algorithms and methods, and serve an even wider array of purposes. Instead of attempting to cover all of the modern machine learning techniques, we will briefly describe two popular ones along with some of their applications.

Neural networks are one of the most popular and most famous machine learning techniques. Neural networks are based on a simplified model of an organic brain. These networks are composed of individual “units” that take in multiple real-valued inputs and output a single real-valued output. These “units” are essentially perceptrons that evaluate a weighted linear combination of its inputs with an activation function that yields some sort of value, usually boolean. These units are arranged into layers that feed their outputs to each other. Ultimately, a final output row that dictates the outcome of feeding the data to the neural network, usually this is a classification. The actual layout of the network, activation functions used, method to update the weights, and connections between the layers vary. Variants of neural networks exist that are better suited to images, recurrent data, and other data types. Designing a neural network for a problem is usually the most difficult part of using a neural network [30].

Neural networks have been applied in many ways as they are suitable for all sorts of situations. Suitable problems for a neural network have the following qualities:

- Individual data points are represented by many attributes.
- The target function output, that is the desired output from the network, can be discrete, continuous, or vector of these.
- The training data is not error free.
- Long training times are acceptable. Training a neural network is

computationally expensive.

- Evaluation of the neural network on a new data point needs to be fast. Despite the long training time, evaluation is very fast.
- Humans do not need to understand how the network works. Interpreting the weights of a network is extremely difficult.

As mentioned earlier, many problems have these qualities [30]. Applications include self-driving car systems, facial recognition systems, Google's Brain (and other classification and recognition systems), financial trading systems, and other applications with demographics, bioinformatics, logistics, and may other fields [30] [29] [14].

Another popular class of methods in machine learning is genetic algorithms. Genetic algorithms provide a method to search a hypothesis space in hopes of optimizing some fitness function. Genetic algorithms are conceptually based on the evolution process from biology. Genetic algorithms can vary in details, but nearly all share the same basic structure. The algorithms basic operation involves updating a pool of possible hypotheses called a population. With each new iteration, all population members are evaluated with the fitness function. A new population is generated by carrying forward a portion of the population based on probabilities generated from the the population fitness. Some of these individuals are also combined to generate new members. An element of mutation is also added to ensure that the algorithms does not become stagnant. The exact details of these parts vary [30]. Essentially, the algorithm keeps hypotheses that perform well, uses them to generate new hypotheses, and introduces random mutations to spur changes and preserve the diversity of population.

The attributes of a hypothesis are the “genes” in a genetic algorithm. Hypotheses are made up of a set of rules (genes) that are encoded into a sequence. These sequences are usually binary, as it ensures that combining hypotheses and mutating them is easy. This concept can be extended to quite large search spaces and complicated genetic sequences. For more information on genetic algorithms see [30].

These algorithms can be applied to a large variety of problems. These algorithms lend themselves well to problems that have well defined outcomes that make good fitness functions. Goals like achieving a maximum distance or score work really well. Thus strategies for various games can be found using genetic algorithms. Another interesting application for genetic algorithms is the tuning of other machine learning algorithms. The parameters of a machine learning algorithm can be seen as a hypothesis, and the accuracy of the algorithm the fitness function. This allows machine learning algorithms to have their parameters optimized. Genetic algorithms also have application in wind turbine placement, power grid balancing, and many other areas [27] [30].

3.2 Computer Vision

The field of computer vision has an apt title. Computer vision can be considered the “construction of explicit, meaningful descriptions of physical objects from images,” [13]. Conceptually, this seems simple, as it is something humans can do intuitively. We perceive the world by extracting information from what we see and combining it with our previous knowledge. The same concepts can be applied to computers, but is much more difficult [13]. In this section we lay out some basic history and some of the modern techniques in computer vision.

3.2.1 History

Computer vision experiments were first conducted in the 1950s, with most of the core concepts being solidified in the late 1970s. Many computer vision techniques were based off of artificial intelligence and machine learning tools, and thus were formed around the same time or slightly after [13].

Computer vision is closely related to the field of image processing, which existed earlier. However, the difference is in goal: computer vision aims to have the computer understand what is contained in an image, where image processing aims to simply recover the 3D structure of the image. Thus, many techniques used in computer vision come from image processing, and extend them [33].

The 1970s saw many advances in the 3D modeling shapes from images, including one of the favorite modern techniques: using generalized cylinders to model parts of objects and arrange them hierarchically. In this period, a lot of the foundations of computer vision were solidified [33].

In the 1980s, a lot of research was directed toward more sophisticated mathematical methods to analyze images and scenes. Advances during this time included methods like image pyramids for image blending and coarse-to-fine correspondence search. Along with image pyramids, new techniques to determine shape, like shape from shading, shape from texture, and shape from focus were created. Other areas that saw improvement were in edge and contour detection, and 3D models [33].

The 1990s saw improvements in many of the previously mentioned areas, along with an explosion of interest in modeling motion, image segementation, and computer graphics. This interest in computer graphics is most important as it led to new techniques to manipulate real-world imagery directly to create animations and models for graphics applications, along with the ability to properly visualize many facets of computer vision. This included the automatic creation of realistic 3D models from collections of images [33].

From the early 2000s to today, we have seen further mixing between computer vision and computer graphics. This includes topics like image stitching, new rendering techniques, and improvements in digital photography. Currently, point based features dominate various algorithms, contour and region based algorithms are starting to gain steam. The largest trend in the last two decades that has dominated the field is the incorporation of machine learning techniques. Because of the immense amount of labeled data, we are seeing machine learning algorithms (often neural networks) applied to image recognition problems [33]. The most recent and popular application of computer vision that has become part of mainstream knowledge is the autopilot feature in Tesla's electric cars. These cars use a combination of computer vision and other technologies to actively drive themselves in highway situations.

3.2.2 Modern Techniques

Modern techniques in computer vision are varied and complicated. Instead of outlining specific algorithms/techniques, as there are a huge number of them, we discuss two classes of algorithms used widely in computer vision applications.

One of the most commonly used parts of computer vision is feature detection and matching. Algorithms in this area aim to find unique features in images (whether they are points, edges, regions, or another feature type) and match them with other features in other images. This is done in a huge variety of ways, but typically, these algorithms follow the same pattern: first features are generated for all of the images involved. These features are then searched for in subsequent (or other) images. These algorithms also make use of machine learning, which makes them much more powerful. Feature detection and matching is useful in many scenarios including shape detection, photo manipulation, and animations [33]. These algorithms are also the basis of many more complicated algorithms in computer vision, including those discussed in the next paragraph.

Recognition is the term used to describe algorithms and systems that aim to describe what an image contains. These algorithms detect objects, faces, people, and other details in an image. As well as detection, these algorithms aim to recognize these objects, that is distinguish them as separate entities. This could be as simple as identifying a car as a car, or as complicated as determining the identity of a person by looking at their face. These algorithms represent the cutting edge of computer vision, and the area that most work is currently being invested. Once again, this is an area where machine learning is heavily used. Neural networks, particularly convolutional neural networks, are very popular [33].

Applications of recognition algorithms are practically endless. With the massive amounts of training data be collected/generated in annotated and labeled databases, systems can be trained to recognize objects ranging from cars on highways to wanted criminals in security camera footage. These algorithms have also been used to help intelligently edit photos. This includes filling in missing pixels from an image using other images of the same (or similar) scene [33]. The future of this area of computer vision is promising, as computational power increases and available datasets get large, we should see computers able to effectively recognize just about anything.

Chapter 4

Optical Statistics Collection: Methodology

In this section the various methods used (or planned) to collect basketball statistics optically will be detailed. Due to issues stemming from some of these methodology choices, many parts of the planned system were never implemented. We will discuss the original plan for those systems, along with possible issues. The various parts of the methodology include:

- The camera system and its operation.
- The computer vision techniques used to manipulate the video feeds, track the players and ball, and collect data.
- The software built to process the video feeds and collect data.

4.1 Camera System

There are two common options for collecting the type of spacial and temporal data that we need from a complex scenario like a basketball game. The first involves using physical trackers like RFID (Radio Frequency Identification Devices) tags or Bluetooth beacons. These trackers have varying ranges, accuracies, and pinging frequencies. We found in our research that an effective system based on this technology would be prohibitively expensive. Systems within our budget of $1000 lacked both range and pinging frequency. Another notable issue is that these systems would require setup of receivers close to the court, which could interfere with play. These systems would also require physical devices attached to the players and ball, which would be problematic in official games and could also interfere with play.

The second approach involves using cameras and computer vision techniques to track the players and ball. Because the cameras are mounted so far away from the court, they cannot interfere with the game, and would be permissible in official games. Multiple cameras are required to create a clear picture of the game, and camera systems are much cheaper than their RFID counterparts. However, as we discovered in attempting to build a camera system, cheaper consumer cameras are not sufficient for the task and the programming requirements for such a system are much higher. This ultimately proved to be the source of the majority of the issues we encountered in implementing our system.

In this section, we will describe our original plans for the camera system, the pros and cons of the selected cameras, and why they proved to be insufficient.

4.1.1 Cameras

The cameras needed for our system had a few important requirements:

- Inexpensive: We wanted the cameras to be around $100 so that we could purchase as many as we need, and still remain within our budget.
- Wireless: We needed cameras that could be accessed wirelessly, so that they could be accessed remotely. The cameras were to be mounted on a gym ceiling, so they needed to be able to be activated wirelessly.
- Reliable: Basketball games can last for two or more hours, so our cameras needed to be able to record the entire time.
- Field of View: In order to get a clear picture of the entire basketball court, the cameras needed to have a wide field of view.
- Frame rate: This requirement was a bit less important, but in order for the optical trackers to work effectively and collect good data, the cameras needed to record around or above 10 frames per second. This is a basic minimum value, as higher frame rates will yield better results.

Initially, this list of requirements seemed daunting, as most commercially available cameras that meet all of these requirements are quite expensive. Ultimately, we found that security cameras provided the best feature set for the price. The purpose of security cameras is to run around the clock, and record areas that have the possibility of crime. This means that the cameras must be reliable, and have a decently high field of view. These cameras are also typically located high up, on the sides of buildings or on ceilings, meaning they are typically wirelessly accessible. Most security camera applications require multiple cameras, so they are also typically on the cheaper side. The one requirement that security cameras typically do not fair so well on is frame rate. Because the footage from these cameras need to record so much footage and this footage is simply used for identifying people, the frame rate does not need to be very high, or consistent. We deemed the lower frame rate to be acceptable, and that we could solve the inconsistency problem with synchronization. This turned out to be the first of many issues that we could not quite overcome.

We settled on a \$100 security camera sold on Amazon, the IP3M-943B, manufactured by Amcrest (Figure 4.1). The camera has a resolution of 1080p, with a "ultra wide" field of view, meaning it is about double that of a normal camera. The IP3M-943B records at up to 20 frames per second, more than enough for our application, and is completely wirelessly accessible. It can record to a memory card, or stream to a PC over a network [1].

Despite these positives, there are a few issues with the selected cameras. First and foremost, the camera is only accessible through Amcrest's written drivers. Originally, we did not anticipate this to be a problem, however, this is the root cause of our issues with synchronization and frame rate variability. The frame rate on the cameras fluctuates quite a bit, about ±1 frame per second from the frame rate specified for the camera. This is not a problem for CCTV cameras, so it is, from the perspective of the camera's developer, irrelevant. Also, the camera is rather difficult to set up for our purposes, as its

Figure 4.1: The Amcrest IP3M-943B security camera [1].

power cord is short and it is prone to occasional failure. Fixing these failures requires a reboot, which, to our knowledge, can only be done with physical access to the camera.

We ended up purchasing six of these cameras, to be configured as outlined in the next section. These cameras fell within our budget, but the issues outlined above proved to be the first of many problems that prevented this system from becoming a reality. In a later section, we will discuss possible changes and alternative approaches we could have made to alleviate the issues caused by our camera selection.

4.1.2 Location Configuration

The cameras needed to be positioned in such a way that all parts of the court are covered. Also, the cameras need to be able to pick up on all three dimensions. This could be done in multiple ways. We came up with two possibilities, that would, in theory, capture all dimensions. We were not able to actually set up the cameras to test, for reasons outlined later, so we were unable to prove the efficacy of these setups.

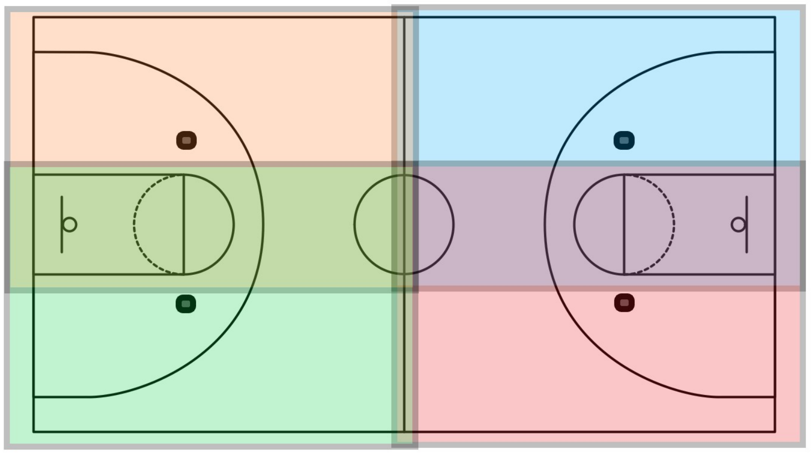


Figure 4.2: A mock-up of locations of the ceiling mounted cameras. The colors roughly represent the fields of view of the different cameras, with clear overlap over the center of court.

The first setup we looked at involved six cameras. Four cameras would be used to collect the (x, y) positions of the players and ball, and the other two would be dedicated to the z position of the ball. The first four cameras would be set up on the ceiling looking straight down at the floor (Figure 4.2). Two cameras would be placed on each side, as to have half of the court in their field of view (Figure 4.3). The view from the cameras on each side would overlap heavily over the key, and all four cameras would overlap slightly at half court. This allows a clear view of the action, and just enough overlap to see when player cross to the other half. The other two cameras would be placed on the wall, about rim height, in opposing corners of the gym. The cameras would face the basket on the opposite side of the gym so that the ball would be visible to at least one camera the entire time. These cameras would simply track the ball, gathering its z position. One of these cameras also would have the scoreboard in its field of view, allowing OCR to be used to capture game information to go along with the spatial information.

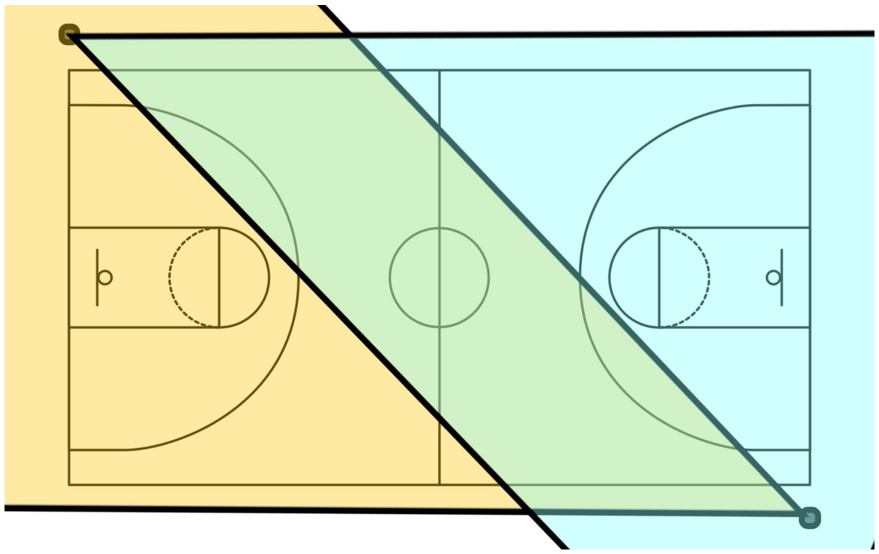


Figure 4.3: A mock-up of locations of the side mounted cameras. The colors roughly represent the fields of view of the different cameras.

The largest issue with this setup is coordinating the trackers between the cameras, as tracking happens on a single video feed at a time. Also, because the cameras are all looking at different areas, the videos have to be perfectly synchronized in order for collected data to be accurate. This is the approach that we planned to move forward with. In further sections, we discuss how we planned on making this setup work.

The second setup we looked at would have been similar to what the SportVu system employs (Figure 2.1). This allows for all three dimensions to be captured, and for a more complete 3D view of the court. We believe that this is the reason that SportVu uses this setup, as it, with the right calculations and enough computational power, would allow for what would essentially be a 3D recreation the game. We deemed this approach to be beyond the time constraints of this project, leading to our choice in the simpler setup.

As mentioned earlier, we were unable to set up our cameras in the gym to test the setup for a few reasons. First and foremost, the gym was not set up to accommodate a camera system of this nature. There are few power sources available for the cameras to use and access to the areas we wish to set up the cameras is limited. In order to actually plug the cameras in, some remodeling would have to be done to run power to the correct places. The second issue was a matter of inopportune timing. A planned remodel of the main gym would end up removing any cameras we set up. In a later section, we will discuss the issues we faced in setting up the cameras further.

4.1.3 Recording and Processing

Recording from the cameras has proven to be quite tricky. The goal here is to record from all cameras at the same time. The desired end result is six video files, each one containing a full games worth of footage from the camera. This seems like a simple task, and if we were recording locally, it would be. However, our cameras are to be installed high up, and thus all footage must be recorded over the local network.

Initially, recording was attempted with Amcrest's Surveillance Pro software (Figure 4.4). This software is designed with the cameras' original

CCTV purpose in mind: continuous recording and motion detection [1]. Thus, some of the features needed for our purposes are not present. We were unable to find a good way to initialize recording for all cameras simultaneously, and sustain the recording for more than an hour without leaving the cameras on perpetually. This software does a good job managing the cameras, but not recording from them.

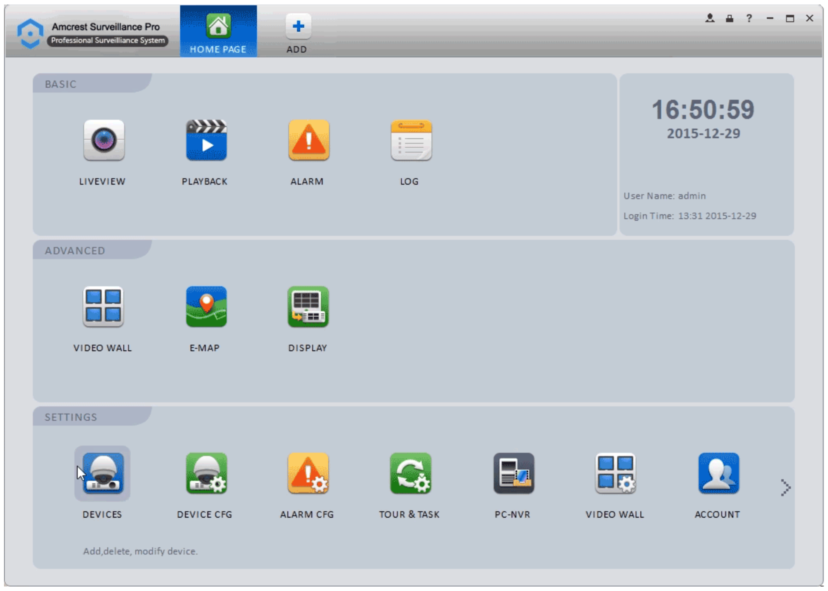
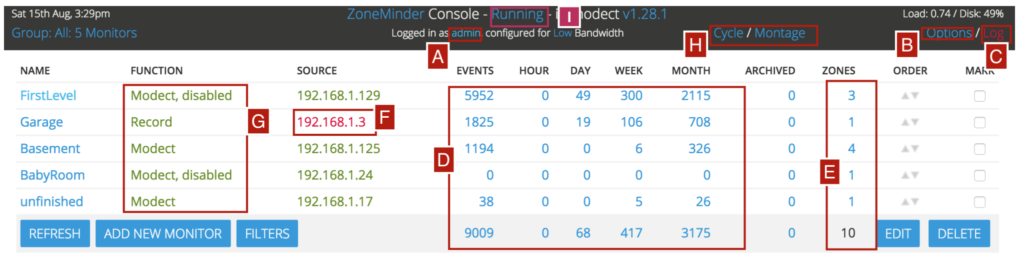


Figure 4.4: The Amcrest Surveillance Pro Software [1].

The second attempt to record was done with a Linux based software called ZoneMinder. ZoneMinder, like the Amcrest software, is a CCTV management system. However, it had a some extra features that made recording easier. ZoneMinder, due to its open source nature, has a larger degree of freedom in setup and customization. We used the cameras' `rtsp` interface to access them in ZoneMinder, and thus were able to access their streams in real time. Most importantly, we were able to record from all cameras, over the network.

ZoneMinder records by taking screen shots multiple times every second. Videos can then be exported as videos from inside ZoneMinder. We were also able to initiate and stop recording on all of cameras at the same time [10].



NAME	FUNCTION	SOURCE	EVENTS	HOUR	DAY	WEEK	MONTH	ARCHIVED	ZONES	ORDER	MARK
FirstLevel	Modect, disabled	192.168.1.129	5952	0	49	300	2115	0	3	▲▼	☐
Garage	Record	192.168.1.3	1825	0	19	106	708	0	1	▲▼	☐
Basement	Modect	192.168.1.125	1194	0	0	6	326	0	4	▲▼	☐
BabyRoom	Modect, disabled	192.168.1.24	0	0	0	0	0	0	1	▲▼	☐
unfinished	Modect	192.168.1.17	38	0	0	5	26	0	1	▲▼	☐
			9009	0	68	417	3175	0	10		

Figure 4.5: The ZoneMinder CCTV software [10].

It may seem that ZoneMinder meets all of our requirements. However, there were a handful of issues. First and foremost of which is that ZoneMinder is complicated to install and use. This is important as this system must be relatively easy to use, as per the goal of the project. Another issue, like with the Amcrest software, is that ZoneMinder is not intended to be used outside of CCTV purposes. The end result is that the recordings have inconsistent frame rates and qualities. Ultimately, ZoneMinder is the backup choice for recording.

Our primary choice in recording is our third attempt at recording: a custom C++ program (Appendices A.1, A.2, A.3). In writing our software, we have a few important things that we need to take into account: network latency, hard drive bandwidth, and video quality. We need to balance these limitations in order to make our software meet expectations. To start with, we use OpenCV to access the cameras, and to save the retrieved frames. This is the bulk of what the program does, however, the issue is that this operation, for multiple cameras, uses a large amount of hard drive bandwidth, and potentially can have issues with network latency. The first step to addressing these issues is to make each camera's recording operation happen on its own thread. This allows the usage of full system resources, and allows I/O to happen on a per camera basis. This is important: without multi-threading the whole program would have to wait every time a frame is being fetched or written. Multi-threading is enough to solve the issues with network latency, however, this is not sufficient to solve our issues with hard drive bandwidth.

The solution to the bandwidth issue turned out to be simple: encoding the video. Instead of simply saving the frames or putting them into an uncompressed video, the frames are compressed to reduce their size before being added to a video. This reduces load on the hard drive enormously and adds a modest amount of work for the CPU. Ultimately, this results in the ability to save full video from all cameras at the desired frame rate.

In order to make the software easy to use, we built a simple interface using Qt5. Qt5 is a multi-language, multi-platform GUI package. It allows the creation of widgets ranging from basic GUI buttons to large custom widgets. The goal of the interface was to make repeat operations a single button press, and for adjustments to the recoding cameras to be simple. The interface consists of entry fields to put camera IP addresses, buttons to increase or decrease the number of cameras being used, a *test* button, and a *record* button. The *test* button runs a quick test to see if the IP addresses entered are for valid cameras, if a cameras is valid its entry field will turn green after the test, if it is not valid it will turn red. The *record* button will record from all entered cameras for two hours. The entered IPs are saved between sessions, so that they remain in the entry fields when the program is run again. This makes

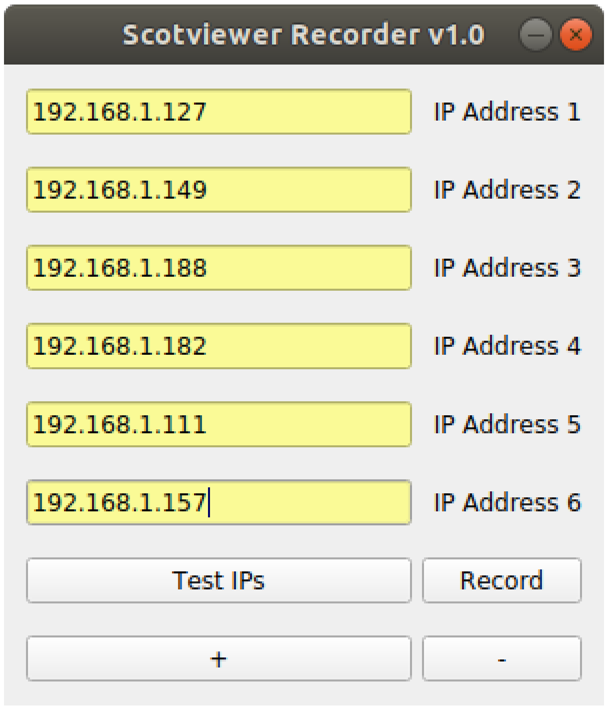


Figure 4.6: The Qt5 interface for our recorder program.

recording a simple task that anyone can do.

4.2 Computer Vision

In terms of computer vision, we are looking to describe the position of the players and ball in a basketball game. However, we are not just looking to determine their position in a single image, but rather in a video stream. This increases the complexity of the problem, as we not only have to recognize the position of the players, but also their change in position over time. In this section, we will describe how we planned to tackle this problem along with the issues we were faced with.

4.2.1 OpenCV

Before we explain the exact methods we planned to use to track the ball and the players, we must first introduce the primary tool set we utilized: OpenCV. OpenCV (Open Source Computer Vision Library) is exactly as it is named. Because it is released under a BSD license, OpenCV is free to be used both commercially and academically. It has interfaces for C++, Python, C, and Java, and is usable on every major operating system [3]. OpenCV has extensive documentation for all of its interfaces, making it easy to use. For these reasons, OpenCV is the best choice for our purposes.

OpenCV was originally built to be a common framework for computer vision applications. It has evolved to be much more than that, it is the go to starting point for just about everything computer vision. Because of its BSD license, it is extremely easy for businesses (or individuals) to use, and gives them the freedom to easily modify the code. Many companies use OpenCV, ranging from big names like Google, Microsoft and IBM to small Silicon Valley start-ups. These companies use OpenCV for a huge variety of applications, from stitching StreetView images together and detecting intruders in CCTV footage, to detection of drowning incidents and interactive art [3].

OpenCV has over 2500 optimized computer vision and machine learning algorithms, ranging from classic to state-of-the-art. It has a huge amount of functionality in a variety of areas including:

- Image Processing,
- Image and Video I/O,

- Camera Calibration,
- Object Detection,
- Machine Learning,
- Image Stitching,
- Facial Recognition,
- Fuzzy Mathematics,
- Object Tracking,
- Text Detection and Recognition,

along with many others [3]. Our system applies many areas of OpenCV; as we explain our planned methodology, we will introduce the OpenCV functionality we use.

4.2.2 Video and Image I/O

The first step in any computer vision task is reading in images or video. Typically, computer vision tasks are done frame by frame, so videos must be decomposed into individual frames. In our case, our video feeds must be synchronized, and read in frame by frame, from the start of the game to the end. This will allow us to track the ball and the players from the beginning of the game to the end. The first step is to synchronize the video feeds. This seemingly simple task turned out to be one of a handful of major issues in this project.

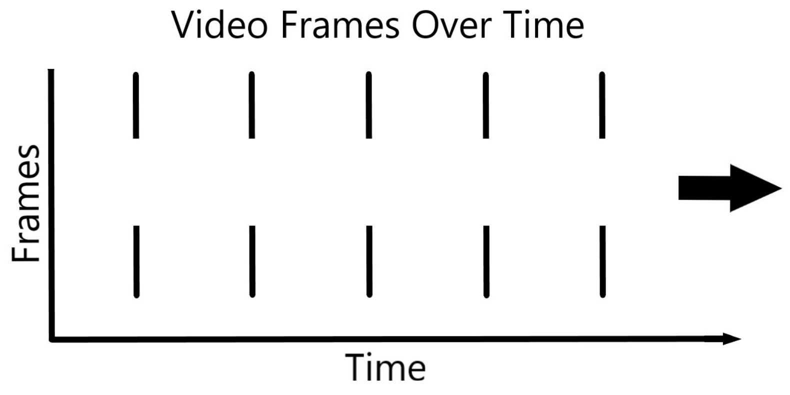


Figure 4.7: A graph of the frames from two cameras, perfectly synchronized. In this situation, the only synchronization required is to find the first frame where all cameras are recording.

On the surface, it may simply seem like the first point in time where all video feeds are recording needs to be found. This would be true if the cameras had perfectly consistent frame rates and the recorded frames lined up perfectly in time (Figure 4.7).

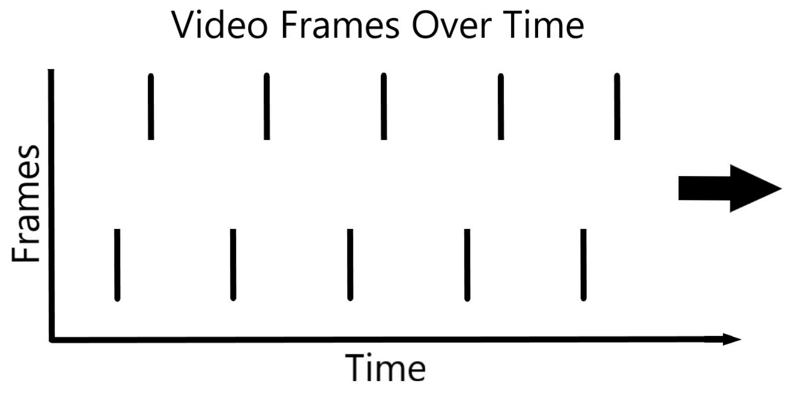


Figure 4.8: A graph of the frames from two cameras, with consistent frame rates, that started at slightly different times. The only synchronization that can be done is the same as in Figure 4.7.

However, the cameras, even when set to record at the same time, do not start at the exact same instant. This leads to the frames being a few milliseconds off from each other (Figure 4.8). This is not a big issue when looking at slow moving objects, but because of the speed that the basketball can move, this would be enough to both throw off the trackers, and add minute inaccuracies to the data. Further compounding the problem, the cameras we selected do not have consistent frame rates (Figure 4.9).

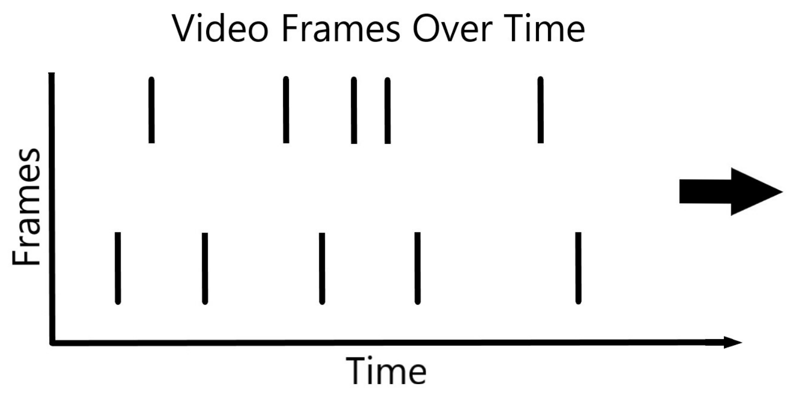


Figure 4.9: A graph of the frames from two cameras, with inconsistent frame rates. Synchronization must be done constantly in this situation to ensure that the frames are always as close together as possible.

We have found that the frame rate actually fluctuates about ±1 frame per second from the advertised frame rate. This means that, not only do we have to compensate for the frames being slightly off from each other, but we actually have to synchronize *constantly* to compensate for the frame rate differences.

Our approach to solve this problem was to synchronize as frequently as possible and ignore the slight differences between frames. In order to synchronize the video feeds, we needed some sort of timestamp for each frame. Our initial thought was to grab a timestamp when the frame was received by the recording program, however, due to the internal software of the camera, this number would be inaccurate. Frames seem to have some sort of processing or buffering in the camera itself before they can be retrieved. This results in timestamps that would be an indeterminate amount of time after the recording of the frame. Our next step was to look at the actual timestamp in the video. Like nearly all other CCTV cameras, our cameras put a time stamp with the date and time in the corner of the video. However, this timestamp is not accessible by any means in file or from the camera, as it just added to the captured frame itself. In order to read the timestamps, we would need optical character recognition.

We will describe the exact methodology we used for OCR in a later section, as we use it for gathering data from the scoreboard. Using OCR, we were able to read the timestamps, and get the time of the frame to the nearest *second* with a fairly high accuracy. Thus, at every new second, we are able to line up the frames as best as possible. This, in theory, should be sufficient. However, due to occasional errors in OCR, and the inconsistency in frame rate, we cannot consider the video feeds reliably synchronized. This is one area that will need further work to make this system possible. In a later chapter, we propose some changes that could potentially solve this problem, or make it irrelevant.

From this point forward, we assume that the frames are synchronized. In order to discuss further proposed methodology, we must assume this problem solved. Once we can be reasonably certain the frames will be synchronized, we can pull an individual frame from each camera to process and collect data from. This leads us to image manipulation.

4.2.3 Image Manipulation

Now that we have a frame from every camera, we must combine the relevant images together to create an overview of the court, and manipulate images into a usable form. The ideal outcome of this step is a 2D top-down view of the entire basketball court. The two side angles would not be stitched or combined in any way as it would be unnecessary.

There are two ways we looked at to approach this problem: image stitching and simple overlaps. The first method, image stitching, would be the theoretical best bet. Image stitching is a computer vision technique used to take multiple overlapping images and generate a seamless image from them (Figure 4.10). These techniques are some of the oldest and most frequently used computer vision methods, and are commonly used in digital camera and cell phones to generate panoramic images [33].

Originally, image stitching was done through simple maximization of pixel-to-pixel similarity. This proved inefficient, and thus modern algorithms moved to extracting features and matching them between the images. This approach is much more robust and faster when implemented properly. This process can feature numerous feature generation algorithms, and alignment methods, pixel mappings, and image manipulation algorithms [33]. To read further on the different elements of image stitching see [33].

By stitching the images, we would have a seamless overlap. This would make the transitions between cameras as smooth as possible for the trackers. OpenCV has a powerful image stitching pipeline, with a variety of algorithms that can be used [3]. For our purposes, we would need to adjust and fine tune

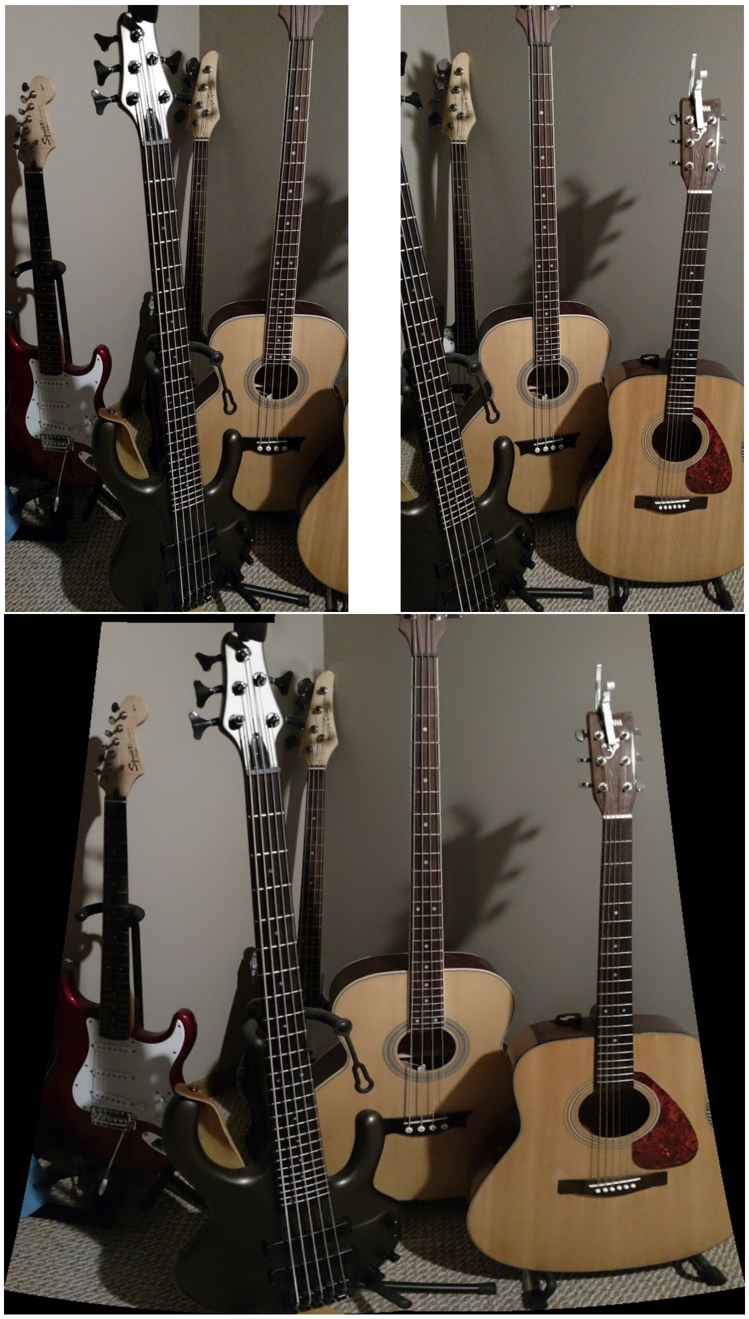


Figure 4.10: Two similar images of guitars, before being stitched by the default OpenCV stitcher and after.

the pipeline. One important thing we would need to do would be to cache the homography matrix that maps the pixels between the images. This should not change as our cameras do not move, so we want to cache it to improve performance and maintain a consistent alignment.

Stitching is not necessarily problem free. There could be a slight problems with angle differences between cameras, as they would not necessarily be right next to each other as they typically are in image stitching scenarios. However, we do not know to what extent this will affect our ability to stitch the images as we have not been able to set the cameras up as desired.

The alternative is to do some sort of blending or overlap. This is simpler to implement, does not require any complicated manipulation of the frames, and would be the only option if image stitching proved unsuccessful. The overlapping portions of the frames could be simply blended, so that the transition would be seamless. However, this would not reduce issues caused by the angle between the cameras. It remains to be seen if the tracking algorithms can handle these blended transitions. Instead of blending, the camera feeds could simply be lined up to cover the entire court without overlapping. The trackers would likely be unable to move from one feed to another, but creative usage of re-initialization and motion vectors of the trackers could solve this problem.

The exact details of image manipulation largely depend on the camera setup, and because the cameras were never set up in our desired locations, we do not know the exact details of how image manipulation would be done. In a later section, we will discuss alternative approaches we could take and possible solutions. Again, to further discuss our theoretical system, we need to assume this problem solved. From this point forward, we assume that we have a full top-down view of the court, that the trackers can move across this view without issue.

4.2.4 Tracking

OpenCV offers a variety of trackers ranging from machine learning varieties to trackers that utilize linear algebra techniques. Before we explain how we used the trackers, we are going to give an overview of some of the various trackers, how they work, and why we did, or did not, decide to use them.

Boosting Tracker

The first tracker that we investigated in OpenCV's library was the Boosting Tracker. This tracker performs real-time object tracking using a version of machine learning algorithm AdaBoost [3].

This tracker was proposed in 2006 by Helmut Grabner, Michael Grabner, and Horst Bischof. Conceptually, this algorithm treats tracking as a classification problem, with the item being tracked a positive example, while the rest of the image is composed of negative examples. This algorithm is online, meaning no prior training data is needed: it is generated as the tracker operates [21].

Before we dive deeper into how the tracker actually works, it is beneficial to quickly discuss the AdaBoost algorithm that the tracker is based on. The AdaBoost algorithm is a boosting classification algorithm, meaning it is comprised of numerous *weak learners*. These learners are essentially classifiers that use a weaker machine learning algorithm. In each iteration these learners create a hypothesis for the training data which is measured for accuracy. The training examples are weighted based on the learners performance so that the learners will more heavily focus on misclassified examples. Once the weak learners generate their hypotheses, they are weighted based on their accuracy. The final hypothesis for the iteration is generated by a linear combination (which can be thought of as a weighted majority vote) of all of the weak learners. Once this hypothesis is generated, the next iteration will begin using the accuracy of the previous hypothesis for error correction [18].

In order for this algorithm to be online, it needs an addition feature called a selector. Selectors are randomly seeded with their own pool of weak learners. These selectors pick the weak learner in their pool with the lowest error *on the samples seen so far*, so it can be used for the final hypothesis [21]. This concept is the basis of making AdaBoost effective, both in general, and in this tracker application.

In order to use AdaBoost to classify parts of images, we need features. The Boosting Tracker uses Haar-like features, orientation histograms and a variation of local binary patterns, simultaneously. In order for this to be possible, all features are computed efficiently, and can be computed in real time. Once features are obtained, hypotheses must be generated. To get a final hypothesis, the probability distribution of positive and negative samples is modeled using a Kalman filtering technique. The different features also require different learning algorithms for the weak learners. For the Haar-like features, a threshold and a Bayesion decision criterion are used. For the other two feature types, nearest neighbor learning is used [21].

Grabner, Grabner, and Bischof demonstrated promising results in their original proposal for their tracker. They were able to show that the tracker was adaptive, due to its ability to rely on multiple different feature types. They also demonstrated its robustness by showing that it is able to handle a variety of situations including occlusion, scale, lighting changes, etc. This is strongly related to the trackers reliance on multiple feature types, and constant learning. The generality of the tracker was also demonstrated. Due to the tracker′s online nature, it can be initialized with a bounding box and track from there. This means that the tracker can be applied to just about any situation. At the time of the proposal of the tracker, there was no standard way to evaluate its performance, so there was no way to determine its abilities compared to other trackers [21]. However, it has been shown to be very effective in a variety of situations, and is often used as a measure of comparison for newer proposed trackers.

We have attempted to apply this tracker to our problem and found that it was very effective for tracking, being able to handle rapid movements, scale changes, and even partial occlusion. However, it had a single, very important shortcoming. This tracker cannot report its failure. Because it is constantly learning, it can learn incorrectly and lose its target object without reporting failure. This is huge problem for our application because we need to be able to reliably know when a tracker needs adjusted or re-initialized so that the user can do so. For further reading on the Boosting tracker see [21] and [3].

MIL Tracker

The next OpenCV implemented tracker we investigated was the MIL tracker. The original article that proposed this tracker is no longer available, so what we know of this tracker comes from OpenCV and our own testing. The tracker was originally proposed by Babenko, Yang, and Belongie, in 2009. MIL uses Multiple Instance Learning (thus the name) to avoid tracker drift, which makes it a robust tracker. Like the Boosting tracker, the MIL tracker is online and uses a classifier to separate the tracked object from the background [3].

In our tests, we found that the MIL tracker was robust, and handled changes in scale and orientation better than other tested trackers. However, this tracker has the same large flaw that the Boosting tracker has: it cannot report failure. Thus, like the Boosting tracker, it cannot be used alone for our application. However, it could be used in conjunction with another tracker, as outlined later in this chapter.

MedianFlow Tracker

The MedianFlow tracker is a tracker designed for smooth and predictable movements [3]. This tracker uses a novel error detection method to remove outlier points in the tracking. Technically, this error detection method can be applied to any tracking algorithm, but in OpenCV and the original research it was applied to a Lucas-Kanade optical flow tracker [24].

Conceptually, the MedianFlow tracker's error detection is based the forward-backward consistency assumption: correct tracking should be independent of time-flow. In simpler terms, the tracker assumes that it could track an object in footage both forward and backward in time. This idea is leveraged directly in the tracker's algorithm to find error of individual feature point trajectories, and thus detect when the tracker as a whole has lost its target object. To determine the error of an individual point trajectory on a sequence of images, the tracker first tracks the point forward throughout the images creating its trajectory. Then the tracker is applied to the frames backwards in time. This creates a trajectory from the final point back (hopefully) to the starting point. The error of the trajectory is then defined as the distance between the forward and backward trajectories. A perfectly tracked point will yield a backwards trajectory back to its origin, while a poorly tracked point will yield a trajectory that moves in the wrong direction. The main advantage of this approach is it is applicable to just about any tracking algorithm [24].

While tracking individual independent points is useful, most tracking problems involve tracking collections of dependent points that make up objects. To track objects, the MedianFlow tracker only needs a bounding box. Once the bounding box is initialized, it is populated with points. Then the points are then updated via the Lucas-Kanade tracker, and their errors are calculated. The points are then filtered by error, so the best 50% of points remain. These points are then used to estimate the movement of the bounding box [24].

The researchers responsible for the MedianFlow tracker found it to perform better on certain tracking sequences than state of the art trackers at the time. These sequences involved consistent and predictable motion, which seems to be this tracker's strong suit [24]. However, there are a few issues in using this tracker for our problem. First, like the previous two trackers we have examined, it cannot properly report its failure. Also, we found that this tracker had problems keeping track of some of the faster movements of the basketball players that the MIL and Boosting trackers were able to catch. For further reading on the MedianFLow tracker, see [24] and [3].

GOTURN Tracker

The GOTURN (Generic Object Tracking Using Regression Network) tracker approaches the tracking task in a similar manner to the MIL and Boosting trackers. However, this tracker has one very important difference, it is offline. This means that it is pre-trained, as opposed to training on the object it is trying to track. The tracker was trained to track generic objects using a wide range of videos and images. The offline nature of the tracker increases its computational performance immensely and allows it to track well on a variety of objects immediately at initialization [22].

The GOTURN tracker is a neural network. It is trained entirely offline on videos with each frame labeled with the location of the object. The objects vary, ranging from human figures and faces to motorcycles and sharks. The network is also trained on single images with objects labeled, so that the network could pick up on a large variety of different objects [22].

This offline training mitigates most of the work that other neural network trackers do at runtime. Also increasing the performance is the fact that the architecture of the network was designed in such a way that only a single feed-forward pass through the network is required to update the tracker. This is because of the regression-based approach the tracker takes to locating the object. The end result is a extremely high performance tracker that can run in excess of 100 frames per second [22].

This tracker can also be specialized specifically for a target object type to improve its accuracy significantly. This is done by replacing the generic training data that the authors used with application specific data. The authors found, at the time of their writing, that this tracker is in the upper echelons of performance, having fairly good robustness, and accuracy rivaling the best, cutting edge algorithms. However, despite the performance and accuracy positives, this tracker has one major flaw: it cannot handle occlusion at all in its current state. This is important, as our targets, the players and ball, will frequently be occluded by both each other and other objects, like the backboard and basket. This issue will likely be ironed out by the authors of this tracker at a future time, but as of this writing, it does not suit our purposes [22]. For further information about the GOTURN tracker see [22] and [3].

TLD Tracker

So far, all of the trackers we have examined have looked at tracking in roughly the same way: take an initial bounding box that represents an objects position and update the position of the bounding box in the next frame. The next tracker we examined approached object tracking in a different way. The TLD (Tracking-Learning-Detection) tracker decomposes the tracking problem into three parts: tracking, learning, and detection, as the title suggests. The tracker simply follows the object frame-by-frame. The detector finds all appearances of the object that have appeared so far and corrects the tracker. The learning portion estimates the detectors errors and updates the detector to avoid future errors [3].

The TLD tracker starts with a model of the object. This is a data structure that consists of positive and negative image patches. Where the positive patches represent the image and the negative patches represent the background. The positive patches are ordered according to when they are added to the model. The model also defines some similarity measures which are used for a nearest neighbor classifier to determine if a patch represents the object. This classifier is used for the detector and to determine when to add new patches to the model [25].

The TLD tracker was proposed by the same researchers that proposed the MedianFlow tracker, and thus they use the MedianFlow tracker for the tracking portion. However, they extend the MedianFlow tracker to detect failure, in order to make it usable in situations where the object leaves the frames and returns. Failure detection is done by a threshold on the median of the residual of the individual point displacements [25].

The detector and learning portion of this tracker is really what sets it apart from others. The detector scans through all of the patches in the image and determines if they represent the object. The detector looks for a large set of different scales and distortions of the object. Because there are such a large number of possibilities that need checked, three separate stages are checked for the sake of efficiency, and if the patch fails at any level, it does not represent the object. These stages represent a cascaded classifier. The first stage is patch variance: all patches that have a variance smaller than 50% of the variance of the target patch are selected for tracking. This stage typically rejects more than 50% of the non-object patches. The second stage of the cascaded classifier is an ensemble classifier that consists of numerous base classifiers that do pixel comparisons between patches. The final stage of the classifier is the nearest neighbor classifier. At this stage, there are only a handful of possible bounding boxes remaining. Anything positively classified by the nearest neighbor classifier becomes a response from the detector [25].

The detector and tracker work together to generate a bounding box. If neither system returns a bounding box, the object is considered not visible. Otherwise, the maximally confident bounding box is output, measured using a similarity value [25]. This is an important thing to note, as it helps explain the trackers sometimes erratic behavior.

The job of the learning portion of TLD is to initialize the detector in the first frame, and then update it with every frame, based on growing and pruning principles. To initialize the detector, the learner takes the initial bounding box and synthesizes 200 positive patches from it. The negative patches are collected from the surroundings of the initial bounding box. These training patches are then used to update the object model and the ensemble classifier. From here, the detector is updated by "P-expert" and a "N-expert". These "experts" represent the pruning and growth pieces of the learners. The P-expert uses the trajectory of the tracker to add new positive examples to the object model. This expert determines if the current location in the trajectory is reliable, and if it is, adds 100 new positive examples from that location. The N-expert generates negative samples from the background. The idea is that if the object location is known, it can only be in that place, so the area around the object (at a certain distance from the object) is added as negative examples [25].

The researchers that proposed the TLD tracker found that it outperformed the other trackers of the time. They ran a large set of experiments against other competitor trackers, including the Boosting and MIL trackers discussed earlier. They found that TLD fared better than all of the other trackers in nearly every experiment, in every measure of error [25].

In terms of our usage, this tracker seems like a good candidate. It can detect its own failure, and is, according to the authors, significantly better than the previous trackers we have examined. However, in our tests, we have found that this tracker behaves erratically in tracker players. The scale changes rapidly and moves all around the players' bodies. This may be an issue with how long the tracker has had to train, but it ultimately makes extracting data from the trackers difficult. We desire smooth movements from trackers, and the TLD tracker doesn't seem to provide. For further reading on the TLD tracker, see [25] and [3].

KCF Tracker

The authors of the paper that proposed the KCF (Kernalized Correlation Filter) tracker noticed that most existing trackers focused heavily on characterizing the object being tracked, and did little to leverage the wealth of negative samples that the image background is composed of. The reason why, until this point, the background has been underutilized, is that there is an unlimited amount of negative samples that can be taken from the background. Utilizing these examples yields an infinite sink of computational power. Most modern trackers try to strike a balance between positive and negative examples, where the KCF tracker, through the use of some linear algebra tricks that we will outline further in this section, is capable of incorporating vastly more negative samples without a severe impact on performance [23].

The authors first discuss some of the needed “building blocks” necessary for their tracker. The first piece is linear regression. The tracker, at its core, uses a form of linear regression called ridge regression. Ridge regression is used because it has a closed form solution. However, this solution requires a large system of linear equations to be solved, which would be computationally expensive. To negate this issue, the authors use properties of cyclic shifts and circulant matrices [23].

A cyclic shift is a vector operation, where all elements are shifted a position over, and the element shifted off the end of the vector wraps to the other side. Consider the vector:

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix}$$

Applying a cyclic shift to the right yields:

$$\begin{bmatrix} x_5 & x_1 & x_2 & x_3 & x_4 \end{bmatrix}$$

We can apply a cyclic shift to vectors $n - 1$ times, as n cyclic shifts yield the original vector.

A further application of the cyclic shift is the circulant matrix. A circulant matrix is a nxn matrix where each row is the cyclic shift of the row above it. A circulant matrix generated from the previous example vector would be:

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ x_5 & x_1 & x_2 & x_3 & x_4 \\ x_4 & x_5 & x_1 & x_2 & x_3 \\ x_3 & x_4 & x_5 & x_1 & x_2 \\ x_2 & x_3 & x_4 & x_5 & x_1 \end{bmatrix}$$

Because a patch from the image is represented as a vector of pixels, a cyclic shift can be used to generate simple translation in one direction. This allows the generation of $n-1$ virtual negative examples from a nx1 vector of pixels. $n-1$ examples are created because n cyclic shifts would yield the original vector. Using this set of negative examples, along with the original example vector, we can create a nxn matrix by concatenating the row vectors together. This matrix is, by definition, circulant [23].

This matrix is used to compute a regression on these examples. Because this matrix is circulant, we can use the Discrete Fourier Transform to make the matrix diagonal. The Discrete Fourier Transform is commonly used to perform Fourier analysis in many practical applications, solve partial differential equations, and prep matrices to help in solving linear systems of equations. Diagonal matrices have many desirable properties, most prominent of which is that it makes calculations involving the matrix much simpler. This is key to the performance of the KCF tracker and why circulant matrices are widely used in numerical analysis. When applied back to the linear regression, the end result is the formula for a regularized correlation filter. It is also worth pointing out that the use of circulant matrices reduces the computational complexity down to $O(nlog(n))$ from $O(n^3)$ [23].

From this point, the authors go into a discussion on previous work they did to take this linear concept and extend it to non-linear regressions. This gets very complicated, so it is best left in the words of the authors themselves. The end result of this work is a kernelized version of ridge regression, which in turn is used in their Kernalized Correlation Filter tracker. For further reading on the mathematics behind KCF see [23].

In terms of performance, the authors found that the KCF tracker performed competitively with most other modern trackers. They also found that it outperformed all other trackers that use a similar methodology, suggesting that the inclusion of extra negative examples provides a performance boost. The authors compared their tracker to many of the others implemented in OpenCV, showing that it performed better in general [23].

For our purposes, the KCF tracker seems like the best single tracker available. It has great performance, handles occlusion and rapid movements fairly well, and is capable of detecting failure. It does, however, have some downsides. The most notable of which goes along with its failure detection: it fails frequently. The authors make no mention of this in their paper, so it likely has something to do with our application. The OpenCV implementation also seems to be unable to detect scale changes and adjust the bounding box accordingly. Once again, the authors make no mention of this in their paper. Ultimately, if we had to choose a single tracker, this would be our choice. However, for our application, a hybrid tracker or a tracker trained specifically for our application, would likely perform better.

Tracker Usage

The trackers we have discussed above are the center piece of our theoretical statistics collection system. These trackers are the source of the information we wish to collect. The trackers follow all of the players, along with the ball on our top-down view of the court. The trackers are also used in the side views to tracker the ball. Thus, the trackers will yield the x and y positions of the players, and the x, y, and z position of the ball. This is the simple part, the complicated issue is ensure the trackers do their job for an extended period of time. This is why the selection of tracker is key: our trackers must be reliable, able to report failure and handle the rapid movement of the players and ball.

We tested the various trackers out on sample footage of a player taking basic shot and collecting rebounds. We found that the KCF tracker individually was the best middle ground between reliable tracking and reliable failure detection. However, we found that it failed frequently, and was unable to handle many scale and orientation changes. Other trackers, especially the MIL, tracker handled these changes better, but could not report failure. Clearly, a single tracking algorithm from the OpenCV library will not be sufficient.

At this point we had three options: creating our own tracker, using a newer algorithm outside of OpenCV, or creating a hybrid tracker. Due to time constraints, creating our own tracker is outside of the scope of this project. That leaves us with using existing trackers. We did not find any trackers outside of OpenCV that provided large improvements in performance, so we are left with looking at simple modifications or combinations of existing

OpenCV trackers. Our theoretical solution is to combine two of the above trackers into a hybrid tracker that has the benefits of both of its components.

Our hybrid tracker is a combination of the KCF tracker and the MIL tracker. We want the failure detection and performance of the KCF tracker, along with the scale and orientation change handling of the MIL tracker. Our combination is implemented in a rather simple way: both trackers are initiated in the same place. The returned value for the center of the tracker is the average between the centers of the two trackers. In order to combat the premature failure of the KCF tracker, we allow a certain (adjustable) amount of failures before the tracker reports failure. In the event of a failure (before the allowed amount has been met) the KCF tracker is reinitialized with the MIL tracker's bounding box. This will update scale changes, and allow the tracker to continue on in what (should) be the right place.

There are, however, some possible downsides to this theoretical tracker that ultimately make it unlikely to be the solution we need. First and foremost, ignoring any amount of failures reported by the KCF tracker could lead to incorrect data. In our tests, we found that as long as the maximum number of failures is kept low, the tracker will not drift very far. However, instances that cause problems with the MIL tracker could persist. For example, it is possible for the MIL tracker to learn a stationary object that occluded the player (like the rim and backboard) and then reinitialize the KCF tracker there after a failure. This would lead to an unreported failure. This tracker would also have lower computational performance because each update would actually involve updating two trackers. Ultimately, a better way to combine these desired traits would be to create a new tracker entirely, combining the various positive features of trackers we have looked at.

Our current tracking solutions do not seem to be sufficient for long term tracking. Data could be collected with these trackers, however, it would require frequent human correction. This area, like many others in this project, will require more work before a working system could be built. In a later section, we describe further approaches we could take in tracking to get more reliable trackers and higher performance.

4.2.5 Optical Character Recognition

In order to gather data about the basketball game to augment the spacial and temporal data, we decided to use Optical Character Recognition to read the scoreboard in our video feed. This allows us to get the current score, the current game clock, and the current shot clock. This data allows our system to keep track of the current state of the game, and make the calculation of various statistics easier.

Before we dive into our usage of OCR, we first give a brief explanation of what it is and different ways it can be done. OCR can be defined as the process of classifying symbols in a digital image. These symbols can be alphanumeric or any other script. Work on OCR goes back to the early 1900s, with the first systems appearing in the 1940s alongside digital computers. These were simple systems designed to read simple fonts, usually typeset [17].

These early systems utilized simple statistical classifiers. Through the next 20 years, these systems were improved and the first commercial systems emerged in the 1950s. Through the next 40 years, tech giants like IBM and

Toshiba built increasingly powerful and robust machines, capable of handling a variety of fonts and handwritten characters. In the 1990s, machine intelligence techniques were added to the existing algorithms, creating increasing complex and powerful OCR algorithm. This new class of algorithms made use of neural networks, hidden Markov models, fuzzy logic, and other techniques [17].

OCR systems rely on a complicated pipeline of preprocessing, image segmentation, feature extraction, training, and recognition. These various pieces can be done in numerous ways, with many different algorithms and techniques. Typically, a system is trained on a large set of labeled characters so that it "learns" a character set. Then, when using the system, a set of text to be recognized is preprocessed, segmented, and then run through the feature generation system. These features are then run through the recognition system, which is the previously trained element. This could be a classifier, neural network, hidden Markov models, etc. This general pipeline is common to nearly all OCR systems. To read more on OCR systems in general, along with modern techniques, see [17].

To do OCR, we use the Google Tesseract Engine. Google's Tesseract is an open source OCR engine. Tesseract was originally developed by Hewlett-Packard in the 80s and 90s, and in 2005 it was open sourced, and has since been developed by Google. Tesseract does not follow the outline we discussed earlier exactly, and turns out to be quite complicated. To read further into the inner works of Tesseract, see [8]. At the most basic level, Tesseract takes in an image and outputs all of the detected text contained within the image. There is more functionality, and multiple character sets and languages that make Tesseract an extremely powerful tool [8].

Our usage of Tesseract occurred in two places, reading of the timestamp on our video feeds (described in a pervious section), and to get information from the scoreboard. To get information from the scoreboard (or the timestamp), the first step is to isolate the scoreboard (or timestamp) in the feed. This is as simple as taking the portion of the video feed where the desired information is located. Because the cameras do not move, this can be done reliably at every frame.

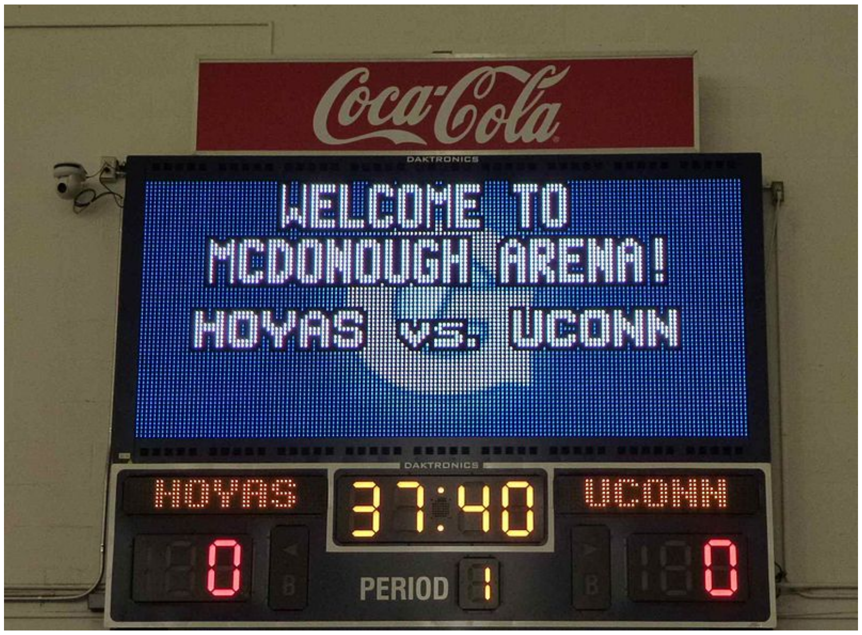


Figure 4.11: A sample basketball scoreboard [9].

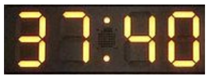


Figure 4.12: The sample scoreboard cropped to a single relevant field [9].

Once the text is isolated, a little bit of preprocessing is done to make the text clearer and easier to read. This ensures that Tesseract gets the most accurate result. The preprocessing typically consists of blurring and applying thresholds to get a binary image, only containing the text.

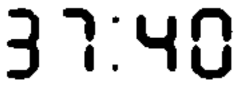


Figure 4.13: The relevant field processed for maximum contrast so that it is optimal for the OCR.

Once the scoreboard is isolated (Figure 4.11), the image is divided up into the individual fields so that each field can be fed to the OCR engine separately (Figure 4.12). At this point, each field is preprocessed (Figure 4.13) and then fed to the OCR, and the result is recorded. Any invalid values are thrown out, however, these are rare as the OCR is very accurate.

One point worth noting: our example here uses seven-segment digital numbers. Every scoreboard does not necessarily use this format: there are also scoreboards that use a series of dots, as well as scoreboards that use a digital screen. This methodology should work in these scenarios as well because of the preprocessing we do. If our current form of preprocessing is not enough it is possible to blur the image further, or train Tesseract using the font of the scoreboard. No matter the format of the scoreboard, we should be able to read it eventually, we may just need to make a few changes.

4.2.6 Bringing the Data Together

Collecting data from the system is not a complicated process. For every set of frames the following data points are collected:

- The (x, y) coordinates of each player on the team along with their name. These values come from the center of the players' tracker bounding box.
- The (x, y, z) coordinates of the ball. The x and y values come from the center of the top-down view tracker bounding box and the z value comes from the center of the side view tracker.
- The current game clock.
- The current shot clock.
- The current score for each team.

In order to collect these values, we have to define the coordinate spaces that they reside on and some special values needed.

We define the 2D coordinates such that $(0.5, 0.5)$ is center court, and the corners are $(0, 0)$, $(1, 0)$, $(0, 1)$, and $(1, 1)$ (Figure 4.14). The z coordinates would need some calculation as there is some perspective issue with the camera location. We cannot make this calculation without knowing exactly how the cameras would be positioned. However, post calculation, we would like a z coordinate of 0 to be the floor and 1 to be around 30 feet in the air. This would allow most shot trajectories to remain in view of cameras.

The special values we would need would indicate when a player or ball is out of frame or if the OCR failed. In these case we would insert $(-1, -1)$ for the players and $(-1, -1, -1)$ for the ball. For the game clock, shot clock, and scores, if the OCR fails, the last known value would be inserted.

All of these collected values would form a csv file with each value being a column and the rows being the timestamps for each collected frame. In this

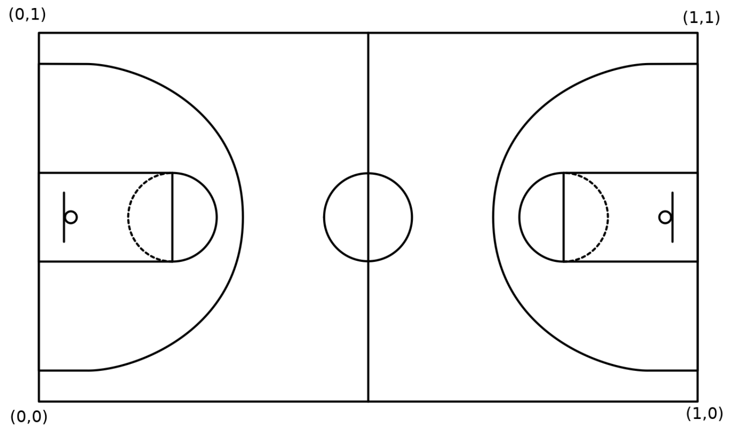


Figure 4.14: The bounds of our court coordinates.

format, data could be saved for later analysis or analyzed immediately.

4.3 Interface and Management Program

Now that we have the technical pieces together, we need to manage them. Part of the original project goal was to create a user friendly software system to manage the camera system and the data collection. The software would allow the user to record a game, run a recorded game through the data collection system, and analyze games to generate statistics reports. These three pieces would be tied together with a larger menu that allows the user to select files and other important pieces of information. Of these pieces, only the recording piece exists in a fully working state (covered earlier). In this section we will cover data collection piece and the analysis piece: the original plan for them and how they would work.

4.3.1 Data Collection Manager

This piece of the software was the hardest to envision, as it requires user interaction to initially identify players, and to ensure the trackers work correctly. Ideally, the user interaction would be kept at a minimum, so it is best to first define what the user needs to do specifically. The user must:

- Enter teams and their full roster (or load them from a database),
- initialize trackers for all players on the court,
- initialize the trackers required for the ball,
- re-initialize any trackers that fail,
- and at any change of players on the court, disable trackers for exiting players, and enable trackers for entering players.

We propose a simple interface to handle these tasks, based on earlier prototype built with TKinter, OpenCV, and Python (Figure 4.15). The iteration we propose would likely be built in C++ with OpenCV and Qt5 for performance reasons.

The interface would be one large window, with a tool bar for the teams and players on the right side, utility buttons on the bottom, and the rest of the window would consist of the three views of the court: the 2D top-down view and the two side views. This would ensure the user had full view of the game while running the data collection and easy access to all needed utilities.

The bottom utility bar would consist of a pause button, a “ball” button, a “forward”, and a button a “back” button. The back button would allow the

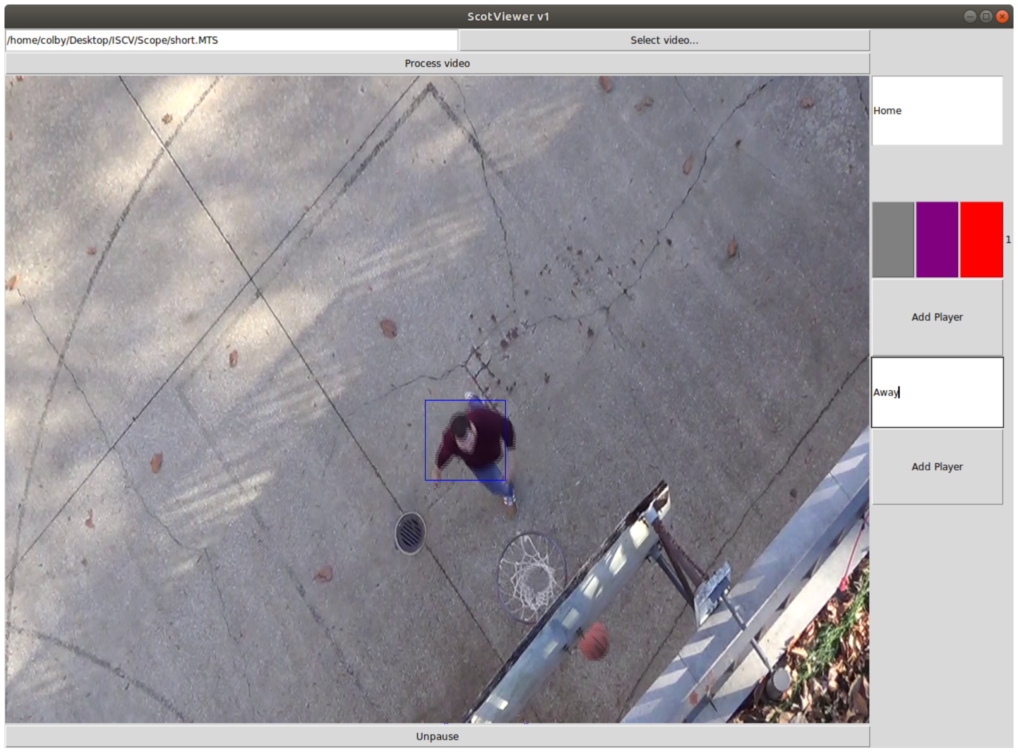


Figure 4.15: Python prototype for the Data Collection Manager.

user to move back a frame at time to correct mistakes. In a similar vein, the forward button would allow the user to skip forward frames to find the next important portion of the game. Pausing would allow player identification and tracker changes to happen at the users pace. These navigation buttons would allow the user to move throughout the game at their pace, ensuring accuracy. The “ball” button selects the ball, which allows the user to initialize the ball tracker by clicking where they would like it initialized on the video feeds.

The right “teams and players” tool bar would be used to select and initialize player trackers and adjust them. Initially, the user would enter or load the two teams, and then add individual players until the entire rosters are in the tool bar. At that point, the tool bar would contain two lists of players, one for each team, with two buttons next to each name. One to select the player, another to disable that players tracker. Selecting the player will allow the user to click on the video where they want to initialize that players tracker.

Now that we have the interactive objects described, we will discuss the overall flow of the program. When the user starts up the program with a game selected, they will first be required to enter or load the teams and a minimum amount of players. Then, they move through the video until they find tip-off. From there, all trackers would need to be initialized. At this point, the user only needs to ensure that the trackers are working as intended, and that if they fail they are re-initialized where they belong. Anytime players enter or leave the game, the user would need to adjust the trackers.

It is not entirely clear if this is the best method to manage the data collection, however, we believe this would be effective. The amount of user interaction entirely depends on how reliably the trackers work: trackers that need adjusted every few seconds would make for a lot of work.

4.3.2 Game Analysis and Statistics Reports

This portion of the interface would be much simpler. The main goal of this portion of the software is to allow the user to generate statistics and visualizations to suit their needs. Thus what this interface contains entirely depends on which statistics are calculated from the data. We envision a suite plots, heat maps, and tables that the user can select and generate from the data. Aside from this, we cannot further describe the interface without knowing exactly what statistics we would use. Due to reasons outlined elsewhere in this chapter, we were never able to generate real data, and thus we were unable to generate real statistics.

One thing worth mentioning is that instead of creating this interface in C++, we believe it would be much better to build it in Python, using Qt5 and matplotlib, a plotting and graph package. The matplotlib package is one of the best plotting packages out there, and, combined with Python's simplicity, would make intricate and adjustable plots easy to create.

4.4 Statistics Calculations and Reports

As mentioned above, we were unable to generate actual data, and thus were unable to generate actual statistics. Thus, all we are able to discuss is some of the statistics we planned to generate from collected data. The NBA and SportVu set a good example of what could be calculated, and we based a lot of our planned statistics off of their example.

Many of the useful basketball statistics do not really rely on spacial or temporal data, and can be calculated effectively without our theoretical system, however, it would be able to calculate some of them automatically, saving time and money. These basic statistics would include various shot percentages, minutes, steals, points, etc. However, this system would still miss out on some statistics that can be hard to attribute to specific players like blocks. Thus this system would augment current statistic collection, but not replace it, similar to the way the NBA currently operates.

The data we planned to collect really shines when used to generate statistics based on location and time. Using this data, shot percentage could be calculated for much more specific locations on the court, allowing coaches and players to see exactly where they make the highest percentage of shots. This is something SportVu does, and provides interesting insights into players' play styles. The spacial data could also be used to determine how well defended a player is when shooting. This allows coaches and players to differentiate between open and contested shots, which, have a huge difference in success rates. This would manifest as some sort of "coverage" measure that would take into account how far the defenders are from the shooter and how many defenders are nearby. Using the temporal data, we can calculate information based on how long players have possession of the ball. For example, we can see how many points a player makes based on how long they have possession in the game. These stats would help normalize statistics between players that have varying play time and possession time.

Part of the goal of this project is to generate data that coaches and players can use to help make strategic decisions. Statistics that would help with those decisions largely depend on what the coaches and players want. New statistics are an active area of research in sports analytics and thus this area of the project is extremely open, and a source of just about endless future work.

Chapter 5

Optical Statistics Collection: Results

The original goal for our Optical Statistics Collection system was to build a system and implement software to help automate statistics collection and collect novel spatial and temporal information. The point of this data collection is to aid coaches and players in strategic decisions. Because our system never became a reality, this goal was not accomplished. Thus, the "results" are not concrete, but rather theoretical.

Our primary result is that we believe that the system we originally envisioned is possible, just outside of the scope of an undergraduate thesis. We simply did not have the time or the money to make this system a reality. Each of the components of the project that we came up with would theoretically be sufficient, but the cut corners and simpler approaches taken would add up to make a lower performance and less accurate system. In order to make a high performance system that is capable of full statistics collection, we would need a team of software developers, significantly more time and money, and entirely custom software and cameras.

In the following section we will outline the challenges we faced in progressing through the project. Many of these issues were hinted at in the methodology, however, we will give a full overview of the causes and effects of these problems.

5.1 Challenges

At every point in this project, we were dealing with some sort of problem. Whether it was shortcomings of the cameras, problems with OpenCV, or logistic issues, we always had something preventing further work on our system. The best way to show why this system wasn't possible is to outline the big problems, explain where they came from, and why we could not solve them. Possible future work to solve these problems is discussed in a later chapter.

5.1.1 Cameras

Our choice in camera was one of the root causes of many of our problems. We chose a CCTV camera primarily in hopes of balancing costs and benefits. Having used these cameras at length, we can safely say that we needed better cameras. The primary issue we had with our cameras was frame rates and consistency.

In order to properly use multiple video feeds to cover the game, we need synchronized video. With our cameras, this turned out to be incredibly difficult. Our cameras had a variable and inconsistent frame rate, making synchronization only possible with outside information. We attempted to use the timestamp to synchronize but this proved to be inefficient and worked inconsistently. The cause of our frame rate problem is the way our cameras had to be accessed. The camera sensors themselves were likely sufficient for our purposes, but the cameras had to be accessed through Amcrest's firmware. Thus we could not synchronize as we recorded, which is ultimately the best case in terms of recording. Our only option was to record frames from the cameras as they are retrieved from the camera after they have been processed by the firmware. The end result is the inconsistency in frame rate that makes synchronization so difficult. We are unsure what about the camera's firmware caused the inconsistency, but it is likely due to the fact the cameras were not intended for applications like ours.

Attempting to use poorly synchronized video would result in inaccurate data. Beyond this, applying computer vision algorithms would become more difficult, and they would be significantly less effective. This wouldn't prevent the system from working per se, but it would severely limit the performance and accuracy.

5.1.2 Computer Vision

Aside from issues with our cameras, we also had issues with some of our computer vision techniques. Our primary issue here dealt with our trackers and how to best leverage them. We investigated all of the useful trackers available in the OpenCV library, and found a few of them useful (KCF and MIL primarily). However, we discovered in early testing, that when applied to our application, especially in tracking the ball, had some insurmountable performance issues. We found that none of the tested trackers could reliably track players for more than a few minutes, and none could track the ball reliably for more than a few seconds. This could be an issue with how the testing cameras were set up, or with the OpenCV trackers. Ultimately, we determined that we needed better trackers. Our hybrid tracker, while theoretically better than the individuals we tested, had problems of its own. We believe that we would need better general trackers, or trackers more suited to our application. This could take the form of a new algorithm or a modification of existing trackers.

The end result if we moved forward with these poorly performing trackers would be inaccurate data, and the data collection process would become quite laborious. The user of our data collection system would have to reset and update trackers constantly. This is unacceptable, and goes against our original project goal. Again, these issues would not stop our system from working, but the performance hit and extra work would reduce the efficiency of our system. We would be unable to process games in a reasonable amount of time, and would likely generate sub-par data.

5.1.3 Logistics

Our logistic issues, as mentioned earlier, prevented us from properly setting up our cameras to test. Due to the age and design of the gym, finding power and mounting points for the cameras was difficult. We would need long power cables, which could be dangerous. Also, getting permanent mountings for the cameras would require renovation of the gym. Ultimately, it made no sense to try and mount the cameras, even temporarily, because our gym could not accommodate them. Even if we did set the cameras up, due to a planned renovation, they would have been removed shortly in the coming Spring. The end result of being unable to set up our cameras is that we could not test *anything*. Without a proper camera setup, we could not test our trackers on realistic data, we could not test our image manipulation, and we could not even test the validity of our theoretical camera setups. This also meant no fine tuning to any computer vision technique we could test.

This issue proved to be the single largest hurdle in building a working system. Unlike the other problems we had, we cannot build a system that we cannot set up. We looked for alternatives and temporary setups, but found nothing that suited our needs. Building a system as we planned essentially requires that the camera system be in place first, as all of the software needs to be built around it.

Chapter 6

Shot Analysis with Pose Estimation: Methodology

Along with using computer vision techniques to collect statistics and analyze entire basketball games, we wanted to do something narrower within the realm of sports analytics. This is where the second part of our project comes in. In this section, we will describe the methodology used to analyze an individual's basketball shot in terms of body pose. We will explain how the shooters body pose is collected, and how we cluster the results to determine whether or not a shooter's body motions are indicative of whether or not they make a shot.

6.1 Recording Poses

In order to analyze a players shot form, we need a way to gather the position of their body as time progresses. This is something that people can intuitively do with their eyes, but for a computer this is quite challenging. The problem is known as pose estimation, which is an active area of computer vision research.

Creating software to do pose estimation is very complicated, so we decided to apply existing software to this problem. In searching for pose estimation software, we came stumbled upon a perfect tool for our needs: OpenPose.

6.1.1 OpenPose

OpenPose is the "first real-time multi-person system to jointly detect human body, hand, and facial key points (in total 130 key points) on single images," [4]. Essentially, OpenPose is capable of taking in an image and detecting key points (eyes, nose, various joints, etc.) on all human figures in the image. This allows the full description of a human pose in an image.

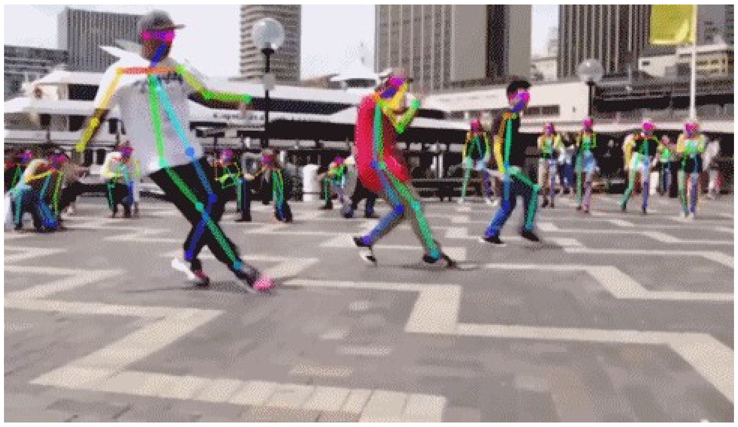


Figure 6.1: An example of body key point detection in OpenPose [4].

OpenPose is capable of detecting poses that include key points of just the body (Figure 6.1), along with poses that include intricate details on the face (Figure 6.2) and hands (Figure 6.3) [4]. For our application the body angles are the most useful. Before we move into how we used OpenPose, we first discuss generally how it works.

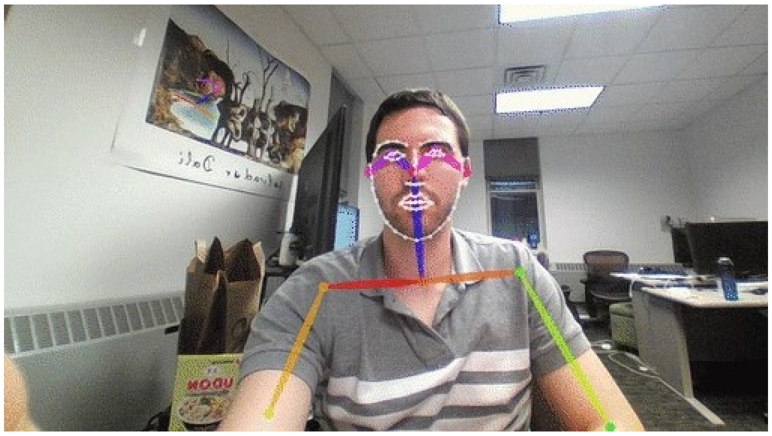


Figure 6.2: An example of face key point detection in OpenPose [4].

OpenPose is an incredibly complicated system, and is the result of the work from six authors (Gines Hidalgo, Zhe Cao, Tomas Simon, Shih-En Wei, Hanbyul Joo, and Yaser Sheikh) and three research papers [4]. It is outside of the scope of this paper to go into great detail on how OpenPose works, so we will just give a brief overview.

We start with the basis of the work, Convolutional Pose Machines. This work itself, is based on a previous work that proposed a pose machine architecture, which the authors describe as "the implicit learning of long-range dependencies between image, multi-scale and multi-part cues, tight integration between learning and inference, a modular sequential design," [35]. The authors (many of those who created OpenPose) extended this pose machine architecture to include convolutional neural networks. These convolutional pose machines are composed of a set of prediction neural

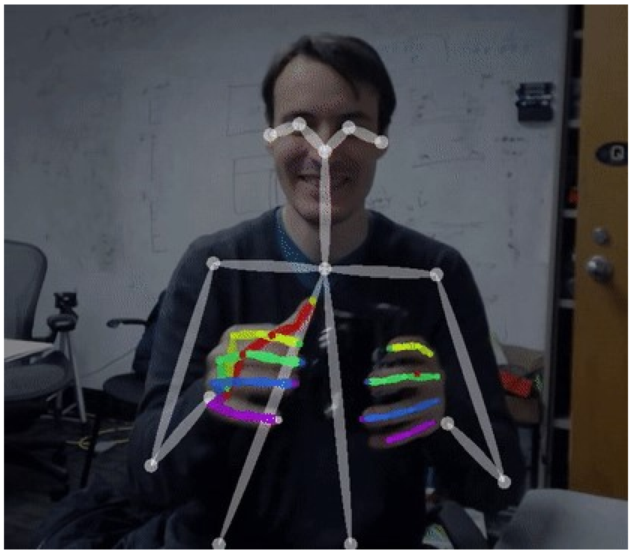


Figure 6.3: An example of hand key point detection in OpenPose [4].

networks that predict confidences for the locations of each body part. This set of networks forms a sequence that is traversed to improve the prediction at each network in the sequence. The other piece the authors contributed is a method to compose the networks in such a way that they have a large receptive field. This is important as it improves accuracy in learning long range spacial relationships. The authors found that the these convolutional pose machines outperform all other pose estimation methods on nearly every training set tested [35].

To use these convolutional pose machines, they are first trained on data sets that contain labeled human poses. Then, feeding an image through the machine will yield estimations for the pose of the subject in the picture. The one notable downside of these machines is that they can only estimate for a single pose in an image: they cannot handle multiple people without errors [35].

The hand and face models were added in other research papers. We do not use these features, so we will not explain them. To read further on these topics see [31] and [4].

The last key piece to get to OpenPose was to extend the functionality to multiple people in a single image. Again, we do not use this feature, so we will not explain it. To read further on the methodology to extend the pose machines to multiple people, see [16] and [4].

OpenPose is pre-trained, and has many available models, ranging from 18 key point body models to the full 130 key point model for body, hands, and face [4].

6.1.2 Using OpenPose

Now that we have an idea about how OpenPose works, we can dive into our usage. We wanted to take a 90 second clip of a player taking a shot, and convert that into numerical data about the positioning of their body. We wanted this to be done in a rather automatic fashion, so a the player can operate the software while taking the shots. In making this software, we decided to focus specifically on basketball free throws, as they are a predictable and repetitive motion. This will afford us the best chance to analyze the players shot form because, in theory, the player's free throws should have consistent form, and slight deviations in form should be detectable. To do this we created a C++ program (Appendix B.1) that we outline in the next few paragraphs.

The overall flow of our program is as follows:

1. At startup, the program is running in real time, grabbing the newest frame and processing it for a pose.

2. The user must generate a reference "starting pose" to look for as the start of free throw. To do this, the user presses a key, and gets into their free throw starting position. After a 10 second delay, the program saves the pose it detects as the reference pose.

3. Now that a reference pose is available, the program checks every frame against the reference pose to see if the user is about to shoot a free throw.

4. If the frame is a match, the program switches to recording and processing every frame. The program records the next 3 seconds (90 frames) of video and estimates the pose for each frame, adding it to a file.

5. The program prompts the user to tell if the recorded action was a shot. If it isn't a shot, the data is thrown out and the acceptable margin of error in checking the reference is decreased. If it is a shot, the user is asked if it was made. The data, along with whether the shot was made is then saved.

6. Return to step 3.

To start, we need to feed images to OpenPose and retrieve information from the estimated poses. We start by reading in images from a webcam using OpenCV. For this application, the greater the frame rate, the better the results. We settled on a laptop integrated webcam. This webcam is capable of recording at 30 frames per second, and was immediately available. Due to the high computational requirements of OpenPose, we found it best to put processing and capture into separate threads.

The capture thread has two states: real time, and full frame rate. In the real time state, the thread grabs the newest frame from the camera, removes all current frames from the its thread-safe queue, and puts this new frame at the top of the queue. This ensures that when the processing thread gets a new frame from the queue, it is the most recently recorded frame. In the full frame rate state, the capture thread puts every recorded frame into the queue. This state occurs when a free throw is detected. Once the required 90 frames have been recorded, the thread idles until the queue is empty. Once the queue is empty, the capture thread returns to the real time state.

The processing thread is where things get interesting. The first thing this thread does is grab the frame from the top of the capture thread's queue. Then, the frame is resized, scaled, and converted so that it can be entered into OpenPose. The frame is then run through OpenPose, and the (x, y) positions of the key points of the first estimated pose are extracted. We are only interested in the free throw shooter, thus why we only extract the first pose. This also means that other figures in the background could potentially interfere with data collection. Once the positions of the key points are extracted, they are paired together in order to generate line segments (defined by their key point endpoints) that represent the shooter's various body parts. The angles between these body parts are then calculated. The end results are calculated angles for the following joints:

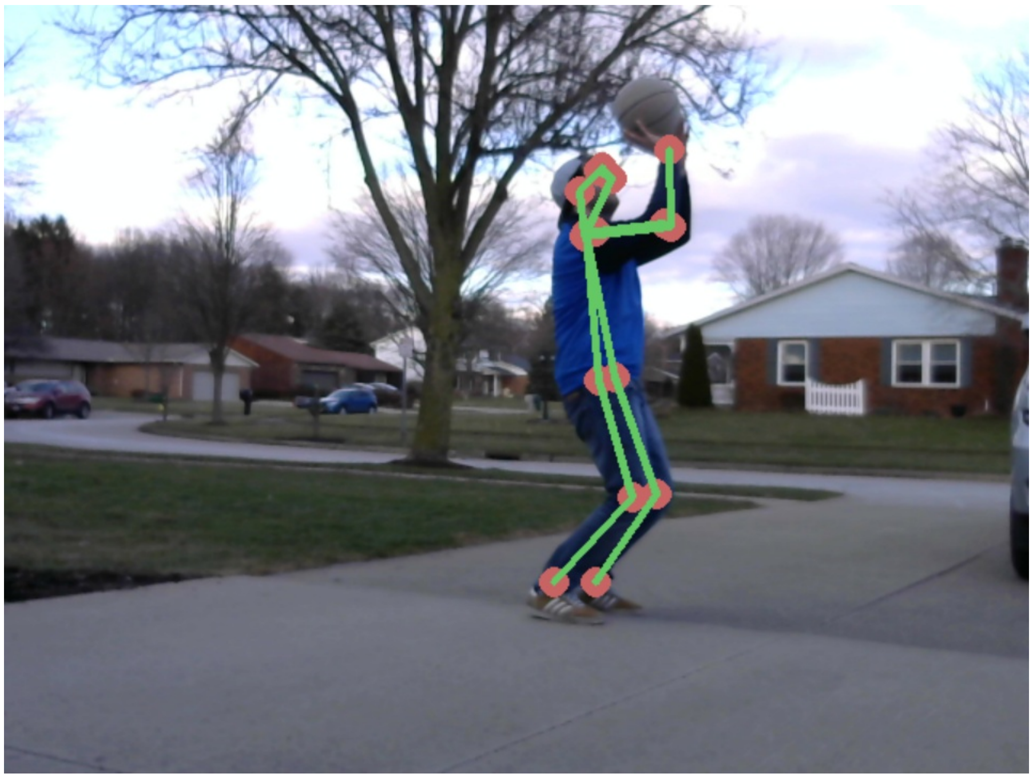


Figure 6.4: A pose estimate from the middle of a free throw.

- the right and left shoulders (angle between shoulder line and neck),
- the right and left arms (angle between shoulder line and the upper arms),
- the right and left elbows,
- the right and left "body" (angle between the neck and the line between the base of the neck and the hip),
- the right and left hips (angle between the line between the base of the neck and hip and the upper leg),
- and the right and left knees.

Using these angles we can describe a shooter's body pose regardless of their size and positioning (as long as the camera remains in the same place). These angles are the values we want to record.

This thread handles the user input needed to control the program. If the user presses the 's' key, the program will give the user the next 30 frames to get into their starting pose. Once the countdown completes, the pose (angles between the joints as described above) is recorded after checking its validity with the user. Once this is done, these recorded angles are checked against at every new frame to see if the user has started a free throw. The check is done by verifying that each angle is within a certain threshold of the reference pose.

If a shot is detected, 90 frames are recorded. These angles pulled from the detected poses are saved into a csv file. At the end of the processing, the program checks with the user to see if the shot was actually a shot. If it was not, the collected data is thrown out, and the detection threshold is multiplied by 0.9 to reduce it slightly. If the shot was valid, the user is prompted to enter whether or not it was made. This value is added to the saved file.

The end result is a csv for every valid recorded shot, labeled with whether or not the shot was made. Using these data points, we can analyze whether or not the user's free throw form has an effect on whether or not they make the shot. In the next section, we discuss the process used to investigate this.

6.2 Data Processing and Clustering

Using the data from a single player shooting multiple free throws, we hope to determine whether or not the player's form is indicative of the outcome of the shot. To examine the relationship between shot form and shot outcome, we decided to cluster the data. Before the data can be fed to a clustering algorithm, it must be first cleaned up and normalized so that each sequence of angles represents the same part of the free throw. Our cleaning and clustering are done in a single Python script that simply requires a directory full of shot data. It is likely that it will need tuned for different recording angles and shooters. Our code is located in Appendix C.1.

6.2.1 Cleaning the Data

Our cleaning of the data was two fold: smoothing and what we refer to as normalization. For us, normalization is the process of getting the data from each shot to begin at the same point in the shooting motion. The data in its original condition is quite rough. Because the estimation of the key points is done from scratch at each frame, similar frames can yield varying results. Essentially, this means that we see slight changes in position even if the shooter's overall pose remains relatively constant. In order to combat this effect, we decided to smooth our data. We do this using the Lowess method. This method yields a smoother curve that has a much clearer shape (Figure 6.5).

The Lowess (not to be confused with the related Loess) method for smoothing is a method for fitting a smooth curve to points in such a way as to be resistant to outliers. This is done by fitting a low-degree polynomial at each data point using weighted least squares. The weights in this situation are allocated so that the further away a point is from the point we are fitting at, the

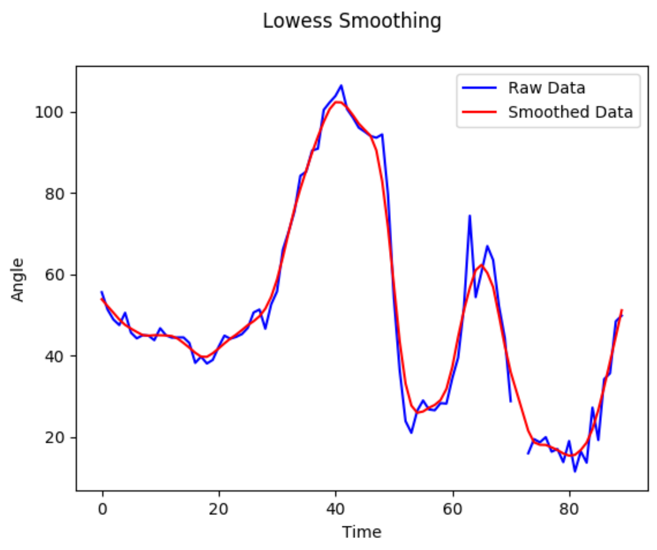


Figure 6.5: An example of data before and after being smoothed using Lowess smoothing.

lower its weight. The end result is a curve that is highly resistant to outliers. This method, as expected, is considered computationally complex [11]. However, for our purposes this is does not matter.

Lowess is an acronym meaning "locally weighted smoothing scatterplots". The highly related Loess method uses a similar methodology, but skips on the weighting process. Essentially, this means that the Loess method does not do anything to limit the effect of outliers [11]. For our usage, this resistance to outliers is important, because our data is prone to being noisy, so we naturally use the Lowess method.

Normalizing the data, which ensures that the start of the sequence represents the start of the shot, is a matter of a simply filtering the data. We looked for a threshold on one of the angles that can be used to determine when a shot has been started. Similarly, we looked at thresholds on the right knee,

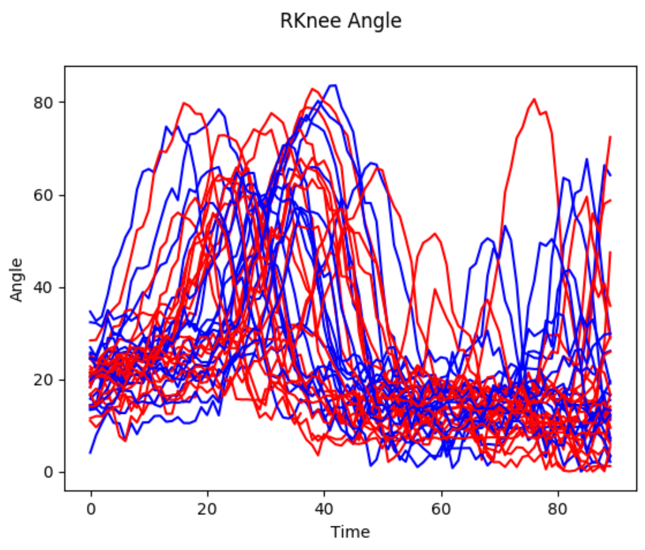


Figure 6.6: An example of right knee angle data before being normalized. Blue and red lines indicate makes and misses respectively.

right elbow, and the right arm. We found that applying a threshold to the right arm angle at 120 degrees yielded decent results (Figure 6.6 and 6.7). Essentially, the first point in the data where the right arm angle exceeds 120 degrees is marked as the start of the shot. The following 40 data points are kept as the “shot”. This makes our data consistently represent the same part of the motion in the free throw. It is possible that a better method to determine the start of a shot exists, but our threshold method yields consistent results that worked for our test data.

Also worth mentioning, we throw out any shots that have any NaN values in the angles we are interested in. We do this because NaN values indicate that the pose estimation failed. We could theoretically fill in the data using surrounding values, however we believe that this could yield inaccurate data. To see how we filtered our data see Appendix C.1.

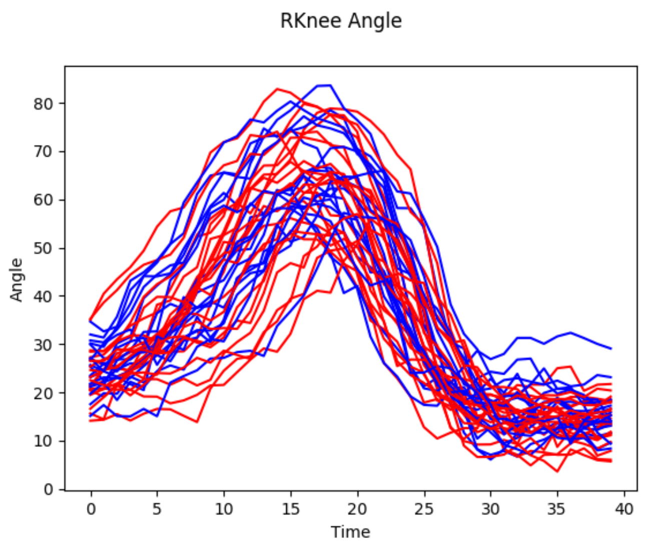


Figure 6.7: The right knee angle data from Figure 6.6 after being normalized. Blue and red lines indicate makes and misses respectively.

6.2.2 Clustering

Having the data in workable format, all that is left to do is look at which shots are most closely related. We hope to find that the misses and makes form distinct groupings, indicating that shot form has a clear effect on shot outcome. The best way to go about finding these groupings is through the use of clustering.

Clustering, also called segmentation analysis or unsupervised classification, is a method of grouping objects in such a way that objects in the same group are similar to each other, but differ from objects in other groups. The groups, unlike in classification, are not defined. The goal is to discover these groups to gain useful insights into the data [19].

The clustering process requires a few important concepts. The first important concept is a data point. This data point is composed of a set of scalar attributes that describe it [19]. In our application, a data point is a single free throw, and the attributes are the sequential angles.

The next concept we need is that of similarities/distances. These are the measures we use to describe how "close" our data points are to each other. Similarity is a measure of how alike to data points are, with higher values indicating higher alikeness. Distance can be see as a sort of reciprocal to similarity, with higher distances indicating more dissimilar data points [19]. Later in this section we will describe the distance measures we settled on.

Another important piece of clustering is the cluster itself. Generally speaking our clusters are sets of objects that are closely related, have small mutual distances or high mutual similarity, and are clearly distinct from the compliment of the cluster [19]. Based on these requirements, we can have a large variety of clusters, with a wide array of shapes and sizes.

Now that we have the key pieces of clustering defined, we can take a look at the general clustering process. The process typically consists of four steps: data representation, modeling, optimization, and validation. The data representation phase determines what sort of clusters are looked for, for example they could be compact (circular/spherical) or extended (allowing for larger and more varied clusters). This can be determined by the type of data, or be arbitrarily chosen based on the desired results. The modeling phase determines the criteria that separates the clusters, usually a similarity or distance measure. The optimization phase calculates this measure and finds the closest clusters. The validation phase evaluates the results of the clustering [19].

Conventional clustering algorithms can be divided into two categories: hierarchical and partitional. Hierarchical clustering algorithms generate clusters through merging smaller clusters or splitting larger clusters. Algorithms that start with a single large cluster and divide it are known as divisive clusters, and algorithms that start with many single item clusters and merge them are known as agglomerative algorithms. Partitional algorithms operate differently: they generate a single level, non-overlapping partition of the data [19]. We will not dig any further into how these algorithms work, as many algorithms exist, there are many ways to implement them. For further reading on clustering algorithms, and clustering in general, see [19].

Our data is temporal, which means we have fewer (useful) methods at our disposal. It is worth noting that we could derive attributes from the data to cluster with instead of using our time series of angles, but we found no logical attributes to generate, and thus we decided to move forward with clustering on the temporal data. We decided to use hierarchical clustering, due its ease of use, open ended nature (can generate a varying number of clusters based on the data), and the fact that visualizing the clusters can be done using dendograms (Figure 6.8).

We decided to use the SciPy implementation of hierarchical clustering. We used the agglomerative variant, with the the "nearest point algorithm" [5]. For our purposes, it was much harder to determine what distance/similarity measure we should use.

Ideally, our distance measurement would take more than simple distance between points in the sequence into account: we want a measurement that looks at the overall *shape* of the sequences. We ended up, after looking into the

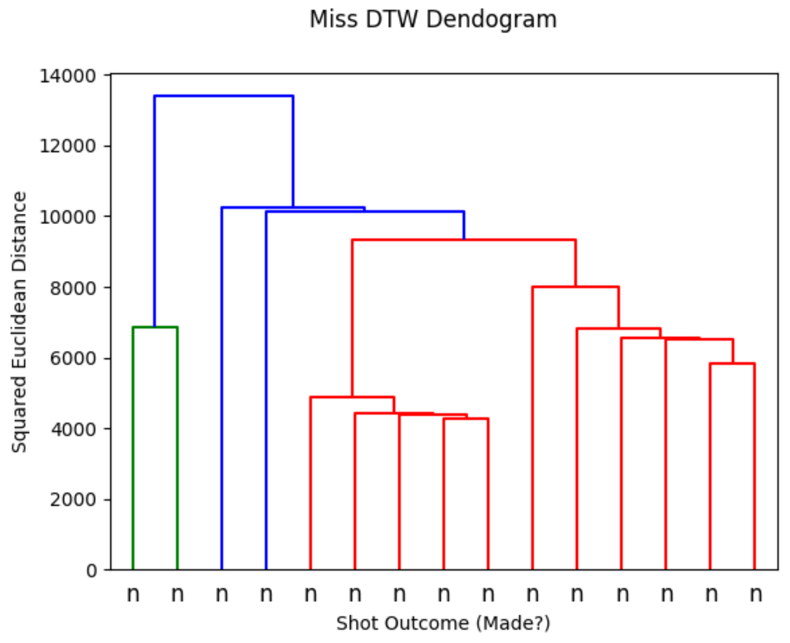


Figure 6.8: An example dendogram of shot data, using the dynamic time warping distance measure. This dendogram contains only missed shots. The different colors represent different clusters generated.

myriad of available distance measures, applying two: cosine distance and squared Euclidean distance combined with dynamic time warping.

Cosine distance is a similarity measure between two vectors. Cosine distance, because it looks at the entirety of the vector rather than individual points, is better suited to pick up on more general trends. Cosine distance, between two vectors u and v is defined as:

$$Cos(u,v) = \frac{<u,v>}{||u||*||v||}$$

where $< u, v >$ is the inner product of u and v and $||u||$ is the normal of vector u [19].

The other method we looked at used dynamic time warping as preprocessing, and then squared Euclidean distance. Dynamic time warping allows for acceleration and deceleration on the time dimension. This helps handle data that may not move at the exact same rate, and sequences that are not the same length. The general idea is to extend sequences by repeating elements, and then calculating the distances between these extended sequences. To read further on this concept see [19].

Squared Euclidean distance is the square of the Euclidean distance between two vectors. Euclidean distance is the most common spatial distance, used, and one of the most intuitive. The Euclidean distance between vectors u and v is defined as:

$$d_{Euclidean}(x, y) = ((x - y)(x - y)^T)^{\frac{1}{2}}$$

and thus the squared Euclidean distance is defined as:

$$d_{Euclidean^2}(x, y) = (x - y)(x - y)^T$$

[19]. In the end, we found the cosine distance generated more distinct clusters, however, in the next chapter we will present the clusters generated by both distance measures.

Our collected data includes a lot of information that can be used to cluster with. Clustering with all of the data would yield horrible results, so we need to select which pieces we will cluster with. Of the collected pieces, the first part we decided to throw out was all the data points from the left side. We decided to record from the right side of the player, as the data from the left side of the player is inaccurate and highly variable due to occlusion from the players body. We can cluster on individual body parts along with concatenated sequences of multiple body parts. To see the code we used for our clustering see Appendix C.1.

Chapter 7

Shot Analysis with Pose Estimation: Results

In order to determine the effect of a player's form on shot quality, we need to collect data. In order for our data to be useful, it must be collected in a single session, with the camera remaining in the same place. We collected data for around half an hour with an amateur (due to convenience). We collected 40 shots, 18 makes and 22 misses. More than 40 shots were taken in that time, however, due to some errors in the data collection, not every shot was recorded. Also, because of null values recorded we ended up with 29 usable shots.

With our data sample collected, the next step is to clean the data. The data is smoothed and then normalized so that each sequence represents the same portion of the free throw. At this point, our data is ready to be clustered. In order to get an idea of the differences of the outcomes of shots, we averaged the makes and misses (Figures 7.4, 7.5, and 7.6).

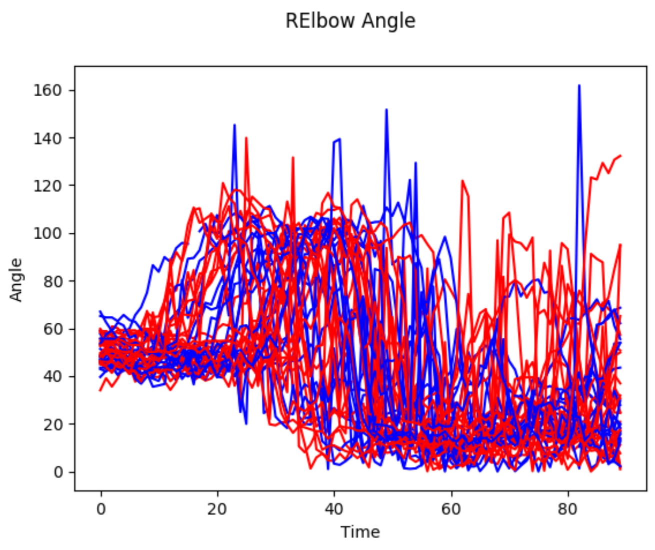


Figure 7.1: The unedited sequence of angles of the right elbow from our data.

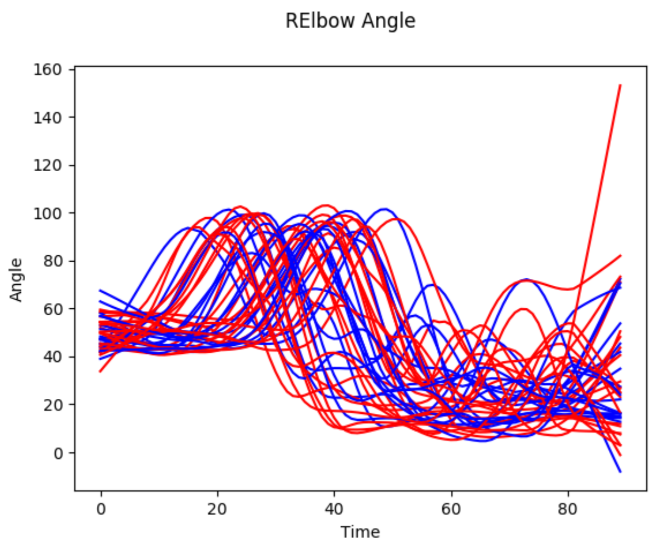


Figure 7.2: The smoothed sequence of angles of the right elbow from our data.

We can see that the average make and the average miss are clearly different, but only slightly. This expected as whether the shot is a make or a miss, the form should be roughly the same. We aim to pick up on the subtle

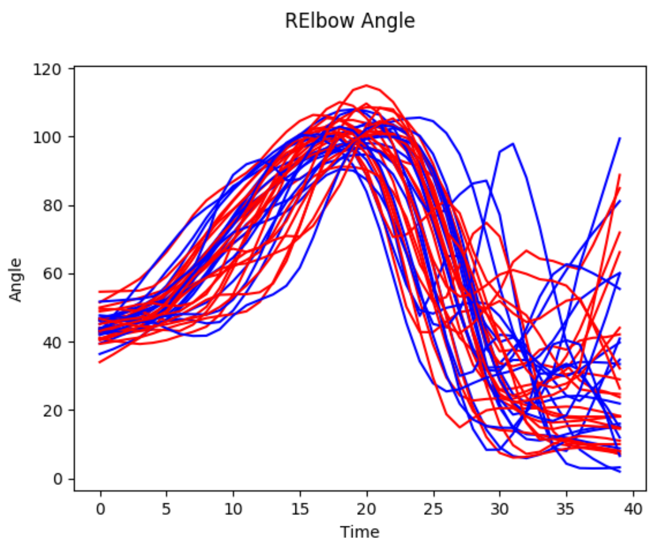


Figure 7.3: The smoothed and normalized sequence of angles of the right elbow from our data.

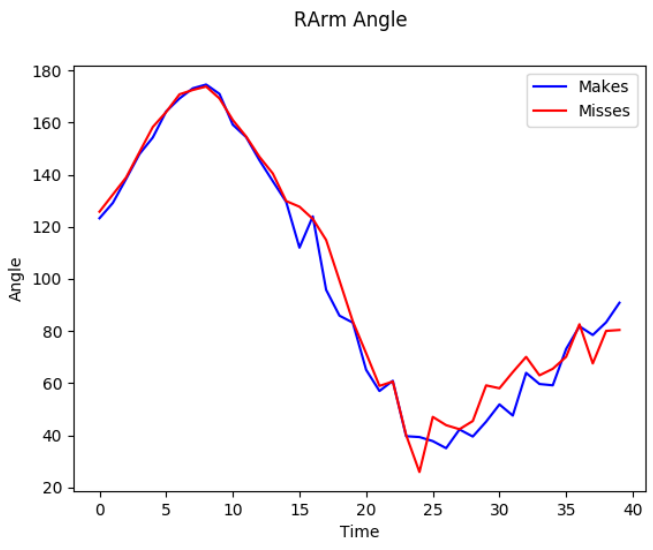


Figure 7.4: The average sequence of right arm angles of the makes/misses from our data.

differences. At this point, we can begin to cluster the data. We clustered the data on the variety of different angle measures, and a handful of

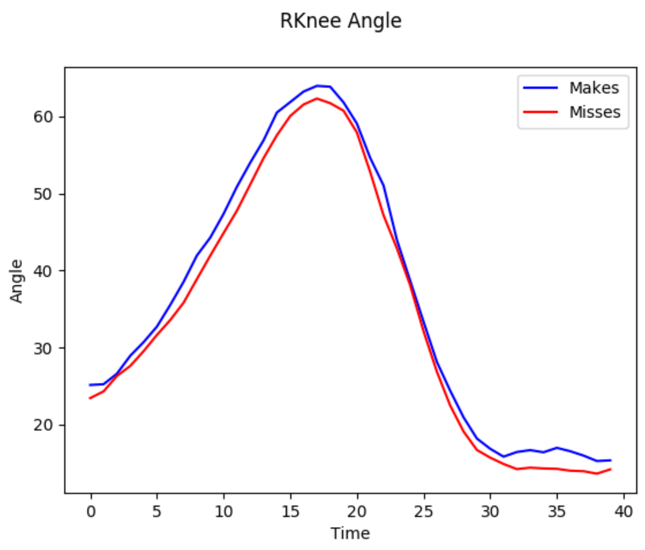


Figure 7.5: The average sequence of right knee angles of the makes/misses from our data.

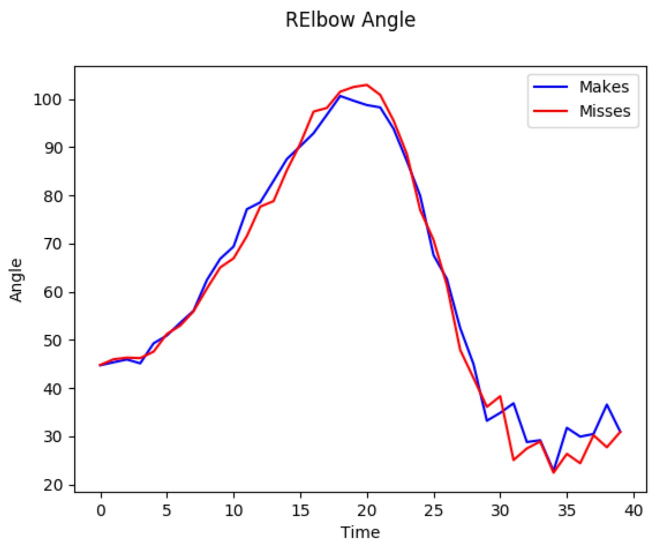


Figure 7.6: The average sequence of right elbow angles of the makes/misses from our data.

concatenations.

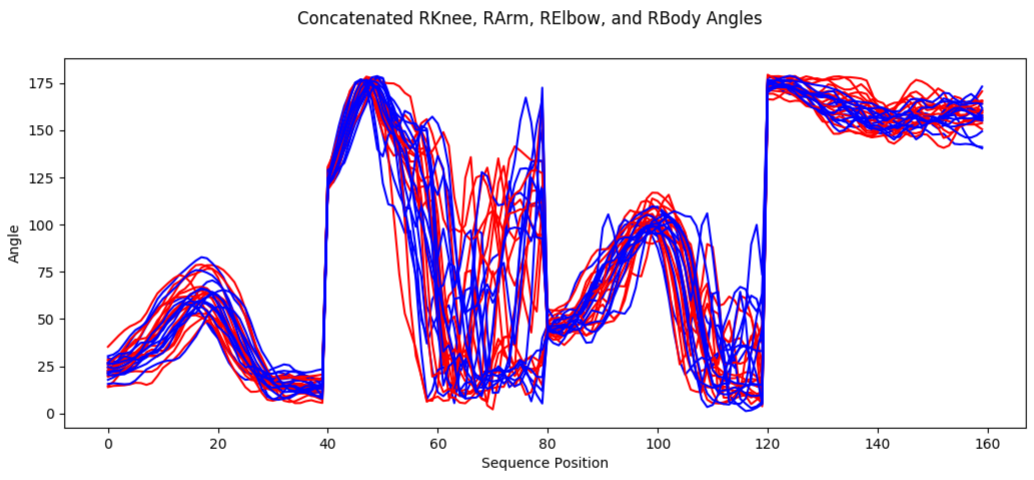


Figure 7.7: The angle data from the right knee, arm, elbow, and body concatenated together for clustering.

We found that concatenating the knee, arm, elbow, and body angles made the most sense (Figure 7.7). This gives the clustering algorithm the maximum amount of information to cluster with, and should theoretically yield the best (or at least most accurate) results.

With this data we run the clustering algorithm three times for both cosine distance and dynamic time warping distance: using all the data, just the makes, and just the misses (Figures 7.8, 7.9, 7.10, 7.11, 7.12, 7.13).

In the dendograms that show both makes and misses together, we can see that the generated clusters are not homogeneous, that is clusters are composed of both makes and misses. This could be due to shot form having little effect on the outcome. This would result in similar shots yielding different outcomes. However, what is more likely, is that this is due to deficiencies in our data, whether it be that we are not picking up on enough detail, we do not

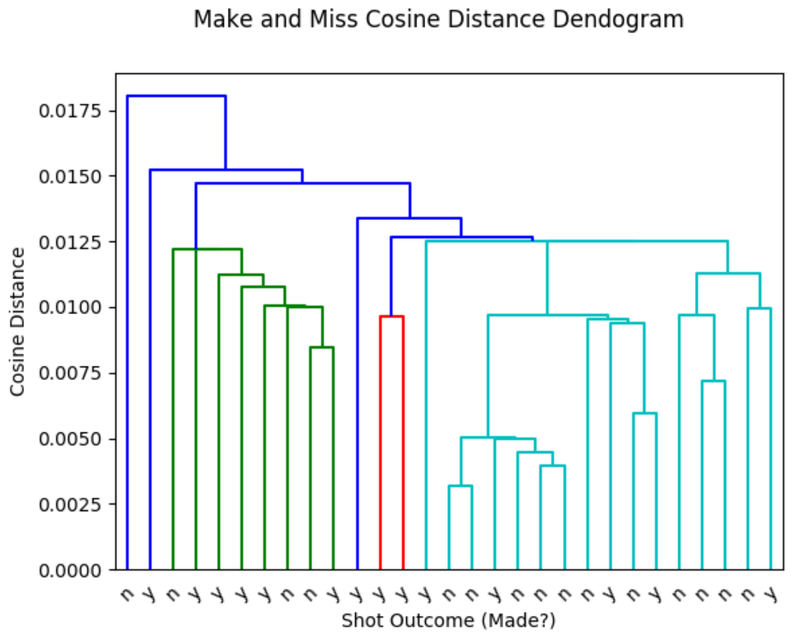


Figure 7.8: Clustering makes and misses together, using cosine distance.

simply have enough data, or we are just looking at the wrong thing. We will detail some of the different things we could have done to improve our data in the next chapter.

Individually, we see that makes are simply not that similar, and that misses, while still not very similar, cluster slightly better. This is counterintuitive, as we expect makes to be the result of good, reproducible form, and misses to be deviations from this form. This could indicate two things: problems with our data, or that there are only a few ways to miss a shot. In this case, the former is much more likely than the latter.

Figures 7.14 and 7.15 show averaged representatives from the generated clusters. We can see that these clusters represent different shot forms, as they are different in every body part used for clustering. However, the utility of

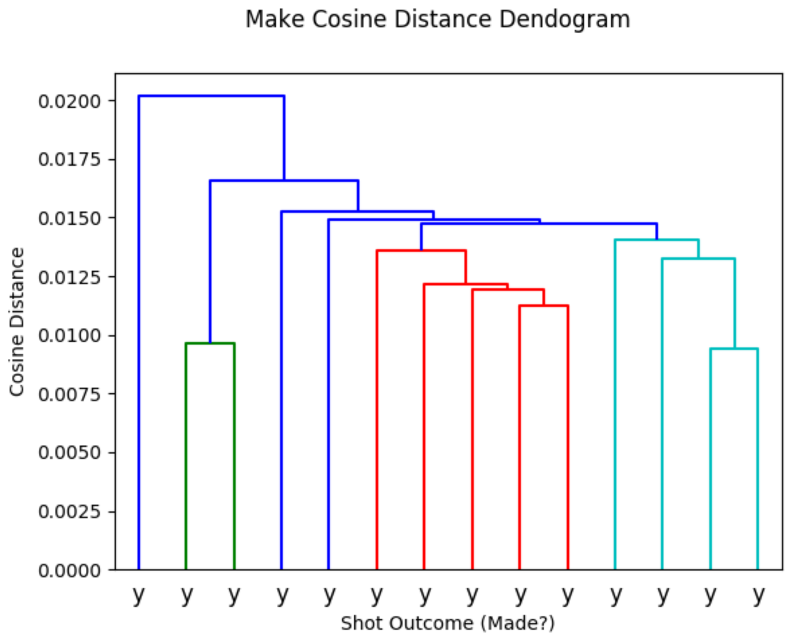


Figure 7.9: Clustering the made shots together, using cosine distance.

these representatives is limited due to cluster size and quality. With more data, especially from a seasoned player, these representatives could be seen as the general forms a missed or made shot can take for the shooter, and can help indicate what to change to avoid missing.

Generally speaking, we cannot make any conclusions using our data, as our results do not indicate strong similarity between made or missed shots.

7.1 Challenges

In this area of our research, we did not face too many issues. The biggest problem we had, as mentioned previously, was our data. There are two issues with the data collected: sample size and our player. We simply did not collect

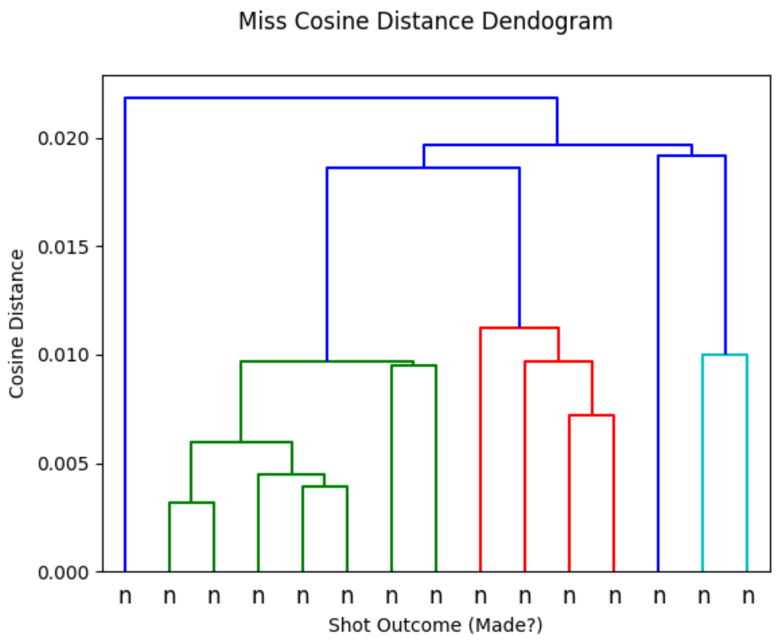


Figure 7.10: Clustering the missed shots together, using cosine distance.

enough data. A larger amount of data would have helped make larger, better clusters. Our player turned out to be a bigger issue than sample size. He is an amateur, so his free throw form is inconsistent. This is problematic because ultimately, we are looking for the intricacies of the player's form that lead to desirable shot trajectories and velocities. Our player's form varied to much between each shot, so these intricacies were impossible to pick up on. A professional, or at least a college of high school player, would likely yield more consistent results. This many not have revealed a clean relationship between form and outcome, but would have allowed us to come to a stronger conclusion.

A more general issue we have with our data is quality. We are collecting angles from a 2D image, this does not yield the best possible data. Depth data

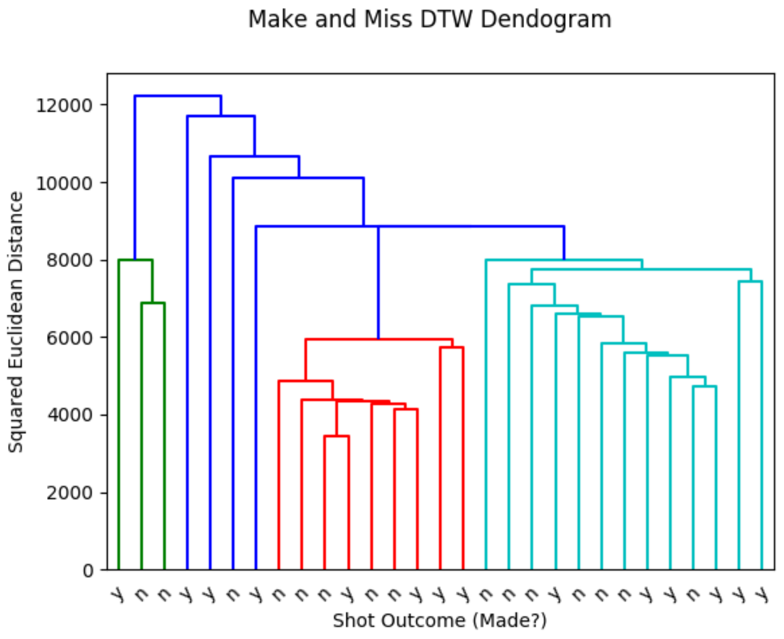


Figure 7.11: Clustering makes and misses together, using dynamic time warping and squared euclidean distance.

would help describe a pose more completely. This would also allow us to reliably use data from the far side of the shooters body. Possible solutions to the problems we described in this section are discussed in the next chapter.

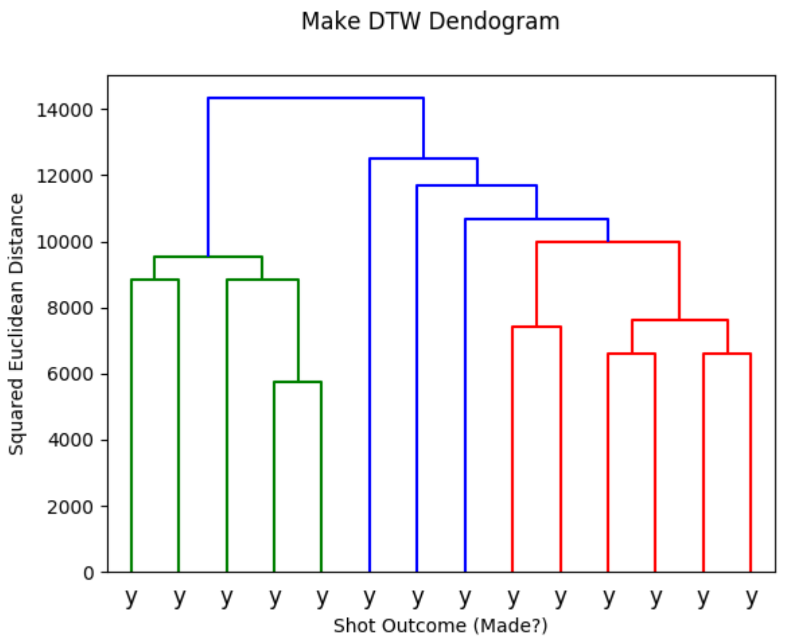


Figure 7.12: Clustering the made shots together, using dynamic time warping and squared euclidean distance.

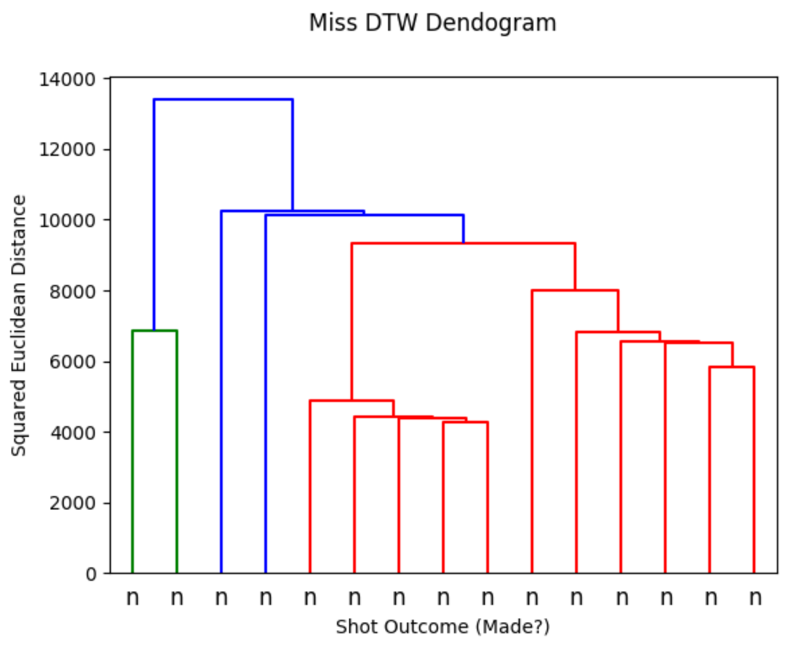


Figure 7.13: Clustering the missed shots together, using dynamic time warping and squared Euclidean distance.

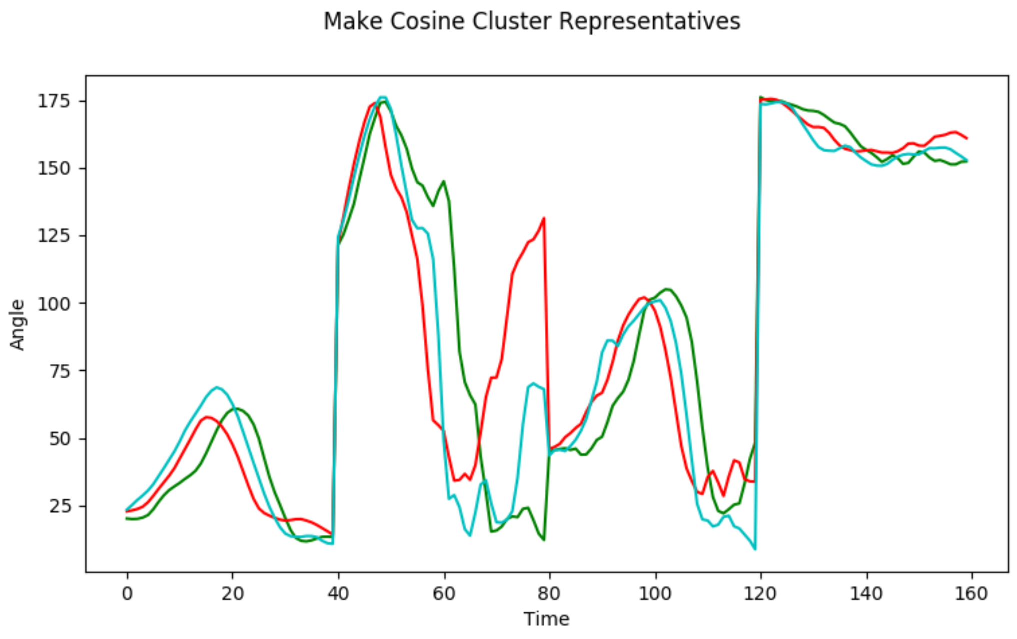


Figure 7.14: The averaged representatives from each cluster generated using just made shots and Cosine distance. The representatives are generated from the cluster of the same color in 7.9.

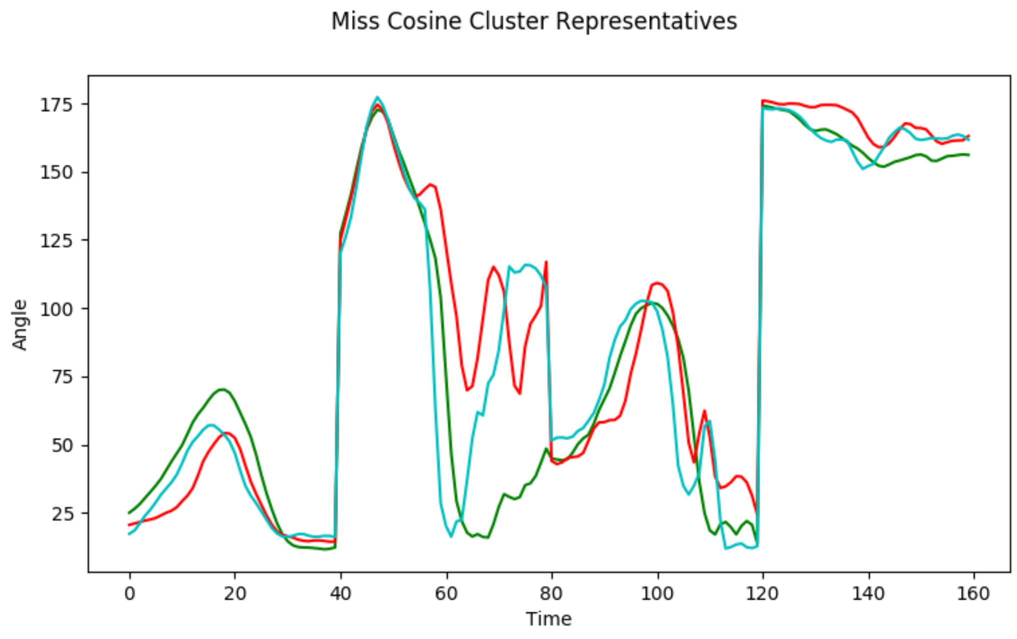


Figure 7.15: The averaged representatives from each cluster generated using just missed shots and cosine distance. The representatives are generated from the cluster of the same color in 7.10.

Chapter 8

Conclusions

In this section, we will offer some concluding remarks on both parts of our project along with suggestions for future work that could be done to remedy some of the issues we encountered, make them irrelevant, improve the performance of our theoretical system, or apply our work to other areas.

8.1 Optical Statistics Collection

As we discussed in the results chapter for our optical statistics collection system, we did not meet our original goals due to numerous issues in building our system. Many of our problems stemmed from budgetary and time issues, and this makes perfect sense: the system we planned to build was incredibly complicated. Similar systems took a few years and millions of dollars to build, so attempting to build one, even as simple and scaled down as ours was, in a less than a year with a budget of $1000 was an extremely ambitious task.

We believe that we can conclude that these types of systems are outside of the grasp of small colleges and high schools for the time being, just due to the resources required. However, we do believe that the building of an open sourced system with a small budget is certainly possible, but it would require a large time investment along with an in-place camera system to build the software from.

8.1.1 Future Work

As mentioned previously, a lot of the problems we faced were due to time and money constraints. Assuming that these two resources were to become irrelevant, there are many changes that we could have made and alternate avenues we could have taken to improve our system. We will discuss these changes here along with possible further areas we could apply or system to.

One of our largest problems turned out to be our choice in camera. As we discussed in a previous chapter, the cameras firmware turned out to cause issues in frame rate and synchronization. A simple solution would be to use different cameras. More expensive cameras, especially non-CCTV cameras would likely have performed better. If money was no object, custom cameras would be the best option. A custom camera, built to our specification, could have a higher frame rate, resolution, and field of view. We could essentially have it built exactly to suit our needs. Another positive of building a custom camera would be that we could implement our own firmware. This would allow use to set the cameras up to synchronize while recording or at least record extremely accurate time stamps that would be accessible outside of the video feed. There are a few downsides to building a custom camera however.

Custom cameras would add quite a bit more work to the project. Writing the firmware alone is much more complicated than writing our simple recording program. Also, custom cameras would likely require more complicated mounting solutions than the CCTV cameras. Despite the extra work, a custom camera setup would have likely improved the outcome of this project considerably.

Another issue we faced was in our computer vision techniques. Our trackers, at least in all of our testing, were underperforming. To analyze an entire game with them would require many hours of user work. This is simply unacceptable and goes against our goal of making the software easy to use. Unlike our issues with the cameras, this problem does not have as clear-cut of a solution. There are a two different avenues that we believe would make the most sense to improve our trackers: writing a custom tracker and using a tracker trained specifically for our application. Writing a custom tracker is no simple task. However, if we wanted a higher performance tracker, it is just about the only option. A tracker that combines techniques from existing trackers would be a good place to start. An easier approach to take would be to use an offline trained tracker, like GOTURN, and train it specifically to track basketball players and basketballs. This would be a good way to improve our application specific performance, and could provide enough of an edge to reduce the user workload.

The last fixable issue we had was our inability to set up our cameras. The solution here is simple: find a place to set up the cameras. We do not believe that there is an alternative path here, the camera system needs to be in place for the rest of the software to be created properly. Ultimately, we should have looked outside of our college for locations to set up the cameras, even if temporary. Finding alternate locations is difficult, as we need a large gym, with a high ceiling. It is possible that we could have worked with the local high school, or another college. If we had a single piece of advice to give to others looking to implement a system like ours, it would be to set up a camera system first, and work from there.

In terms of other areas to apply our system, there are other sports that would be easy to apply such a system to. Volleyball would be the easiest, as the system would already be in place where the volleyball team plays. Slight modifications would be needed to track the ball, but otherwise, everything else would carry over. Another sport it could be applied to, with modification, would be hockey. Hockey has a similar flow to basketball, and most of the system would carry over. There would need to be some changes to track the puck (which could be quite difficult) and to handle the rapid movements of the players. Soccer is another place this system would be applied, however, as it currently works, it would really only work well for indoor soccer. Fundamental changes for camera placement would need to be made to make it work for outdoor arenas. STATS is actively attempting to build such a system for soccer [6], and likely other sports as well.

8.2 Shot Analysis with Pose Estimation

As mentioned in the results section for our shot estimation system, we could not make any conclusions regarding the relationship between shot form and shot outcome. This is due to the quality of our data and the fact that our results did not indicate the existence or nonexistence of such a relationship. The only conclusion we can really come to is that we need better data. Better data could mean a higher quantity and quality of the type of data we looked at, or a different sort of data altogether (like ball trajectory).

8.2.1 Future Work

The primary problem we had turned out to be our data. We determined that the quality of the data was poor due to the player we selected, and that we did not have enough data. To come to a better conclusion, we should have collected a lot more data, and used a professional player. A player with more consistent form would generate much better data, and their consistent form would make it easier to pick up on the subtle differences between makes and misses. Another area in which we could have improved our data would have been to collect pose information from the hands. A lot of the nuance of basketball shot form happens in the wrists and hands. By collecting this information, we would gain insight into the movements of the shooters wrists and fingers.

We could also improve our data by giving more detailed classifications of shot outcome. Instead of a binary make/miss, we could detail where the shot contacted the rim/backboard. This would improve the clustering, as shots that impact in certain areas (the side of the rim for example) can yield different outcomes based on slight differences in trajectory. By classifying this way, we may find that we generate more homogeneous clusters.

Our method for clustering is another area that could likely use improvement. Due to time constraints, we could not fully investigate the plethora of available methods and distance measures. A better method, and a more suitable distance could have yielded better clusters. Additional research into temporal clustering should be done, along with distance measures that are more tailored to time series. For example, creating our own implementation of dynamic time warping would also allow the usage of a larger variety of distance measures.

Another area of future work would be to investigate the other piece of the basketball shot: the ball's trajectory. This information could be derived from video footage by tracking the ball. The ball's initial velocity, launch angle, and starting height could, in theory, be used to reliably determine if a shot was made. This information would also be easy to detect immediately after a shot, so instant feedback could be given.

Appendix A

Recording Software Code

A.1 recorder.h

```
// Colby Jeffries
// recorder.h

// This class is a widget that contains the needed IP entries and buttons to
// start recording.

// Inclusion protection.
#pragma once

// Modules
//    Header
#include "recorder.h"
//    Qt
#include <QWidget>              // Widget parent class for the recorder class.
#include <QLabel>
#include <QLineEdit>            // Editable text box.
#include <QGridLayout>          // Grid Layout manager.
#include <QPushButton>
#include <QApplication>
```

```
#include <QSignalMapper>
//    STD
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <chrono>
#include <vector>
#include <thread>
//    OpenCV
#include "opencv2/opencv.hpp"   // For recording from IP cameras.

// Global Constants
#define FPS 30

// Class that contains a window with all of the necessary pieces to start a
// recording from the cameras. This class is an extension of the Widget Qt
// class. This class generates the window for the recorder, including they
// entry fields for each IP, the test button, and the record button.
class Recorder : public QWidget {

  Q_OBJECT  // Allows usage of Qt signals and slots for this class.

   private:
    QVector <QLabel*> IPLabels;
    QVector <QLineEdit*> IPBoxes;
    std::string IPFile = "lastUsedIPs.txt";
    int cameras;

   public:
    // Constructor. Initializes all elements of the UI.
    Recorder(QWidget *parent = 0, int numCameras = 6);

    // Function for each camera. This function is run for each camera on
    // separate threads. The function grabs a frame from the camera, encodes it,
    // and then saves it.
    static void cameraFunc(std::string ip, std::string outputVid);
    // Reads the file containing the last used IP addresses if it exists.
    // Otherwise, empty strings are set.
    void readIPFile();
    // Clears vectors of widget stuff.
    void deleteStuff();
    // Sets up the widget. Does most of the constructor work. Helper.
    void setUpWidget();

  public slots:
    // Starts recording process for each camera. Initializes the cameras whose
    // IP fields are valid, and then starts their threads.
    void record();
    // Tests the IP for each camera. Sets the entry fields color based on then
    // result of the tests.
    void test();
    // Saves the contents of the IPBoxes, into a file to be loaded next time.
    void saveIPFile();
    // Adds or removes a camera entry.
    void modifyCamera(int add);

};
```

A.2 recorder.cpp

```
// Colby Jeffries
// recorder.cpp

// Implementation of the recorder window class.

// Header.
#include "recorder.h"

// Constructor.
Recorder::Recorder(QWidget *parent, int numCameras) : QWidget(parent) {

  cameras = numCameras;
  setUpWidget();
}

// Clears the widget vectors.
void Recorder::deleteStuff() {

  qDeleteAll(this->findChildren<QWidget*>("", Qt::FindDirectChildrenOnly));
  delete this->layout();

  IPBoxes.clear();
  IPLabels.clear();
}

// Sets up the camera entries, buttons, and connections.
void Recorder::setUpWidget() {
  // Generate a new grid layout.
  QGridLayout *grid = new QGridLayout(this);
  grid->setVerticalSpacing(15);
  grid->setHorizontalSpacing(5);

  // IP Labels, Entries, and Testing Buttons.
  for(int i=0; i<cameras; i++) {
    IPBoxes.append(new QLineEdit(this));
    IPBoxes[i]->setStyleSheet("QLineEdit { background: rgb(250, 250, 150); selection-
    background-color: rgb(250, 250, 250); }");
    grid->addWidget(IPBoxes[i], i, 0, 1, 1);

    QString ipl = "IP Address %1";
    IPLabels.append(new QLabel(ipl.arg(i+1), this));
    grid->addWidget(IPLabels[i], i, 1, 1, 1);
    IPLabels[i]->setAlignment(Qt::AlignRight | Qt::AlignVCenter);
  }

  // Set IPs frome file.
  readIPFile();

```

```
// Test Button
QPushButton *testButton = new QPushButton("Test IPs", this);
grid->addWidget(testButton, cameras, 0, 1, 1);
connect(testButton, SIGNAL(clicked()), SLOT(test()));

// Record Buttons
QPushButton *recordButton = new QPushButton("Record", this);
grid->addWidget(recordButton, cameras, 1, 1, 1);
connect(recordButton, SIGNAL(clicked()), SLOT(record()));

// Add/Remove Camera buttons
QPushButton *plusButton = new QPushButton("+", this);
grid->addWidget(plusButton, cameras + 1, 0, 1, 1);
QPushButton *minusButton = new QPushButton("-", this);
grid->addWidget(minusButton, cameras + 1, 1, 1, 1);
QSignalMapper *signalMapper = new QSignalMapper(this);
if (cameras > 1) {
  connect(minusButton, SIGNAL(clicked()), signalMapper, SLOT(map()));
  signalMapper->setMapping(minusButton, 0);
}
if (cameras < 6) {
  connect(plusButton, SIGNAL(clicked()), signalMapper, SLOT(map()));
  signalMapper->setMapping(plusButton, 1);
}

connect(signalMapper, SIGNAL(mapped(int)), this, SLOT(modifyCamera(int)));

// Register save function when recording is started.
connect(testButton, SIGNAL(clicked()), SLOT(saveIPFile()));
connect(recordButton, SIGNAL(clicked()), SLOT(saveIPFile()));
connect(qApp, SIGNAL(aboutToQuit()), this, SLOT(saveIPFile()));

// Set a larger width.
this->setFixedWidth(300);
this->setFixedHeight((cameras + 2) * 40);

// Set the layout.
```

```
  setLayout(grid);
}

// Adds or removes a camera entry.
void Recorder::modifyCamera(int add) {
  if (add) cameras++;
  else cameras--;
  deleteStuff();
  setUpWidget();
}

// Tests all of the entered IPs to see if they work.
void Recorder::test() {
  for(int i=0; i<cameras; i++) {
    QString ip = IPBoxes[i]->text();
    cv::VideoCapture capture = cv::VideoCapture(ip.toStdString());
    if (capture.isOpened()) {
      // Sets the background to green.
      IPBoxes[i]->setStyleSheet("QLineEdit { background: rgb(150, 250, 150); selection
    -background-color: rgb(250, 250, 250); }");
    }
    else {
      // Sets the background to red.
      IPBoxes[i]->setStyleSheet("QLineEdit { background: rgb(250, 150, 150); selection
    -background-color: rgb(250, 250, 250); }");
    }

    capture.release();
  }
}

// Starts the recording process.
void Recorder::record() {
  std::vector<std::thread> threads;
  for(int i=0; i<cameras; i++) {
    // If the IP is not empty.
    if(IPBoxes[i]->text().toStdString() != "") {
      std::string ip = "rtsp://admin:BlackYellow1@" + IPBoxes[i]->text().toStdString()
     + ":554/cam/realmonitor?channel=1&subtype=0";
      std::string outputVid = "Camera" + IPBoxes[i]->text().right(3).toStdString() + "
    .avi";
      threads.push_back(std::thread(cameraFunc, ip, outputVid));
      threads[i].join();
    }
  }
}

// Thread function for each camera.
void Recorder::cameraFunc(std::string ip, std::string outputVid) {
  // Initialize the capture and writer.
  cv::VideoCapture capture = cv::VideoCapture(ip);
  // cv::VideoCapture capture = cv::VideoCapture(0);   // DEBUG
  if (capture.isOpened()) {
    cv::VideoWriter video;
    video.open(outputVid, cv::VideoWriter::fourcc('X', '2', '6', '4'), capture.get(
    CV_CAP_PROP_FPS), cv::Size(capture.get(CV_CAP_PROP_FRAME_WIDTH), capture.get(
    CV_CAP_PROP_FRAME_HEIGHT)), true);

    // Initialize variables that are needed throughout recording.
    auto start = std::chrono::high_resolution_clock::now();
    auto timing_start = std::chrono::high_resolution_clock::now();
    int frameCount = 0;
    int totalFrames = 0;
    int skipFrameCount = 0;
    int framesToSkip = 0; // How many frames to skip.
    cv::Mat frame;

    // Record for two hours.
    while(totalFrames < (120 * 60 * FPS)) {
      // Record a frame.
      if (skipFrameCount < 1) {
        frameCount++;
        totalFrames++;
        skipFrameCount++;
        capture >> frame;
        video.write(frame);
      }

      // Skip frame(s).
      else {
        if (skipFrameCount < (1 + framesToSkip)) {
          capture >> frame;
          skipFrameCount++;
        }
        else {
          skipFrameCount = 0;
        }
      }

      // Calculate framerate, and print it every 10 seconds.
      auto currTime = std::chrono::duration<double, std::milli>(std::chrono::
    high_resolution_clock::now() - start).count();
      if (currTime > 10000) {
        std::cout << "Framerate: " << frameCount / 10.0 << std::endl;
        frameCount = 0;
        start = std::chrono::high_resolution_clock::now();
      }
    }
  }

  capture.release();
}

// Reads the IP file list and sets the contents of the IP boxes.
void Recorder::readIPFile() {
  for (int i=0; i<cameras; i++) {
    IPBoxes[i]->setText("");
  }
  std::ifstream in(IPFile);
  if (in.is_open()) {
    std::string line;
    int i = 0;
    while((std::getline(in, line)) && (i<cameras)) {
      IPBoxes[i]->setText(QString::fromStdString(line));
      i++;
    }
  }
}

// Saves the contents of the IP boxes to a file.
void Recorder::saveIPFile() {
  std::ofstream out(IPFile);
  for (int i=0; i<cameras; i++) {
    out << IPBoxes[i]->text().toStdString() << std::endl;
  }
  out.close();
}
```

A.3 main.cpp

```
// Colby Jeffries
// main.cpp

// This is the main driver for the ScotViewer recorder.

// Modules
#include <QApplication>   // Qt Application framework/
#include "recorder.h"     // Recorder window header file.

// Main Driver
int main(int argc, char *argv[]) {

  // Initialize the application.
  QApplication app(argc, argv);

  // Generate a 'recorder' window. This window contains the IP address entries
  // along with the buttons needed to start the recording.
  Recorder window;

  // Title and show the window.
  window.setWindowTitle("Scotviewer Recorder v1.0");
  window.show();

  // Start the execution loop for the Qt application.
  return app.exec();
}
```

Appendix B

Free Throw Analyzer Code

B.1 FreeThrowAnalyzer.cpp

```
// Colby Jeffries
// FreeThrowAnalyzer.cpp

// This program is used to collect pose data from a basketball player shooting
// free throws. This program can be modified for other actions of similar
// nature.

// Includes
#include <gflags/gflags.h>
#include <math.h>
#include <stdio.h>
#include <fstream>
#include <iostream>
#include <thread>
#include <sys/stat.h>
#include <ctime>
// OpenCV
#include <opencv2/opencv.hpp>
// OpenPose dependencies
```

```
#include <openpose/core/headers.hpp>
#include <openpose/filestream/headers.hpp>
#include <openpose/gui/headers.hpp>
#include <openpose/pose/headers.hpp>
#include <openpose/utilities/headers.hpp>
#include <openpose/pose/poseParameters.hpp>
// Thread-safe Queue
#include "threadQueue.hpp"

// GLOG/GFLAGS Definitions for OpenPose.
DEFINE_int32(logging_level, 3, "The logging level. Don't touch.");
DEFINE_string(model_pose, "COCO", "Using COCO for 18 keypoints on body.");
DEFINE_string(model_folder, "models/", "Folder path for models.");
DEFINE_string(net_resolution, "-1x368", "Speed v Accuracy. May need fine tuning.");
DEFINE_string(output_resolution, "-1x-1", "Uses input image size.");
DEFINE_int32(num_gpu_start, 0, "May need fine tuning.");
DEFINE_int32(scale_number, 1, "May need fine tuning.");
DEFINE_double(scale_gap, 0.3, "Scale gap between scales. May need fine tuning.");

DEFINE_int32(camera, 0, "Camera to use.");
DEFINE_int32(frameCount, 90, "How many frames to record per shot.");

// Global variables to be accessed by the threads.
std::mutex rf_m;
std::mutex rf2p_m;
int recordingFrames = 0;
int recordingFramesToProcess = 0;
Queue<cv::Mat> frame_queue;

// This function checks one pose against another, with a certain interval
// error.
bool checkAgainstStart(std::map<std::string, float> start,
                        std::map<std::string, float> current,
                        float interval) {
  if(start.empty()) {
    return false;
  }

  int counter = 0;

  for(auto it_s = start.cbegin(), end_s = start.cend(), it_c = current.cbegin(),
      end_c = current.cend(); it_s != end_s || it_c != end_c;) {
    if ((!std::isnan(it_s->second) && (!std::isnan(it_c->second)))) {
      if ((it_s->second > it_c->second + interval) ||
          (it_s->second < it_c->second - interval)) {
        counter++;
      }
    }
    else {
      counter++;
    }

    if (counter > 2) {
      return false;
    }
    std::cout << counter << std::endl;

    it_s++; it_c++;
  }

  return true;
}

// Finds the inner angle between two lines.
float findAngle(std::tuple<std::tuple<float, float>, std::tuple<float, float>> l1,
                std::tuple<std::tuple<float, float>, std::tuple<float, float>> l2) {
  float a = std::get<0>(std::get<0>(l1)) - std::get<0>(std::get<1>(l1));
  float b = std::get<1>(std::get<0>(l1)) - std::get<1>(std::get<1>(l1));
  float c = std::get<0>(std::get<0>(l2)) - std::get<0>(std::get<1>(l2));
  float d = std::get<1>(std::get<0>(l2)) - std::get<1>(std::get<1>(l2));
  return acos(((a * c) + (b * d))/(pow(pow(a, 2) + pow(b, 2), 0.5) * pow(pow(c, 2) +
    pow(d, 2), 0.5))) * (180/M_PI);
}

```

```
// This is our thread function that grabs frames from the camera. It has two
// states. In the first the next available frame is automatically put at the
// front of the queue. In the second it grabs every frame and puts them on the
// queue.
void grabFrames() {
  cv::Mat inputImage;
  int frameSwitch = 0;
  cv::VideoCapture cap = cv::VideoCapture(FLAGS_camera);
  std::cout << "FPS: " << cap.get(CV_CAP_PROP_FPS) << std::endl;
  std::cout << "Width: " << cap.get(CV_CAP_PROP_FRAME_WIDTH) << std::endl;
  std::cout << "Height: " << cap.get(CV_CAP_PROP_FRAME_HEIGHT) << std::endl;
  while(1) {
    cap >> inputImage;
    // Records every frame in a shot.
    if (recordingFrames > 0) {
      std::cout << "Recording frame " << (FLAGS_frameCount - recordingFrames) << std::
    endl;
      frame_queue.push(inputImage.clone());
      rf_m.lock();
      recordingFrames--;
      rf_m.unlock();
    }
    // Puts the newest frame at the top of the queue.
    else if ((recordingFramesToProcess == 0)) {
      while(frame_queue.get_size() >= 1) {
        frame_queue.pop();
      }
      frame_queue.push(inputImage.clone());
    }
  }
}

// This is our processing thread function. It estimates poses for frames,
// controls key inputs, and also saves pose data for recorded shots.
void processFrames() {
  op::log("Web Cam Test", op::Priority::High);
```

```
op::check(0 <= FLAGS_logging_level && FLAGS_logging_level <= 255, "Wrong
    logging_level value.", __LINE__, __FUNCTION__, __FILE__);
op::ConfigureLog::setPriorityThreshold((op::Priority)FLAGS_logging_level);
op::log("", op::Priority::Low, __LINE__, __FUNCTION__, __FILE__);
const auto outputSize = op::flagsToPoint(FLAGS_output_resolution, "-1x-1");
const auto netInputSize = op::flagsToPoint(FLAGS_net_resolution, "-1x368");
const auto poseModel = op::flagsToPoseModel(FLAGS_model_pose);
if (FLAGS_scale_gap <= 0. && FLAGS_scale_number > 1)
    op::error("Incompatible flag configuration: scale_gap must be greater than 0 or
    scale_number = 1.", __LINE__, __FUNCTION__, __FILE__);

op::log("", op::Priority::Low, __LINE__, __FUNCTION__, __FILE__);
op::ScaleAndSizeExtractor scaleAndSizeExtractor(netInputSize, outputSize,
    FLAGS_scale_number, FLAGS_scale_gap);
op::CvMatToOpInput cvMatToOpInput;
op::CvMatToOpOutput cvMatToOpOutput;
op::PoseExtractorCaffe poseExtractorCaffe{poseModel, FLAGS_model_folder,
    FLAGS_num_gpu_start};
op::OpOutputToCvMat opOutputToCvMat;
poseExtractorCaffe.initializationOnThread();

const auto& poseBodyPartMappingCoco = op::getPoseBodyPartMapping(op::PoseModel::
    COCO_18);
const auto& poseBodyPartPairsCoco =  op::getPosePartPairs(op::PoseModel::BODY_18);

std::ofstream shotFile;
cv::Mat inputImage;
cv::Mat outputImage;
char key;
bool poseEstimation = true;
float interval = 7.5;
int poseCountdown = -1;
std::string fileName;
std::string videoFileName;

std::map<std::string, std::tuple<float, float>> keyPointPositions;
```

```
std::map<std::string, std::tuple<std::tuple<float, float>, std::tuple<float, float
  >>> bodySegments;
std::map<std::string, float> jointAngles;
std::map<std::string, float> startPose;

cv::namedWindow("Monitoring Window");
while(key != 'q') {
  inputImage = frame_queue.pop();
  cv::resize(inputImage, inputImage, cv::Size(), 1.0, 1.0);
  const op::Point<int> imageSize{inputImage.cols, inputImage.rows};
  std::vector<double> scaleInputToNetInputs;
  std::vector<op::Point<int>> netInputSizes;
  double scaleInputToOutput;
  int numberPeopleDetected = 0;
  op::Point<int> outputResolution;
  std::tie(scaleInputToNetInputs, netInputSizes, scaleInputToOutput,
  outputResolution) = scaleAndSizeExtractor.extract(imageSize);
  const auto netInputArray = cvMatToOpInput.createArray(inputImage,
  scaleInputToNetInputs, netInputSizes);
  auto outputArray = cvMatToOpOutput.createArray(inputImage, scaleInputToOutput,
  outputResolution);
  if (poseEstimation) {
    poseExtractorCaffe.forwardPass(netInputArray, imageSize, scaleInputToNetInputs);
    const auto poseKeypoints = poseExtractorCaffe.getPoseKeypoints();
    outputImage = opOutputToCvMat.formatToCvMat(outputArray);
    numberPeopleDetected = poseKeypoints.getSize(0);
    const auto numberBodyParts = poseKeypoints.getSize(1);
    if (numberPeopleDetected > 0) {
      for(int i=0; i < numberBodyParts; i++) {
        const auto baseIndex = poseKeypoints.getSize(2)*(0*numberBodyParts + i);
        const auto x1 = poseKeypoints[baseIndex];
        const auto y1 = poseKeypoints[baseIndex + 1];
        std::tuple<float, float> temp(x1, y1);
        keyPointPositions[poseBodyPartMappingCoco.at(i)] = temp;
        if ((x1 > 0) && (y1 > 0)) {
          cv::circle(outputImage, cv::Point(x1, y1), 5, cv::Scalar(100, 100, 200),
  8);
```

```
            }
        }

        for(int i=0; i < poseBodyPartPairsCoco.size(); i = i + 2) {
            std::tuple<std::tuple<float, float>, std::tuple<float, float>> temp(
    keyPointPositions[poseBodyPartMappingCoco.at(poseBodyPartPairsCoco[i])],
    keyPointPositions[poseBodyPartMappingCoco.at(poseBodyPartPairsCoco[i + 1])]);
            bodySegments[poseBodyPartMappingCoco.at(poseBodyPartPairsCoco[i]) + " - " +
    poseBodyPartMappingCoco.at(poseBodyPartPairsCoco[i + 1])] = temp;
        }

        jointAngles["LShoulder"] = findAngle(bodySegments["Neck - LShoulder"],
    bodySegments["Neck - Nose"]);
        jointAngles["RShoulder"] = findAngle(bodySegments["Neck - RShoulder"],
    bodySegments["Neck - Nose"]);
        jointAngles["LArm"] = findAngle(bodySegments["Neck - LShoulder"], bodySegments
    ["LShoulder - LElbow"]);
        jointAngles["RArm"] = findAngle(bodySegments["Neck - RShoulder"], bodySegments
    ["RShoulder - RElbow"]);
        jointAngles["LElbow"] = findAngle(bodySegments["LShoulder - LElbow"],
    bodySegments["LElbow - LWrist"]);
        jointAngles["RElbow"] = findAngle(bodySegments["RShoulder - RElbow"],
    bodySegments["RElbow - RWrist"]);
        jointAngles["RBody"] = findAngle(bodySegments["Neck - Nose"], bodySegments["
    Neck - RHip"]);
        jointAngles["LBody"] = findAngle(bodySegments["Neck - Nose"], bodySegments["
    Neck - LHip"]);
        jointAngles["RHip"] = findAngle(bodySegments["Neck - RHip"], bodySegments["
    RHip - RKnee"]);
        jointAngles["LHip"] = findAngle(bodySegments["Neck - LHip"], bodySegments["
    LHip - LKnee"]);
        jointAngles["RKnee"] = findAngle(bodySegments["RHip - RKnee"], bodySegments["
    RKnee - RAnkle"]);
        jointAngles["LKnee"] = findAngle(bodySegments["LHip - LKnee"], bodySegments["
    LKnee - LAnkle"]);

        for(auto const& item : bodySegments) {
```

```
      float x1 = std::get<0>(std::get<0>(item.second));
      float y1 = std::get<1>(std::get<0>(item.second));
      float x2 = std::get<0>(std::get<1>(item.second));
      float y2 = std::get<1>(std::get<1>(item.second));
      if ((x1 > 0) && (y1 > 0) && (x2 > 0) && (y2 > 0)) {
        cv::line(outputImage, cv::Point(x1, y1), cv::Point(x2, y2), cv::Scalar
(100, 200, 100), 3);
      }
    }

    // Processes frame from a recorded shot and saves the data.
    if (recordingFramesToProcess > 0) {
      std::cout << "Processing frame " + std::to_string(FLAGS_frameCount -
recordingFramesToProcess) << std::endl;
      for(auto const& item : jointAngles) {
        shotFile << item.second << " ";
      }
      shotFile << std::endl;
      rf2p_m.lock();
      recordingFramesToProcess--;
      rf2p_m.unlock();
      // At the end of the shot check if it is valid and if it was made.
      if (recordingFramesToProcess == 0) {
        std::cout << "Was that a shot? [y/n] : ";
        char shot;
        std::cin >> shot;
        while((shot != 'y') && (shot != 'n')) {
          std::cout << "Was that a shot? [y/n] : ";
          std::cin >> shot;
        }
        // Get rid of invalid shot data, reduces the margin of error on the
        // pose check.
        if (shot == 'n') {
          shotFile.close();
          remove(fileName.c_str());
          std::cout << "Reducing accuracy interval on checking shot pose." << std
::endl;
```

```
              std::cout << "If shots continue to be incorrectly detected, reset shot pose." << std::endl;
              interval = interval * 0.9;
            }
            else {
              std::cout << "Make? [y/n] : ";
              char made;
              std::cin >> made;
              while((made != 'y') && (made != 'n')) {
                std::cout << "Make? [y/n] : ";
                std::cin >> made;
              }
              shotFile << made << std::endl;
              shotFile.close();
              // shotVideo.release();
            }
          }
        }
        else {
          std::cout << std::endl;
          for(auto const& item : jointAngles) {
            std::cout << item.first << " : " << item.second << std::endl;
          }
        }
      }
    }
    else {
      outputImage = opOutputToCvMat.formatToCvMat(outputArray);
    }

    // If the pose is close to the start pose, record a shot.
    if ((poseCountdown == -1) && (recordingFramesToProcess == 0) && (checkAgainstStart(startPose, jointAngles, interval))) {
      rf2p_m.lock();
      recordingFramesToProcess = FLAGS_frameCount;
      rf2p_m.unlock();
      rf_m.lock();
      recordingFrames = FLAGS_frameCount;
      rf_m.unlock();
      auto t = std::time(nullptr);
      auto tm = *std::localtime(&t);
      std::ostringstream oss;
      oss << std::put_time(&tm, "%d-%m-%y---%H-%M-%S");
      fileName = "SavedShots/" + oss.str() + ".txt";
      videoFileName =  "SavedShotsVideos/" + oss.str() + ".avi";
      shotFile.open(fileName);
      for(auto const& item : jointAngles) {
        shotFile << item.first << " ";
      }
      shotFile << std::endl;
    }

    cv::resize(outputImage, outputImage, cv::Size(), 1.5, 1.5);
    cv::imshow("Monitoring Window", outputImage);
    key = cv::waitKey(1);
    // Start pose countdown.
    if ((key == 's')) {
      poseCountdown = 30;
    }
    // Toggles pose estimation.
    else if ((key == 'e')) {
      if(poseEstimation) {
        poseEstimation = false;
      }
      else {
        poseEstimation = true;
      }
    }

    // At the end of a start pose timer, check if the pose is correct.
    if (poseCountdown == 0) {
      std::cout << "Correct? [y/n] : ";
      char check;
      std::cin >> check;
      while((check != 'y') && (check != 'n')) {
        std::cout << "Correct? [y/n] : ";
        std::cin >> check;
      }
      if (check == 'y') {
        startPose = jointAngles;
      }
      poseCountdown--;
    }
    else if (poseCountdown > 0) {
      poseCountdown--;
    }
  }
}

// Main driver, initializes both the frame grabbing thread and the processing
// thread.
int main(int argc, char *argv[]) {
  gflags::ParseCommandLineFlags(&argc, &argv, true);

  std::thread prod(grabFrames);
  std::thread cons(processFrames);

  prod.join();
  cons.join();
}
```

Appendix C

Clustering Code

C.1 FreeThrowClustering.py

```
# FreeThrowClustering.py
# Colby Jeffries

# Modules
import pandas as pd
import os
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.nonparametric.smoothers_lowess import lowess
from scipy.cluster.hierarchy import linkage, dendrogram, fcluster
from scipy.spatial.distance import pdist, squareform
import mlpy

# Generate a matrix of distances between shots using dynamic time warping
# with squared Euclidean distance.
def dtw_matrix(m):
    return_m = []
    for i in range(len(m)):
        temp = []
        for j in range(len(m)):
            temp.append(mlpy.dtw_std(m[i], m[j], squared=True))

        return_m.append(temp)

    return return_m

shotDirectory = '02-21 FreeThrows'
columns = ['LArm', 'RArm', 'LBody', 'RBody', 'LElbow', 'RElbow',
            'LHip', 'RHip', 'LKnee', 'RKnee', 'LShoulder', 'RShoulder']
colorlist = ['g', 'r', 'c', 'm', 'y', 'k']

complete_data = []
combined_data = []
make_data = []
miss_data = []

# Iterate through all of the shots in the directory.
for file in os.listdir(shotDirectory):
    df = pd.read_csv(shotDirectory + '/' + file, sep=' ')
    columns = list(df)[:-1]
    # Ensure that the data is of the proper length.
    if (len(df.index) == 91):
        startPoint = 0
        # Whether or not the shot was made.
        make = df['LArm'][90]
        # Grab all of the relevant data.
        RKnee = df['RKnee'].values.tolist()[:-1]
        RArm = df['RArm'].values.tolist()[:-1]
        RElbow = df['RElbow'].values.tolist()[:-1]
        RBody = df['RBody'].values.tolist()[:-1]
        # Find the first point in the shot where the RArm angle
        # is greater than 120. This will be the start point
        # of the shot.
        for i in range(len(RArm)):
            if (RArm[i] > 120):
                startPoint = i
                break

        # Ensure that there are no null values.
        check = False
        for i in range(len(RKnee)):
            if pd.isnull(RKnee[i]):
                check = True

            if pd.isnull(RArm[i]):
                check = True

            if pd.isnull(RElbow[i]):
                check = True

            if pd.isnull(RBody[i]):
                check = True


        if not check:
            # Normalize the data so it is from the start of
            # the shot through the next 40 frames.
            RKnee = RKnee[startPoint:startPoint+40]
            RArm = RArm[startPoint:startPoint+40]
            RElbow = RElbow[startPoint:startPoint+40]
            RBody = RBody[startPoint:startPoint+40]
            # Smooth the data using Lowess smoothing.
            x = np.arange(len(RKnee))
            smoothed = lowess(RKnee, x, is_sorted=True, frac=0.1, it=0)
            RKnee = list(smoothed[:,1])
            smoothed = lowess(RArm, x, is_sorted=True, frac=0.1, it=0)
            RArm = list(smoothed[:,1])
            smoothed = lowess(RElbow, x, is_sorted=True, frac=0.1, it=0)
            RElbow = list(smoothed[:,1])
            smoothed = lowess(RBody, x, is_sorted=True, frac=0.1, it=0)
            RBody = list(smoothed[:,1])
            # Concatenate all of the data into a single
            # long time series.
            full_data = []
            full_data.extend(RKnee)
            full_data.extend(RArm)
            full_data.extend(RElbow)
            full_data.extend(RBody)
            if make == 'y':
                make_data.append(full_data)
            else:
                miss_data.append(full_data)

            combined_data.append(full_data)
            complete_data.append((full_data, make))

# Generate figure of full concatenated data.
plt.figure('Concatenated RKnee, RArm, RElbow, and RBody Angles')
plt.suptitle('Concatenated RKnee, RArm, RElbow, and RBody Angles')
plt.xlabel('Sequence Position')
plt.ylabel('Angle')
for i in complete_data:
    x = np.arange(len(i[0]))
    if i[1] == 'y':
        plt.plot(x, i[0], 'b')
    else:
        plt.plot(x, i[0], 'r')

# Calculate cosine distances between shots.
data_dist = pdist(combined_data, 'cosine')
# Generate clustering data.
link_mat = linkage(data_dist)

# Generate figure, plot the dendogram and show it.
plt.figure('Make and Miss Cosine Distance Dendogram')
plt.suptitle('Make and Miss Cosine Distance Dendogram')
dendrogram(link_mat, labels=[x[1] for x in complete_data])
plt.xlabel('Shot Outcome (Made?)')
plt.ylabel('Cosine Distance')
plt.show(block=False)
```

```

data_dist = pdist(make_data, 'cosine')
link_mat = linkage(data_dist)
# The distance threshold is that used by the dendogram.
clusters = fcluster(link_mat, 0.7*np.max(link_mat[:,2]), 'distance')
cluster_data = []
for i in range(max(clusters)):
    cluster_data.append([])

for i in range(len(clusters)):
    cluster_data[clusters[i] - 1].append(make_data[i])

plt.figure('Make Cosine Cluster Representatives')
plt.suptitle('Make Cosine Cluster Representatives')
plt.xlabel('Time')
plt.ylabel('Angle')
for i in range(len(cluster_data)):
    if len(cluster_data[i]) > 1:
        temp = [sum(x) for x in zip(*cluster_data[i])]
        temp = [x / len(cluster_data[i]) for x in temp]
        plt.plot(np.arange(len(temp)), temp, colorlist[i % len(colorlist)])

plt.show(block=False)

plt.figure('Make Cosine Distance Dendogram')
plt.suptitle('Make Cosine Distance Dendogram')
dendrogram(link_mat, labels=['y' for x in range(len(make_data))])
plt.xlabel('Shot Outcome (Made?)')
plt.ylabel('Cosine Distance')
plt.show(block=False)

data_dist = pdist(miss_data, 'cosine')
link_mat = linkage(data_dist)
clusters = fcluster(link_mat, 0.7*np.max(link_mat[:,2]), 'distance')
cluster_data = []
for i in range(max(clusters)):
    cluster_data.append([])
```

```

for i in range(len(clusters)):
    cluster_data[clusters[i] - 1].append(miss_data[i])

plt.figure('Miss Cosine Cluster Representatives')
plt.suptitle('Miss Cosine Cluster Representatives')
plt.xlabel('Time')
plt.ylabel('Angle')
for i in range(len(cluster_data)):
    if len(cluster_data[i]) > 1:
        temp = [sum(x) for x in zip(*cluster_data[i])]
        temp = [x / len(cluster_data[i]) for x in temp]
        plt.plot(np.arange(len(temp)), temp, colorlist[i % len(colorlist)])

plt.show(block=False)

plt.figure('Miss Cosine Distance Dendogram')
plt.suptitle('Miss Cosine Distance Dendogram')
dendrogram(link_mat, labels=['n' for x in range(len(miss_data))])
plt.xlabel('Shot Outcome (Made?)')
plt.ylabel('Cosine Distance')
plt.show(block=False)

# Calculate distances between shots using dynamic time warping
# and squared Euclidean distance.
data_dist = squareform(dtw_matrix(combined_data))
link_mat = linkage(data_dist)

plt.figure('Make and Miss DTW Dendogram')
plt.suptitle('Make and Miss DTW Dendogram')
dendrogram(link_mat, labels=[x[1] for x in complete_data])
plt.xlabel('Shot Outcome (Made?)')
plt.ylabel('Squared Euclidean Distance')
plt.show(block=False)

data_dist = squareform(dtw_matrix(make_data))
link_mat = linkage(data_dist)
```

```

plt.figure('Make DTW Dendogram')
plt.suptitle('Make DTW Dendogram')
dendrogram(link_mat, labels=['y' for x in range(len(make_data))])
plt.xlabel('Shot Outcome (Made?)')
plt.ylabel('Squared Euclidean Distance')
plt.show(block=False)

data_dist = squareform(dtw_matrix(miss_data))
link_mat = linkage(data_dist)

plt.figure('Miss DTW Dendogram')
plt.suptitle('Miss DTW Dendogram')
dendrogram(link_mat, labels=['n' for x in range(len(miss_data))])
plt.xlabel('Shot Outcome (Made?)')
plt.ylabel('Squared Euclidean Distance')
plt.show()
```

Bibliography

[1] Amcrest technologies. https://amcrest.ca/.

[2] Nba stats. http://stats.nba.com.

[3] Opencv 3.1.0 documentation. https://docs.opencv.org/3.1.0/.

[4] Openpose.
https://github.com/CMU-Perceptual-Computing-Lab/openpose.

[5] Scipy documentation.
https://docs.scipy.org/doc/scipy/reference/index.html.

[6] Stats llc. https://www.stats.com/.

[7] Stats sportvu basketball player tracking.
https://www.stats.com/sportvu-basketball/.

[8] Tesseract documentation.
https://github.com/tesseract-ocr/tesseract.

[9] Wikimedia commons. `https://commons.wikimedia.org/wiki/Main_Page`.

[10] Zoneminder. `https://www.zoneminder.com/`.

[11] Lowess. In David Nelson, editor, *The Penguin Dictonary of Mathematics*. Penguin, London, Uk, 4th edition, 2008.

[12] Benjamin C. Alamar. *Sports Analytics: A Guide for Coaches, Managers, and Other Decision Makers*. Columbia University Press, New York, USA, 2013.

[13] Dana H. Ballard and Christopher M. Brown. *Computer Vision*. Prentice-Hall, Englewood Cliffs, USA, 1982.

[14] Simone Bassis, Anna Esposito, and Francesco Carlo Morabito. *Recent Advances of Neural Network Models and Applications Proceedings of the 23rd Workshop of the Italian Neural Networks Society (SIREN), May 23-25, Vietri sul Mare, Salerno, Italy*. Springer International Publishing, Cham, Switzerland, 2014.

[15] Jay M. Bennett. Baseball. In Jay M. Bennett, editor, *Sports in Statistics*, chapter 2, pages 25–64. Arnold Applications of Statistics, London, UK, 1998.

[16] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

[17] Arindam Chaudhuri, Krupa Mandaviya, Pratixa Badelia, and Soumya K Ghosh. *Optical character recognition systems for different languages with soft computing*. Springer, Cham, Switzerland, 2017.

[18] Yoav Freund and Robert E. Schapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 14:771, 1999.

[19] Guojun Gan, Chaoqun Ma, and Jianhong Wu. *Data Clustering: Theory, Algorithms, and Applications*. ASA SIAM, Philadelphia and Alexandria, USA, 2007.

[20] Bill Gerrard. Is the moneyball approach transferable to complex invasion team sports? *Internation Journal of Sport Finance*, 2:214–230, 2007.

[21] Helmut Grabner, Michael Grabner, and Horst Bischof. Real-time tracking via on-line boosting. *BMVC*, 1:47–56, 2006.

[22] David Held, Sebastian Thrun, and Silvio Savarese. Learning to track at 100 fps with deep regression networks. *European Conference on Computer Vision*, 2016.

[23] Jo£o F. Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. High-speed tracking with kernelized correlation filters. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 2015.

[24] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Forward-backward error: Automatic detection of tracking failures. *Internation Conference on Pattern Recognition*, 2010.

[25] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *Internation Conference on Pattern Recognition*, 6(1), 2011.

[26] Jonah Keri. *The Extra 2Baseball Team from Worst to First*. ESPN Books, New York, USA, 2011.

[27] Oliver Kramer. *Genetic Algorithm Essentials*. Springer International Publishing, Cham, Switzerland, 2017.

[28] Michael Lewis. *Moneyball: The Art of Winning an Unfair Game*. W. W. Norton and Company, New York, USA, 2003.

[29] Bernard Marr. A short history of machine learning – every manager should read. February 2016.

[30] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science, New York, USA, 1997.

[31] Tomas Simon, Hanbyul Joo, Iain Matthews, and Yaser Sheikh. Hand keypoint detection in single images using multiview bootstrapping. In *CVPR*, 2017.

[32] Hal S. Stern. American football. In Jay M. Bennett, editor, *Sports in Statistics*, chapter 1, pages 3–24. Arnold Applications of Statistics, London, UK, 1998.

[33] Richard Szeliski. *Computer Vision Algorithms and Applications*. Springer, London, UK, 2011.

[34] Robert L. Wardrop. Basketball. In Jay M. Bennett, editor, *Sports in Statistics*, chapter 3, pages 65–82. Arnold Applications of Statistics, London, UK, 1998.

[35] Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. Convolutional pose machines. In *CVPR*, 2016.

[36] Steven Wu and Luke Bornn. Modeling offensive player movement in professional basketball. *Peerj Preprints*, 2017.