NTNU – Trondheim
Norwegian University of
Science and Technology

Two-hand, camera-based gesture recognition for SoundDream

Henrik Hjorthen Støren

Master of Science in Computer Science
Submission date: July 2015
Supervisor: Asbjørn Thomassen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Hand gesture recognition is the task of having a machine recognize the hand gestures made by a human. In this thesis the main focus has been to research AI methods for gesture recognition. I investigate machine learning methods and image processing techniques to see if they are suited for hand gesture recognition with a color camera. I have used a PlayStation Eye camera, and written two different programs that use images captured by it to recognize and distinguish between 6 different static gestures. One of the programs uses only image processing techniques to recognize the gestures, while the other uses image processing to construct a feature vector, and then uses the KNN algorithm to predict the gesture in the image. The thesis is a proof of concept that will show the results of both programs and compare the two different approaches. I will also give a comparison between my approaches and what other researchers have done.

Sammendrag

Gjenkjenning av håndgester er å få en maskin til å gjenkjenne gester som gjøres av et menneske. Denne rapporten har som hovedfokus å undersøke metoder i kunstig intelligens som brukes til å gjenkjenne gester. Jeg undersøker maskinlæringsteknikker og bildebehandlingsteknikker for å se hvilke som er egnet for å gjenkjenne gester med et fargekamera. Jeg har brukt et PlayStation Eye kamera, og skrevet to forskjellige programmer som bruker bilder fra kameraet for å gjenkjenne og skille mellom 6 ulike, statiske gester. Et av programmene bruker kun bildebehandlingsteknikker for å gjenkjenne gester, mens det andre bruker bildebehandling for å bygge opp en vektor med verdier, som så brukes i KNN-algoritmen for å gjette gesten i bildet. Prosjektet er et ”proof of concept” som viser resultatene fra begge programmene og sammenligner de to fremgangsmåtene. Jeg kommer i tillegg til å sammenligne mine fremgangsmåter med hva andre forskere har gjort.

Preface

This report is submitted to the Norwegian University of Science and Technology in fulfillment of the requirements for master thesis.

This work has been performed at the Department of Computer and Information Science, NTNU, with Amanuensis Asbjørn Thomassen as the supervisor.

Acknowledgements

I would like to thank Asbjørn Thomassen for his work as my supervisor, and Tore Ommedal for inspiring me with his system for conferences. I am also grateful for all the help given by Håvard Wormdal Høiby and Sondre Lefsaker on how to use LaTeX. Finally I would give an extra thanks to Håvard Wormdal Høiby for proofreading my thesis.

Problem Description

The main task will be to recognize different hand gestures by using a color camera, and to compare different approaches that achieve this. To accomplish that, an evaluation of image processing and machine learning methods will be essential.

Table of Contents

List of Figures

List of Tables

Abbreviations

HCI	=	Human Computer Interaction
FPS	=	Frames Per Second
AI	=	Artificial Intelligence
KNN	=	K-Nearest Neighbours
SVM	=	Support Vector Machine
HMM	=	Hidden Markov Model
NB	=	Naive Bayes
EM	=	ExpectationMaximization
CART	=	Classification and Regression Trees
ANN	=	Artificial Neural Network
SIFT	=	Scale-Invariant Feature Transform

Chapter 1

Introduction

In this chapter I give an introduction to the thesis. I start with some background and motivation for the project, and move on to the goal and my research questions. After that I present the structure of the rest of this thesis.

1.1 Background and Motivation

Imagine that you did not need a display of knobs and buttons, or even a touch-screen to control your device, but that you could do so simply by moving your hands in front of a camera. This is one of the topics being researched in the field of Human Computer Interaction (HCI). HCI is an ever growing field of study in computer science, and in this thesis I will explore how hand gestures can be used to control a device.

The inspiration for the project came from a friend of my supervisor, Tore Ommedal, who wanted to build a system for conferences. The system would consist of two components, each containing a camera, a speaker, a microphone, and a LED light. Every participant in a conference would have the system mounted on the back of his/her chair, having one component on either side of his/her head. The cameras would be used to capture frames above - and in front - of the user's shoulders, where he would perform hand gestures to control the system. Typical commands would include controlling the volume of the speakers, or turning noise cancellation on or off. This system is called SoundDream, and an image of how a prototype is shown in Figure 1.1. In addition to being used for conferences, this system could also be used to control other systems. Controlling programs used for design, like Photoshop, could be done in a more intuitive way than using the mouse and keyboard. Another application is user interaction with games. Sony, Microsoft, and Nintendo have done extensive work already on how users can interact more intuitively with their consoles.

Having done some work previously with the Microsoft Kinect v2 I was already in the mindset of using gestures for controlling a system. This work has made me eager to explore the topic further. I have also done quite some work with image processing in previous

projects and I had hopes to incorporate my knowledge on the topic into this project, while still learning something new.

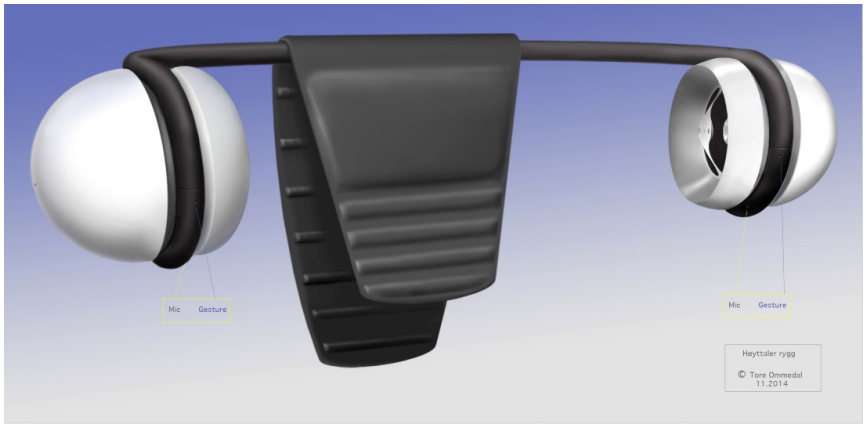

Figure 1.1: Prototype drawing of gesture reconition system, as proposed by Tore Ommedal.

1.2 Goal and Research Questions

Goal: *Compare methods in Artificial Intelligence to see which is best suited for hand gesture recognition.*

There are lots of methods in the field of Artificial Intelligence (AI) that could be used for hand gesture recognition. Throughout this thesis I will discuss a number of methods within the fields of image processing and machine learning. I will present programs that make use of the most promising of these methods in order to compare them. When I had formulated the goal for my thesis, there were a few questions that arose. Following are the 4 research questions that define the work done in this thesis.

Research Question 1: *Which image processing techniques are best suited for hand gesture recognition?*

Research Question 2: *Which machine learning algorithms are best suited for hand gesture recognition?*

Research Question 3: *How should the different approaches be compared and evaluated?*

Research Question 4: *How does the performance of a purely image processing system compare to the performance of the best machine learning systems when it comes to hand gesture recognition?*

1.3 Research method

In this project I have written two programs that takes as input a query image and gives a predicted gesture as output. The first program is written using only image processing techniques to recognize the hand gestures. The second is an implementation of the K-Nearest Neighbours algorithm, where image processing is used to construct the feature vector. These two approaches were chosen because others have had great experiences with them [20, 5]. By taking the practical approach and writing the programs and doing experiments, I get insight to the various problems one might face - problems that might have been overlooked by taking a more analytic approach. By predicting the gesture from images where the gesture is known, I can easily measure the performance of the programs and in order to validate correctness. In my experiments I have used images with a set of gestures shown in Figure 3.2, that the systems should be able to recognize, and experimented with altering some variables to increase the performance. The experimentation will be discussed further in Chapter 4.

1.4 Structure

This thesis will describe the project in its entirety. The introduction is already given in this chapter. In chapter 2, I will give necessary background theory on various topics. In chapter 3, I will explain in more detail what hand gesture recognition is, followed by what other researchers have done, and then I will go through the architecture of my two programs. Chapter 4 will describe the experiments done to answer the research questions given in Section 1.2. I first describe the different experiments that was conducted, and then present the results of the experiments. In chapter 5, I will evaluate the systems and the two approaches based on my experiments, and give answers to the research questions and a discussion on how I achieved the goal of this thesis. In Chapter 6, I state the contributions I have made to the field, suggest some future work and give the conclusion of this thesis.

Chapter 2

Background Theory

This chapter first gives some details about the camera used to capture images. These images are the input data used by the programs I have written. Then necessary background theory is covered. The two fields of study the reader needs insight into in order to understand the following chapters are image processing and machine learning. These fields are the main focus of this chapter. In the section dedicated to image processing I describe the image processing techniques used in my research. In the section covering AI and machine learning, I first clarify the difference between the two terms, and then discuss data mining and the K-Nearest Neighbours algorithm as subsections of AI and machine learning.

2.1 PlayStation Eye

Through my own experience with the Kinect, my first thought was that *it* would be very well suited for this project. These thoughts were based on how well the Kinect can recognize different body parts of a user and it already has built-in functions to distinguish between a few gestures. However, it is quite expensive and also quite big, which makes it nonviable for the project. My supervisor suggested I use the PlayStation Eye, having supervised another project a few years back where the students had used the same camera. The camera is quite small(80mm 55mm 65mm) and known for its high frame rate and low price.This makes it a good choice for my project. It is capable of capturing 60 fps at a 640x480 pixel resolution, and 120 fps at a 320x240 pixel resolution. Having a high frame rate will allow the user to perform- and the program to interpret, several gestures per second which makes the program feel more responsive. A picture of the camera can be seen in Figure 2.1. The PlayStation Eye is intended by Sony to only be used with a PlayStation. Therefore, Sony themselves has not released a driver or any other support for the camera to be used with a computer. However, a driver for this purpose is provided by CodeLaboratories [1].

2.2 Image Processing

In short, image processing is to apply mathematical operations on an image to achieve some goal. This goal might be that you have a lot of noise in your image which you would like to remove. In that case applying a smoothing filter [19] would be one solution. An example of this can be seen in Figure 2.2. Other more advanced tasks include finding contours in an image, or finding the center of mass of an oddly shaped figure.

In my implementation I have utilized several image processing techniques. I have used the image processing library OpenCV [2], because it is easy to use and contains methods that does exactly what I needed for my programs. In the following subsections I will explain some of the techniques used and why they are useful for hand gesture recognition.

Figure 2.1: The PlayStation Eye camera.

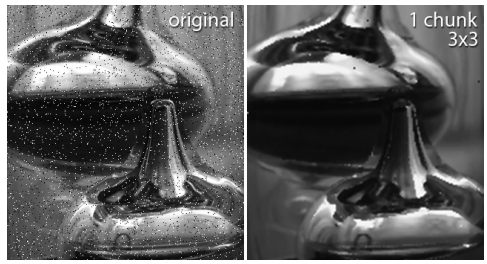

Figure 2.2: Example of smoothing an image to remove noise.

2.2.1 Segmentation

Image segmentation is one of the oldest and most widely studied problems in image processing [19]. The goal of image segmentation is to divide an image into regions or segments. Dividing the image is done based on some criterion. This could be to consider the value of the pixels in the image, be it gray scale or color, or it could be something more advanced like considering texture in the image. The resulting regions are either 4-connected or 8-connected and not overlapping. A region being 4-connected means that two pixels that both fulfill the criterion will be in the same region if they are neighbours in a cross-like pattern. Pixel 'A' will only be considered to be the neighbour of pixel 'B' if they lie next to each other horizontally or vertically. 8-connectivity is the same as 4-connectivity with the addition of pixels being neighbours also if they lie next to each other diagonally. In Figure 2.3 you can see how an image can contain more segments if it's 4-connected than if it's 8-connected. The result of the segmentation will in most cases be a binary image with pixels having the value 1 (white) if they satisfy the condition, and 0 (black) if they do not.

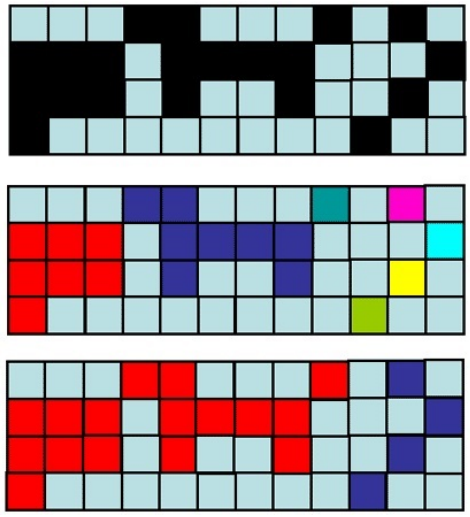

Figure 2.3: 4- and 8-connectivity. The top-most figure shows a binary image. The middle figure shows 4-connected segments in different colors. The bottom figure shows 8-connected segments in different colors.

For the task of hand gesture recognition, segmentation is very useful in detecting the hand. To do this however, we need to know what to look for in the image. If we know approximately what color the user's skin will have, we can simply set a color range that includes this color. This has been done successfully in many other projects [13, 24, 14]. However, if the system have to accommodate users with widely varying skin color, it might be necessary for the user to wear a glove with a known color and make the segmentation based on this color instead.

Since the color of the hand is not uniform and there might be similar colors in the background, the result of segmentation will often return not only the segment we are interested in, in this case the hand, but also other small segments here and there in the image. Also, there will often be ”holes” in the segment which may be caused by the user having a lot of hair on his hand or sub-optimal light conditions that cause a glare on part of the hand or on the glove used.

2.2.2 Erosion and Dilation

Erosion and Dilation are very common operations in image processing and computer vision. These operations are performed on binary images and are called morphological operations. This is because they alter the shape of the objects in the input image [19]. These operations (and many other image processing operations), are based on the mathematical concept of convolution, only in image processing we convolve two binary images, as opposed to mathematics, where the operation is performed on functions. In image processing we convolve a binary image with a much smaller matrix, called the structuring element, to produce a new binary image as the output. This structuring element can be anything from a simple 3x3 matrix to more complex structures like a disc or other odd shapes. Once the structuring element is chosen, it is positioned at every possible location in the image and convolved with the region it overlaps to produce the output.

Erosion
Much in the same way that water erode porous rock, carving away at the fragile mountain, erosion in image processing carves away parts of the objects in an image. An example of erosion can be seen in Figure 2.4 a and c. The idea is to remove anomalies and noise from the image, resulting in a cleaner object. This, however, will often render the object we are interested in being trimmed and smaller than we want. To compensate for this, dilation is often used on the image after erosion.

Dilation
Dilation is the opposite of erosion. With this operation we seek to make objects bigger instead of cropping them. The idea behind dilation is to fill in gaps or holes that we want to consider as part of the object, but that wasn't included, or removed during previous operations. An example of dilation can be seen in Figure 2.4 b.

As I mentioned earlier, segmentation is not perfect and might leave us with segments that contain holes or have anomalies. To rectify both of these problems a combination of erosion and dilation is useful. If we first perform erosion to remove anomalies on the hand (maybe some of the background was considered to be part of the hand), and then perform dilation to fill holes and gaps, this yields a good result [19]. Combining the two operations in this order (erosion first, then dilation), is called Opening. Even though opening a binary image will usually remove a lot of the unwanted segments from the image, there might still be some segments left that were large enough to pass through the opening, but that we still not want.

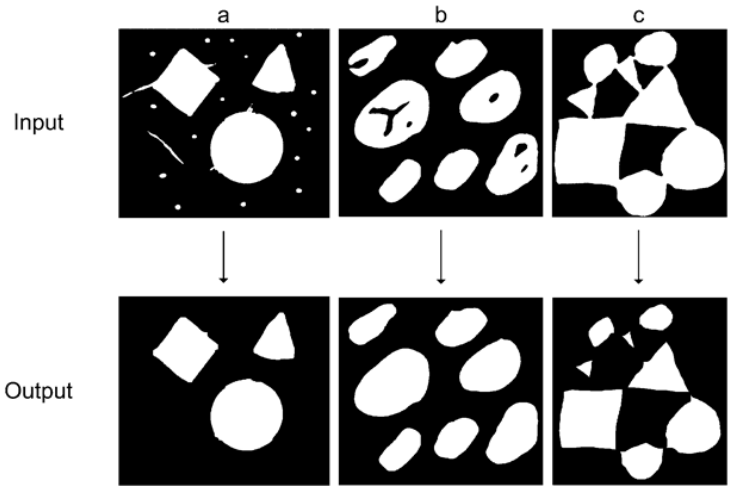

Figure 2.4: Morphological operations. (a)Erosion, (b)Dilation, (c)Erosion.

2.2.3 Find Contours

Finding the contours of the objects in an image is another very common task to perform in image processing. This is usually done on a binary image, and the result will be one list for every segment in the image. These lists will contain the coordinates of all the pixels that lie on the edge of the respective segment in an ordered fashion. Finding the contours in an image can be very useful for the task of hand recognition. In this case especially so, where the distance between the hand and the camera is never more than a meter. This means that after the segmentation it is safe to assume that the largest segment - and therefore also the largest contour - is that of the hand. By making this assumption we now know exactly which pixels of the image belongs to the hand.

There are several different ways to extract the contours of an image. As I already mentioned, I am using OpenCV for my image processing. This library contains a function for finding contours based on the Suzuki and Abe algorithm (or Suzuki85)[18]. The algorithm defines two types of borders; outer borders and hole borders. The difference between these can be seen in Figure 2.5. They define two components, namely 0-components and 1-components. These represent the background and holes (black pixels), and the segments (white pixels), respectively. The algorithm goes as follows:

1. Start scanning the image horizontally from the top leftmost pixel.

2. Interrupt the scan whenever you hit an outer border pixel or a hole border pixel, as shown in Figure 2.6. (If the pixel satisfies both conditions, then it must be regarded as the starting point for the outer border.)

3. Assign a uniquely identifiable number to the newly found border (sequential number of the border) denoted NBD.

4. Determine the parent border of the newly found border like so:
a) During the scan we remember the sequential number LNBD of the previously encountered border during the scan.

b) LNBD should be either the parent (surrounding) border of NBD or a border that shares the parent with NBD.

c) We can then decide the sequential number of the parent border of the newly found border with the types of the two borders by checking Table 2.1.

5. Follow the pixels on the border and mark them.
a) If the pixel lies between the 0-component that contains the pixel (i, j+1) and the 1-component that contains the pixel (i, j), change the value of pixel (i, j) to -NBD.

b) Otherwise set the value of pixel (i, j) to NBD, unless (i, j) lies on an already followed border.

6. When the entire border is marked, resume the scan and continue from point 2. When we reach the bottom right pixel in the image we are done.

	B' (LNBD) = Outer border	**B' (LNBD) = Hole border**
B (NBD) = Outer border	The parent border of the border B	The border B'
B (NBD) = Hole border	The border B'	The parent border of the border B

Table 2.1: Decision Rule for the Parent Border of the Newly Found Border B.

2.2.4 Find Convex Hull and Convexity Defects

When you have a contour, you may need to know exactly what shape that contour has. In order to evaluate this, it might be useful to know how many dents and protrusions there are in the contour. This is the task performed by finding the convexity defects of a contour. However, this can not be done until we have first found the convex hull of the contour, as the convexity defects of a contour are partly defined by the convex hull of that contour.

The convex hull of a contour is the set of points that form the smallest convex shape that encloses the contour. In the physical world this would be equivalent to having some object, say a star-shaped wooden toy, and then putting a rubber band around it. The rubber band would now be the convex hull of the star.

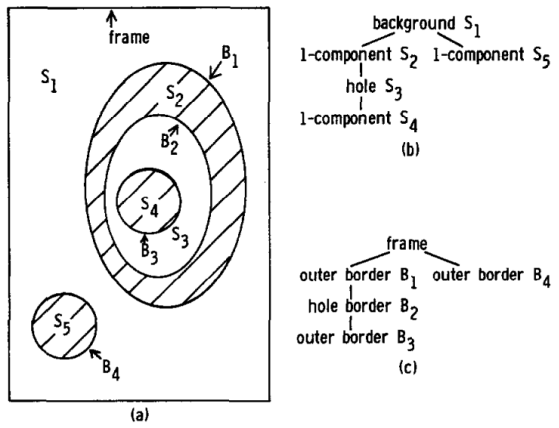

Figure 2.5: (a) Image with some segments(b) Surroundness among connected components and (c) among borders.

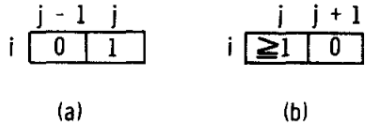

Figure 2.6: Conditions for pixel(i,j) being a (a) outer border pixel, or (b) hole border pixel

As with the task of finding the contours, OpenCV also has a method to find the convex hull of a contour. This method is based on Sklansky's algorithm[16] which runs with O(N logN) complexity in its current implementation. The algorithm goes as follows:

1. Choose the uppermost left vertex of the contour and call it O. Also assign it to the variable V_1.

2. In a counter-clockwise fashion, pick the two next vertices. Let the first be V_2, and the second V_3.

3. Let u_1, u_2, and u_3 be the position vectors of V_1, V_2, and V_3 respectively.

4. Calculate the values c and d according to $c = u_2 - u_1$ and $d = u_3 - u_1$.

5. Calculate the vector product $p = c \times d$.
a) If p is negative, then V_2 is concave, and is removed. This might cause V_1 to become concave, so we need to go back and check V_1 again. Let $V_2 = V_1$ and V_1 be the vertex that was V_1 in the previous iteration. Continue from point 3.

b) Else if p is positive, we let $V_1 = V_2$ and $V_2 = V_3$, and pick the next vertex in a counter-clockwise fashion for the value of V_3.
Continue from point 3.

6. When all the vertices have been checked and we return to O, we are done and the set of vertices that remain forms the convex hull of the contour.

Now that we have the convex hull of the contour, we can find its convexity defects. OpenCV has a method for this as well. This method will find the defects between a convex hull and a contour. Every returned defect is a tuple consisting of a start point for the defect, an end point, the point within the defect that is the farthest away from the convex hull, and the distance between this last point and the convex hull. If we assume that the contour we are analyzing is the contour of a hand, the convexity defects can be very useful in analyzing it further. It can for instance be used to find the fingertips on the hand, or it can be used as a feature for some machine learning algorithm. These two uses will be discussed further in Chapter 3. In Figure 2.7 you can see an example of how some of the convexity defects of the contour of a hand can help in analyzing it.

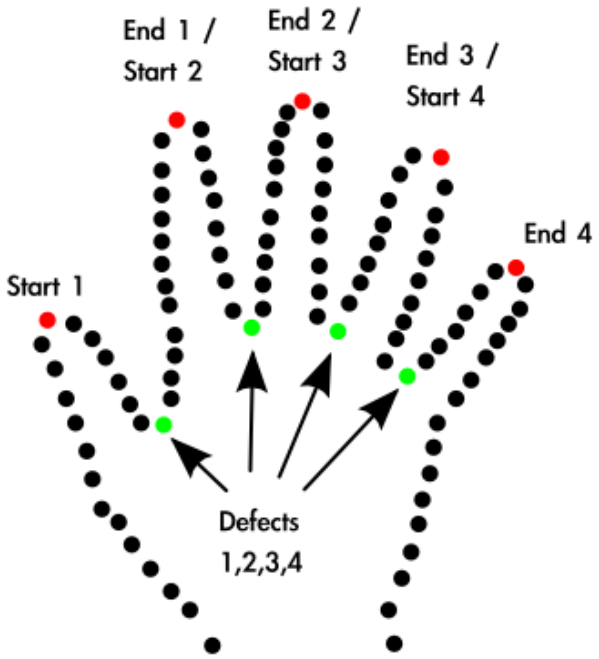

Figure 2.7: The convexity defects of the contour of a hand. The red dots are (some of) the vertices of the convex hull, and start-/end-points for the defects. (Also fingertips). The point farthest away from the hull is marked with green for every defect.

2.3 Artificial Intelligence and Machine Learning

The field of Artificial Intelligence (AI) is one of the newest fields in science and engineering, yet it is still very huge. AI is something that everyone living in a modern society will encounter every day, probably without realizing it. Companies like Google and Facebook uses AI for searching and configuring the presentation for the user using their applications. Supermarkets use AI to estimate where in the store a certain product should be placed to sell more of it. Hospitals use AI for diagnosing patients. AI is all around us. Tasks or goals of AI include, but are not limited to: Reasoning, Knowledge representation, Automated planning and scheduling, Machine learning, Natural language processing, Computer vision, Robotics, and General intelligence.

Many people will tell you that AI and Machine Learning are simply different terms for the same thing. According to Russell and Norvig [15] however, they are not. In fact Machine Learning is a sub-topic of AI. Machine learning deals with designing and developing algorithms to evolve behaviors based on empirical data. One key goal of machine learning is to be able to generalize from limited sets of data. Such algorithms will build a model from the data, and make decisions or predictions based on the model. This allows for more dynamic programs in the sense that they do not need to follow strictly static instructions, but can take one path or another, and even evolve by learning/discovering new paths.

In the above paragraph I mentioned that the algorithms, using Machine Learning, will build a model from available data. In the case of gesture recognition the available data will consist of a set of images or short video clips. Video clips will be most useful for dynamic gestures, while for static gestures images will suffice. These images and video sets will be sets where we already know which gesture is displayed. Having this prior knowledge will let us train the algorithm to have it learn how each gesture is represented, in order to predict which gesture is displayed in a new image/video clip.

Images in themselves do not really help us. They need to be processed and analyzed. This will be done by utilizing image processing techniques, such as the ones explained in the previous sections, to extract features that will build up a feature vector. A feature vector will replace an image with a lot of numbers. If we instead think of the vectors as points in an n-dimensional coordinate system (n being the number of features extracted from each image), we might be able to extract some pattern or logic from the data. Doing this is called Data Mining, which I will explain further in the following subsection.

2.3.1 Data mining

In our modern world we have computers and inexpensive disks that make it a lot easier to save data that would previously have been discarded[22]. Almost every move we make as individuals or as large industries are in some way tracked and stored in a database. It is practically impossible for a human being to draw something useful out of all this data, but for a computer it is not.

The task of data mining is to extract patterns and useful information from this ocean of data. Data mining is the practice of applying algorithms with the data available from a domain to solve related problems. To do this, the algorithms make good use of theory from statistics. These algorithms are used a lot by big companies. As I have already mentioned, internet companies like Google and Facebook will use their data to customize the user interface. If you have been browsing for fantasy books on Amazon, it is not unlikely that you will see an advertisement for fantasy books somewhere on Facebook. Data mining is the technology which enables this. All of this may seem very simple, but it is not. In order to successfully extract useful information from the data there are a lot of things that need to be done right. First off you need to consider which part of the data (which attributes) you want to evaluate. After that you need to make sure that the value ranges of the different attributes don't vary too much. Then you need to choose an appropriate algorithm to do the actual prediction.

As I mentioned in the previous paragraph, it is important that the value ranges of the attributes do not vary too much. I will go a little more into detail on this. First off, a lot of the data mining algorithms attempts to perform some sort of classification or clustering. In order to do this, they will often need to implement a distance metric. Typically the distance is measured with a Minkowski distance. Between two points X = (x_1, x_2,..., x_n) and Y = (y_1, y_2,..., y_n), the Minkowski distance is defined as $(\sum_{i=1}^{n} |x_i - y_i|^p)^{1/p}$. When p = 1 this is called the Manhattan distance, and when p = 2 it is called the Euclidean distance. The difference between these can be seen in Figure 2.8, where the green line is the Manhattan distance, and the red line is the Euclidean distance between two points in the two-dimensional space.

Figure 2.8: The Manhattan distance(green) and the Euclidean distance(red) between two points.

This is where the value ranges of the attributes come into play. Imagine if you have a case of round objects. Some are bowling balls, and some are coconuts. Now, for the sake of simplicity and to make my point more obvious, say that in order to classify these objects

we will consider only two attributes. These attributes are the weight of the object, and whether the object is edible or not. Let's say that the weight of the coconuts will vary from 500 g to 2000 g, and the weight of the bowling balls will vary from 1000 g to 7000 g. If an object is edible the value of the attribute will be 1, and if it is not edible it will be 0. Now consider a bowling ball weighing 1500 g and a coconut weighing 1500 g. In a coordinate system these points will practically lie on top of each other, and the distance between them would be minimal. If we next consider another bowling ball that weighs 4000 g, it will be extremely far away from the first bowling ball in comparison to how far the coconut is from the bowling ball. This is all due to the ranges we have for the different features. If we instead of measuring weight in g would measure it in kilograms, the range of the bowling balls would be between 1 and 7, while the coconuts will be between 0.5 and 2. If we also change the range of edible to be from 0 (not edible) to 10 (edible), our classifier would have much better performance. Now you may argue that for the sake of this classification it would be redundant to consider weight, since the edible feature is enough to classify the objects. For this simple example that is true, but if we put in some hazelnuts and some tennis balls into the case, both of our features would be necessary.

The simple example in the previous paragraph illustrates both how important value ranges are, as well as how important it is to pick the correct features. In the original example it would suffice to say whether or not an object was edible, while in the extension we would also need to consider the weight. For both of them it would be useless to consider the shape of the objects, since they are all round. (A feature called roundness that consider how perfectly round the object is, however, could prove useful). It is worth noting that for a feature that is not naturally a numerical value, such as edible/not edible in this example, or color, or a medical symptom, we replace the natural value range of the feature with numerical values so that we can calculate a distance. As we saw in the example, these numerical values should be chosen carefully and not assigned arbitrarily.

Now that we have a basic understanding of the data representation in data mining we can move on to discuss the algorithms used. According to [23] the 10 best-known algorithms in data mining are: C4.5, K-means, Support vector machines (SVM), Apriori, ExpectationMaximization (EM), PageRank, AdaBoost, K-nearest neighbor (KNN), Naive Bayes, Classification and Regression Trees (CART). C4.5, SVM, KNN, Naive Bayes, and CART fall into the category of classifiers. If you have a defined set of classes, like hand gestures for example, a new data object that passes through one of these algorithms will end up being classified as one of the predefined classes. Usually for this to work there is some training involved, where we have a data set of which we already know the class of every instance. We will then tell the system which features of the data to look at, and build the classifier from this. In the following subsection I will go more into detail on how the KNN classifier works. K-means and EM are clustering algorithms. The objective of these algorithms is to divide the data into clusters. That means that they need to find some pattern that separates the data into distinct groups. This can be useful for detecting anomalies, like in the case of a stolen credit card. If there is all of a sudden a huge over-seas transaction on an account, clustering could help the bank detect this anomaly, and alert the customer and potentially take action against it. The apriori algorithm attempts to find frequent item sets

in the data and derive association rules. This can be useful for the owner of a supermarket. If the owner of a supermarket is able to derive that beer and snacks are often bought in the same transactions, he can decide to put the beer and the snacks next to each other in the store in order to speed up the shopping of the customers, and also attempt to influence more customers to buy both of the items. AdaBoost is an ensamble method. That means that it will be combined with one or more other learners in an attempt to be more accurate. Other researchers, such as Viola and Jones[21] have found this algorithm very useful. PageRank is a search ranking algorithm, and is the foundation of the search engine Google. The algorithm will assign a rank to every hyperlink, and produces a static ranking of web pages.

As you may understand by now, machine learning and data mining are both huge areas of research, and they are both very useful. In the following subsection I will explain, in more detail, how the KNN algorithm works. This algorithm I used for hand gesture recognition, and the details on how it was used will be given in Chapter 3.

2.3.2 K-Nearest Neighbours

The K-Nearest Neighbours (KNN) algorithm is designed to perform classification. It is an improvement and an extension of a look-up table. It is an algorithm where both the distance metric (”nearest”) and the number of neighbours can be decided by the programmer. It is one of the simplest machine learning algorithms, but still very effective in some cases. A shortcoming of the algorithm is that it is sensitive to the structure of the data. This is where it becomes important to pick the good features, the good distance metric, and make the value range of the different features approximately similar, so that the distance in one dimension does not overrule all the others. It is also important to note that the algorithm will work best in low dimensional spaces (data with few features), and will become exponentially worse as the number of dimensions increase. This is referred to as ”the curse of dimensionality” in the literature[15].

The result of the training part of this algorithm is basically a matrix and a vector. The matrix will contain the data. Every row represents an instance, and every column represents a feature. The vector will have one element for every instance of training data (one for every row in the matrix). The element at position i in the vector will be the class of the instance in row i of the matrix. Alternatively the vector can be added on to the matrix as an extra feature (column).

Now that the data representation is taken care of, it is time to choose a distance metric, and a value for k. The most common distance metric is Euclidean distance. This is how most people think of distance between two points. It is simply a straight line from point A to point B as you saw earlier in red in Figure 2.8. Another common, but quite more complex distance metric is Mahalanobis distance. With this metric the distance of an observation x = (x_1, x_2,..., x_n) from a set of observations with mean μ = (μ_1, μ_2,..., μ_n) and covariance matrix S is defined as $D_M(x) = \sqrt{(x-\mu)^T S^{-1}(x-\mu)}$. This results in an elliptical distance, as opposed to the circular of the Euclidean distance metric. The difference between these two can be seen in Figure 2.9. Here all the points, including

the two green dots, that lie on the turquoise line has the same distance to the black dot in the center according to the Mahalanobis distance metric. The two green dots have very different distances from the black dot according to the Euclidean distance metric, as shown by the two red circles and the length of the black arrows. If you try using Euclidean distance as your metric, but do not get any good results, it might be that the data is structured in an odd fashion in the feature space. Using the Mahalanobis distance instead could help in solving this problem.

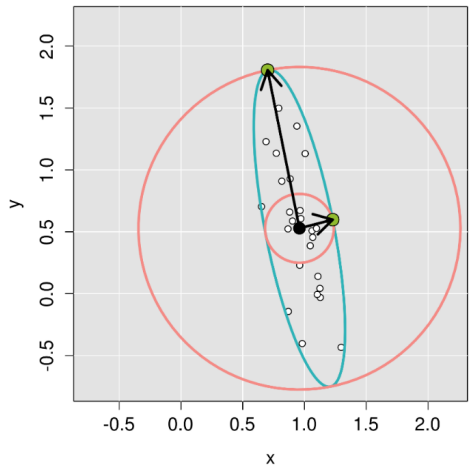

Figure 2.9: Difference between Mahalanobis distance(turquoise) and Euclidean distance(red).

Choosing a value for k can have a big impact on the results of the algorithm. It is this value that says how many neighbours we will look at when we do the classification. The easiest way to explain what the role of this value is in the algorithm is to explain the algorithm itself. I will start explaining from when the training is complete and the distance metric is chosen. The algorithm will then take a new instance as input and attempt to classify it.

1. Extract from the new instance all the same features that were extracted from the training data, and store them in a vector.

2. For every row/instance in the training data apply the distance metric to calculate the distance between it and the new instance.

3. Store a tuple consisting of distance and the class of the training data instance for all the instances in the training data.

4. When the distance between the new instance and all the instances in the training data has been calculated, sort the list containing the distance - class tuples by ascending distance (having the element with the shortest distance as the first element).

5. From this list extract the k first elements. These will represent the k instances from the training data that are closest to the new instance according to the chosen distance metric. (The k nearest neighbours).

6. The new instance will have the class of the most frequent class among the k extracted elements.

Choosing a good value for k can be difficult. It is usually set to be a small value between 5 and 11, but this will largely depend on the size of the training data, as well as the number of classes. One way to make a decision on the value for k is to do validation. This is done by having a test set of instances. It is important that this test set does not include any of the instances used for training, or it will obviously perform extremely well, and we would not really get anything useful from the validation. Therefore we use new instances that in total represent all the different classes, and we already know which class each instance should have. Now we set a value for k and run the algorithm on the test set. The performance is measured as how many instances the algorithm can correctly classify. It is common to run the algorithm several times with different values for k to find which one is best suited for the specific problem. When the best one is found, the validation process is complete, and we can start classifying new data.

Chapter 3

Hand Gesture Recognition

In this chapter, I will start off by explaining exactly what hand gesture recognition is, and what it can be used for. It obviously has something to do with recognizing the hand of a person, but in the first section of this chapter I will dive a little deeper. Next, I will discuss some previous work on solving this problem. The two driving questions for this discussion is: ”What have other researchers studied?”, and ”What results have this yielded?”. Finally, I will explain my approach to reach the goal of this thesis, as stated in Chapter 1.

3.1 What is Hand Gesture Recognition

Hand gesture recognition is a topic in HCI. The goal is for a computer to interpret human gestures via mathematical algorithms. By doing this the hope is that interaction between humans and computers will be more natural and intuitive. At the time of writing, the most common way for humans to interact with computers is to use a combination of a mouse and a keyboard, but with touch-screens becoming ever more popular. While the use of touch-screens is already an improvement from the use of mouse and keyboard with regard to intuitiveness, hand gesture recognition takes it one step further by eliminating the need to directly interact with a physical object. One of the reasons for why hand gesture recognition is only recently starting to see the light of day, is because it is computationally expensive compared to interpreting physical input devices, such as mouse and keyboard and touch-screens. However, with modern computers and their processing power it is now possible to interpret images and video and extract gestures from them in real-time. (As opposed to older computers where you had to wait several seconds before the input was registered and interpreted by the computer).

There are several interesting applications of hand gesture recognition. One example is computer games. In strategy games, where the player need to move troops around on a map, there is a concept called micro-management. The player has to keep track of where all his troops are at any given time, and be able to issue commands to troops on different parts of the map in quick succession. If instead of moving the mouse around he/she could

move his/her fingers in the air and make gestures to issue the commands, I foresee that the micro-management would become a lot easier. Another example, which is rooted in everyday life, is to help the physically impaired to interact with computers, such as interpreting sign language. Even for people who are not physically impaired it would be beneficial to interact with systems by using hand gestures. Some systems are controlled by voice, which can be troublesome in areas with a lot of noise. Other than that, hand gesture recognition could be used for robotics [7] and for virtual reality [17].

While we still have some way to go before we make input devices such as mice, keyboards and even touch-screens obsolete, there has been a lot of interesting research on the topic of hand gesture recognition over the past years. Some of these are highlighted in the next section.

3.2 Previous Work

Through my research I have found that most researchers describe hand gesture recognition as a three-step pipeline from when the image is acquired. These three steps are: extraction method, feature estimation and extraction, and classification or recognition. The first step deals with pre-processing the image. Here we need to perform segmentation on the image to extract the important part of the image. This segmentation process will vary depending on whether the system should consider only static gestures, or if it should also be used for dynamic gestures, in which case the hand will need to be tracked as well as segmented. There are two main ways to do segmentation of the hand. Some researchers make use of a colored glove to make the segmentation more robust and invariant to illumination and background, while others make the segmentation based on skin color. Ibraheem et.al. conducted a study [9] where they compared skin color based segmentation techniques. They found that different techniques performs better in some environments than others, so that the chosen technique should depend on the environment of the research. Overall they concluded that segmentation based on skin color is easy and efficient, but with the drawback of being sensitive to illumination changes and interference with backgrounds with similar colors as that of the skin. Other studies such as [24, 6, 26] have also shown that segmentation based on skin color can be efficient. In [11] Lamberti and Camastra made a system where they use a glove with a few colors to make the segmentation more robust and invariant to illumination and background, which also proved to work well with a recognition rate close to 98%.

After the extraction of the hand comes feature estimation and feature extraction. It can be a difficult task to figure out which features of the image are important in order to recognize the gesture depicted, as there are so many possibilities. Seeing as hand postures and gestures will vary greatly from person to person, due to the fact that not all hands are similar, it is essential to capture the invariant properties of the hand [8]. In this article Hasan and Abdul-Kareem describe two methods for gesture recognition. In the first method they use the contour of the hand as the feature with the intention of eliminating the problem with size difference and translation. In addition to the hand contour they use ”general features”. These describe the height and width of the hand, and compounded

with the hand contour, this results in a feature vector with 1060 elements. In the second method, they use complex moments of the segmented gesture as features. This also deals with the problem of rotation. For both methods they have the program differentiate between 6 static gestures. With the first method this results in a recognition rate between 50 and 88% for the different gestures, and a total recognition rate of 70.83%. With the second method they produce a recognition rate between 75 and 100% for the different gestures, and a total recognition rate of 86.38%. In [10] Ke et.al. propose a system that uses 2D Gabor transformation, primarily used to analyze the texture of the hand region, for feature extraction. These features are then used to distinguish between 5 different gestures, and results in an accuracy of 98.76%. Despite what is stated by Hasan and Abdul-Kareem in [8] about having to extract invariant properties of the hand, there are researchers that has had success with extracting shape-based features. One of these is [13] where Panwar uses orientation, center of mass, status of fingers (folded or not), status of thumb, and the location of the fingers in the image as his features, and end up with an approximate recognition rate of 94% on a set of 45 different gestures with 10 samples of each. Another example is [20] where Trigueiros et.al. propose 2 different sets of features to be used in classification. One of them uses the mean intensity value of the pixels in the region where the hand is, the variance in intensity in the same region, the area covered by the hand, the perimeter of the hand, the angle of the hand, and the number of convexity defects. The second set also uses the mean, variance and angle from the first set, and adds on 36 values from the orientation histogram of the hand as well as 100 values from the hand's radial signature. Both methods yielded good results, but the first one was a bit better.

The final step in the pipeline is classification or recognition. As with feature extraction there is a huge number of different approaches that could be used to classify the gesture. In [10] Ke et.al. used a Support Vector Machine (SVM) to make the classification based on the features extracted from the training data. Hasan and Abdul-Kareem [8] make use of a Neural Network (NN) for their classification, while in [25] Ying Yin and Davis have successfully used a Hidden Markov Model (HMM). In [20] Trigueiros et.al. attempt to find which machine learning technique is best suited for hand gesture recognition. They use both KNN, Naive Bayes (NB), NN, and SVM to do the classification. All these techniques are run twice on two different sets of features, as described in the previous paragraph. A summary of how the different techniques performed on the two feature sets in terms of precision can be seen in Table 3.1.

	KNN	**Naive Bayes**	**NN**	**SVM**
Feature set 1	96.89%	22.47%	96.80%	91.59%
Feature set 2	87.67%	85.12%	85.12%	81.44%

Table 3.1: Average precision of the different classifiers used on the two feature sets described in [20].

From this table it is easy to see that Feature Set 1 gave better results than Feature Set 2, except when using NB as the classification technique. This fact suggests that there is not necessarily one absolute best way to do hand gesture recognition. Also when looking at the

other research I have referenced in this section there are clearly other ways to go about the classification, both in terms of feature extraction and choice of classification technique, in order to get good results. Considering all the possible combinations of features extracted and combine this with the possible classification techniques it is fairly obvious that the number of possibilities is huge, and there are likely great combinations that have never been researched before.

While most researchers consider the hand gesture recognition task as the three-step pipeline described above, there are some that take a different approach. In [5] Chowdary et.al. explore 4 different algorithms for hand gesture recognition that does not use any sort of machine learning. Therefore there is no database or training data to base the decision on. In all the approaches they define a gesture simply by how many fingers are held up in the image. This means that the system will interpret a hand showing the index finger with the other fingers folded the same way as it would interpret a hand showing only the little finger. The three first algorithms they suggest are pretty simple, and not very robust. All 4 algorithms start off by taking an RGB image as input and converts it to a binary image (presumably based on some threshold). The first approach then counts the number of white pixels in the image, and based on some predefined ranges they make a decision on which gesture is shown. This method is insensitive to rotation, but quite sensitive to scaling. If the user's hand is too far or too close to the camera, it will be interpreted as a different gesture than intended. In the second approach they have the user draw black circles or put black markers on every finger. Having the binary image, they then proceed to count how many circles are in the image. This number should be equivalent to how many fingers are shown. In addition to being insensitive to rotation, this algorithm is also insensitive to scaling, however, they experienced problems with detecting the thumb. In the third algorithm they first perform morphological operations, which I described in Section 2.2.2, on the image, and then a hit or miss transform, which results in an image showing only the edges along one side (the right side in this case), as shown in Figure 3.1. After that, they again apply the morphological operation dilation, to make sure that there are no holes in the lines. After that, they count each object and this count says how many fingers are shown.

Figure 3.1: Image used in the article [5], showing the results of the hit or miss transformation.

In the fourth approach the authors have first smoothed the image with a median filter, and then divided the image in two halves horizontally. After that they perform 3 - 5 horizontal scans in each half to detect the fingers. To make the approach insensitive to rotation, the image is also divided into two halves vertically and then these two halves are also scanned (vertically) to detect the fingers. To conclude the authors say that the last approach is the most robust, being able to correctly predict the gesture in 82.47% of the images.

3.3 My Approach

As I have already mentioned, the research goal of this thesis is to ”Compare methods in Artificial Intelligence to see which is best suited for hand gesture recognition”. To do this I have taken two different approaches in writing two programs to perform hand gesture recognition. Both of them tries to recognize 6 different static gestures. Examples of these 6 gestures can be seen in Figure 3.2. The first program I wrote uses only image processing techniques, similarly to what Chowdary et.al. did in [5], only I go through a few more steps. This is to make the solution more robust and hopefully be able to recognize most of the gestures correctly. In the second program I use a database of 600 images and extract features from all of them to build up feature vectors to be used as a data set. I then use this data set to make predictions on new images by using the KNN classification algorithm. The reason why I chose these two approaches is because I wanted to see how a program that performs gesture recognition only by using image processing techniques compares to one of the best[20] machine learning techniques. For both the first approach and the feature extraction of the second approach I used the library OpenCV[2] for my image processing. My two approaches will be discussed in greater detail in the following two subsections, and in Chapter 4 I present my results of the experiments to test and compare the two approaches.

Figure 3.2: Examples of the 6 different gestures to be recognized.

3.3.1 System 1: IPRec - Gesture Recognition with Image Processing

When I first started working on this thesis I thought that I should write at least one program that would perform gesture recognition. My first idea was to perform the hand gesture recognition using only image processing techniques. Before I started conducting a literature review on this I had some initial thoughts as to how I should go about doing it. I obviously needed to segment the image to extract the pixels that represent the hand, and I would probably have to refine the result of the segmentation in some way. Next I would have to get into some more advanced methods to recognize which gesture was shown in the image. I was happy to find that other researchers have had success with gesture recognition by only using image processing techniques [5]. Although they did not have quite as high recognition rates as I had expected, it gave me confidence to pursue the matter further. Researchers using machine learning techniques for prediction, such as [24, 26, 8, 20], as well as the OpenCV forum [3], also helped to let me know what I should have my program look for in the image in order to have it recognize the gesture. Throughout the development of my first approach I tested the resulting application at each step to make sure that the solution was a viable alternative in practice and not just theoretically good. Using this strategy also told me if the program could recognize gestures in real-time, or if the processing power required was too much.

The first step I had to take in order to make a real-time gesture recognition program was to make sure that I was able to capture images and continually feed them to the program. To do this I used the PlayStation Eye discussed in Section 2.1. However easy this may sound, it was not. I struggled a lot in the beginning to make the drivers work and have my program detect and use the camera. Initially I got an old driver for the camera from supervisor, that the other students I mentioned in Section 2.1 had used for their project. When this was made it was free to download and available to whoever wanted to use it. However, the driver is now outdated, and it was impossible to find documentation on how to use it. I tried with C++, C#, Python and Java, but I could not get it working in either language. After some time I had to give up and purchase the newest version of the software for the camera. Being most comfortable with Java, I chose this as my developing platform. With the newest software and some help from the CodeLaboratories [1] forum I was able to get it working.

Once I had an image to work with, it was time to start processing it. As almost all researchers I discussed earlier seems to agree, the first step is to segment the image. I started out by having the user wear a glove with a known color. This was chosen because my initial research suggested that segmenting the hand based on skin color was next to impossible. However, as I kept on reading research papers I soon discovered that it is very much possible to do segmentation of the hand based on skin color. I then took a few pictures of my hand with the PlayStation Eye and analyzed the color ranges of the hand in order to find the color range to use for the segmentation. The easiest way to do this with OpenCV is the function *inRange(Mat src, Scalar lowerb, Scalar upperb, Mat dst)*, where src is the source image and dst is the resulting binary image. lowerb and upperb are the lower and upper bounds, respectively, represented as tuples of the form [B, G, R], where the letters stands for blue, green, and red, respectively and are integers ranging from 0 to 255. Using

this function will result in a binary image where every pixel from src that lies within the range [lowerb, upperb] will be white, while those that fall outside the range will be black. After tuning the bounds a little, I ended up with a good segmentation of the hand, with only a few, small other segments. To refine the segmentation I then eroded and dilated the image.

At this point it was time to figure out how the system should separate the different gestures from one another. As discussed in Section 2.2.4, convexity defects seems like a good path to investigate. This was also suggested in various posts on the OpenCV forum. As already mentioned in Section 2.2.4, OpenCV has a function to find convexity defects, called ”convexityDefects”. In order to use this function, however, I first had to find the contour of the hand, as well as the convex hull of the hand. Using OpenCV’s function ”findContours” (discussed in Section 2.2.3), I got a list with all the contours in the binary image acquired in the previous paragraph. Assuming the hand is the biggest contour in the image, I could then extract this contour from the list with the help of the function ”contourArea”, which returns the area encircled by a contour. If the biggest contour in the image was not big enough (according to the constraints I had set based on camera resolution and the distance between the camera and the user’s hand), to be considered a hand, the image was dropped and the recognition process cancelled. After a sufficiently large contour had been extracted, I used the function ”convexHull” to get the convex hull of the biggest contour, which based on the same assumption as above, would be the hand. Having the contour and the convex hull of the hand I could now extract the convexity defects. However, OpenCV’s function for doing this will return all the convexity defects between the convex hull and the contour. For my application I am only interested in the defects assosiated with the fingers on the hand, which means that I somehow had to find these defects. In Section 2.2.4 I mentioned that a convexity defect is defined by it’s start point, end point, point on the contour farthest away from the convex hull, and the distance from this last point and the convex hull. This distance can be used to filter out the defects that doesn’t lie between fingers. The ones that do lie between the fingers will have a distance that is quite a lot larger than the others. I experimented with a few different values for this, and eventually landed on a value that gave good results. This value will depend on the resolution of the image, as well as how far away from the camera the hand is. The process described in this paragraph is visualized in Figure 3.3.

Figure 3.3: The process of finding the correct convexity defects after acquiring the binary image of the hand.

This leaves me with only the defects that lies between the fingers. Now it is time to find out how many fingers, and thus which gesture, is shown in the image. I can use the number of defects, as well as their start- and end-points to count the fingers. (When I refer to different gestures in the following explanation, I will be referring to those shown in Figure 3.4, where the convexity defects can also be seen). If there are no defects left, that means that no fingers are held up, as in Gesture0. If there is only one defect, as in Gesture1, that means there is only one finger being held up. If there are two defects, as in Gesture2 , there are two fingers in the image. However if there are three defects, it is either Gesture3 or Gesture4 . Notice that in Gesture3 there is at least one defect where the distance between the start-point and the end-point is much greater than any of the defects in Gesture4. This is what I ended up using to separate between the two gestures. If the distance is large enough it is Gesture3, and 3 fingers are being held up, otherwise it is Gesture4, and 4 fingers are being held up. Finally, if there are 4 defects, 5 fingers are being held up, like in Gesture5.

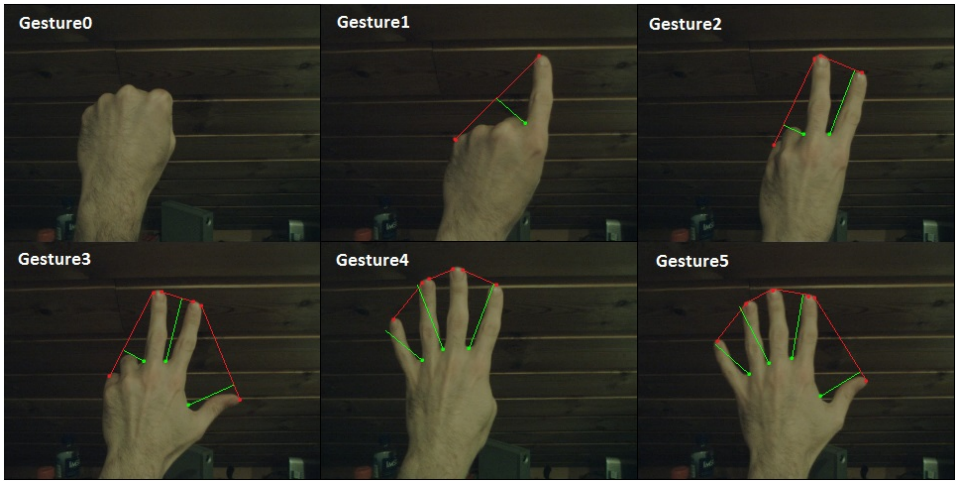

Figure 3.4: Examples of the 6 different gestures to be recognized with convexity defects drawn on. The red points represent the start- and end-points of the defects, while the green represent the point farthest away from the convex hull. The green line is the distance between the convexity defect and the convex hull and is orthogonal to the red line.

In Figure 3.5 you can see the results of each step described in this approach. The top left image is the original image captured by the PlayStation Eye. The second image on the first row is the original image after being segmented based on skin color. The first image on the second row is the segmented image after morphological operations have been applied. The second image on the second row is the original image with the largest found contour drawn on. The first image on the third row is the original image with the largest contour, the convex hull and all the convexity defects drawn on. The second image on the third row is the fingertips drawn on to the original image.

Figure 3.5: The results of after each main step in IPRec.

3.3.2 System 2: KNNRec - Gesture Recognition with K-Nearest Neighbours

For my second approach, I wrote a program that follows the KNN algorithm to perform the gesture recognition. The reason why I chose this algorithm over any of the other common machine learning algorithms is that it proved to be the best among the algorithms tested in [20]. Other than that article it did not seem like other researchers have used the algorithm for gesture recognition. With my program I attempt to recreate the results achieved in [20]. As the paper does not give any information on which gestures their method can recognize, I will be using the ones depicted in Figure 3.2. This approach actually consists of 3 different programs. One to capture images, one to train the algorithm/construct the data set, and one to perform the recognition based on the data set.

The first thing that needs to be in place for any machine learning technique to work is the data set that the algorithm will use to make the classification. To create this data set I wrote a small program that uses the PlayStation Eye to capture images and save them to a specified folder. I then proceeded to capture 100 sample images for every gesture to be recognized which I would use for my training data. I also captured a total of 67 images representing all the gestures to be used for the experiments in Chapter 4 (which later will be referred to as the Test Set). During this capturing process I moved my hand around a bit, so that not every image would be the same, but I made sure to always keep my hand within the frame and not to move it too much in the depth direction. Next up I had to extract features from all of these images to build the actual data set. In [20] they had most success with their first data set consisting of the features: the mean intensity value of the pixels, the variance in intensity, the area covered by the hand, the perimeter of the hand, the angle of the hand, and the number of convexity defects. In the following paragraphs I explain how each of these features were extracted from the image.

Mean intensity and Variance in intensity:
To get the mean intensity and the variance in intensity, I first had to convert the image from RGB to gray scale. This was done using OpenCV's function *cvtColor(Mat src, Mat dst, int code)*, where src is the source image, dst is the destination, or output image, and the code says what you want to convert from, and what you want to convert into. Here I used the code *Imgproc.COLOR_BGR2GRAY* which converts from RGB to gray scale. Next I made an array of size 256 (as the pixels in the gray scale image can have any integer value (intensity) ranging from 0 to 255), and iterated through the pixels of the gray scale image. For every pixel in the image I extracted its intensity, n, and added 1 to position n in the array I made. After the iteration was complete I iterated through the array and summed up the total intensity in the image, and divided it by the number of pixels in the image to get the mean intensity. Next I calculated the variance in intensity. The formula to get the variance is $Var(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$, where x_i is the i'th pixel, n is the total number of pixels, and μ is the mean intensity in the image. This works because the background is static in my case, and I do not change the distance between my hand and the camera. If you have a background that keeps changing, or if you change the distance between the hand and the camera, you will first have to find a bounding box around the hand and only consider the pixels within this box to make the mean intensity and variance in intensity

invariant to changes in the background and the scale of the hand. This is what the authors of [20] did.

Area and Perimeter of the hand:
In order to get the area and perimeter of the hand, I only needed to get the contour of the hand. To do this, I again had to segment the image, but unlike what I did in Section 3.3.1, I now wanted to see if I could also segment the image based on intensity values. I therefore analyzed a few of my gray scale images and found appropriate lower- and upper bounds to use with the function *inRange(Mat src, Scalar lowerb, Scalar upperb, Mat dst)*. This segmentation also worked very well. Next, I performed erosion and dilation just like I did in Section 3.3.1, and then extracted the contours of the image. After that, I iterated through the list of contours to find the one covering the greatest area (the hand), using *contourArea(contour)*. I also saved the index of this contour so that I would know which contour to extract the perimeter from. I then used the function *arcLength(curve, closed)*, where curve is the contour of the hand and closed is a flag that says whether the curve is closed or not. In my case it is closed. This function will return the length of the contour, i.e the perimeter.

Angle of the hand:
The orientation, or angle of the hand can be calculated by using image moments. They involve sums over all pixels and are defined as $M_{ij} = \Sigma_x \Sigma_y x^i y^j I(x, y)$. Using moments, the center of the hand can be calculated as $x_c = \frac{M_{10}}{M_{00}}$ for the x-coordinate, and $y_c = \frac{M_{01}}{M_{00}}$ for the y-coordinate. By using the intermediate variables

$$a = \frac{M_{20}}{M_{00}} - x_c^2 \qquad b = 2(\frac{M_{10}}{M_{00}} - x_c y_c) \qquad c = \frac{M_{02}}{M_{00}} - y_c^2$$

then angle of the hand can then be calculated as $\theta = \frac{\arctan(b,(a-c))}{2}$. To do this i used OpenCV's function *moments(Mat array, boolean binaryImage)*, where array is the image to be used, and binaryImage is a flag that says whether array is a binary image or a gray scale image. For this to be accurate with a binary image, it is very important that there is only one segment in the image, or else the center described above will be calculated as the center between all the segments. I therefore created a new binary image with only black pixels, drew the largest contour (the hand) on it with white pixels, and filled it using the function *drawContours(image, contours, contourIdx, color, thickness)*, where image is the image to be drawn on, contours is the list with all the contours I extracted earlier, contourIdx is the index of the largest contour, and thickness says how thick the contour should be drawn. If this last value is set to be negative the function will draw the interior of the contour. Now that I had a binary image with only the segment of the hand, I could send it into the function that extract the moments and then calculate the angle of the hand as described above.

Constructing the feature vector and the data set:
The last feature to be extracted is the number of convexity defects. I did this in exactly the same way as I did in 3.3.1, and also filtered out defects that was not between the fingers in the same way. At this point I have extracted all the features that they used in [20] with

their most successful attempt. Next I made an array of size 7 designated for the 6 extracted features. In the last entry of the array I put the class of the image. Examples of feature vectors are included in Appendix A. After this, I made a for-loop that iterated through all my 600 images to extract the same features and construct a similar feature vector. I stored all the feature vectors as entries in an ArrayList. Throughout the entire feature extraction process I kept track of the largest value for all the features. When I had iterated through all the images I then normalized all the feature values by dividing all of them by the largest value for that feature. This made it so that all the features have a value ranging from 0 to 1. If I had not normalized the values, there would have been a huge difference in value ranges. Number of convexity defects would range from 0 to 4, while the area covered by the hand could be as large as 50,000 .(Remember the discussion in Section 2.3.1). Finally, I made 2 .txt-files. One to hold the data set, and one to hold the largest values for all the features. This concludes the training of the KNN algorithm.

To perform the recognition I wrote another program that looks very much like the one described above. It loads the data set from the .txt-file, extract the values and put them into feature vectors that in turn are stored in an ArrayList. The largest values found during training are also loaded into an array. Next, a query image is loaded in, the features are extracted in the same way as above, and then normalized using the largest values from the training. Then the classification is performed as described in Section 2.3.2 and given as output to the user as text in the console. This process is further explained in Chapter 4 where I will discuss the experiments that I performed.

Chapter 4

Experiments and Results

In this chapter, I will perform experiments on the two approaches described in Sections 3.3.1 and 3.3.2, evaluate the results, and compare the two approaches based on the experiments. I will start by giving some information on the components used in the experiments, such as what hardware was used. Next, I will say something about the data used to perform the experiments. Then, I present a review of the actual experiments where I show the results. These experiments are conducted in order to answer research question 3 and research question 4 from Chapter 1.

4.1 Hardware and Software

For the implementation of my two approaches and the experiments conducted I have used a HP laptop with a 64-bit Windows 7 operating system. It has an Intel Core i7 processor with 4 cores running at 2.20GHz and 12GB RAM. The capturing device I have used is the PlayStation Eye described in 2.1. The use of the PlayStation Eye is the reason why I have done the implementation in Windows 7 rather than in Windows 8, as the camera's driver was not supported in Windows 8. (I changed from Windows 8 to Windows 7 because the old driver did not work in Windows 8. I have not checked to see if the latest driver is supported in Windows 8).

The software used for the experiments is the programming environment Eclipse and the Java programs I described in Sections 3.3.1 and 3.3.2. These implementations will be the foundation for all the experiments, but for some of them I have made a few changes. These changes will be explained in Section 4.3, where a more detailed description of all the experiments will be given.

4.2 Experimental Data Set

Throughout this chapter I will be referring back to the Test Set that I defined in Section 3.3.2. The Test Set contains images from all 6 gestures to be recognized by my two approaches. The set consists of 11 samples of Gesture0, 16 samples of Gesture1, 11 samples of Gesture2, 8 samples of Gesture3, 12 samples of Gesture4, and 9 samples of Gesture5. These images are saved to disk as JPEG-files and read into the programs one by one during the classification/prediction.

4.3 The experiments

In this section, I describe the different experiments. I state what the purpose of each experiment is, and if I made any changes to the code, those changes will be reviewed as well.

4.3.1 Experiment 1: Will IPRec be able to recognize gestures in real-time?

With my inspiration for this thesis being SoundDream, described in Section 1.1, I had to make sure that the systems I created was able to perform gesture recognition in real-time. This experiment is designed to make sure that the image processing done in IPRec is not too heavy computationally for the computer to be able to process the images and make predictions in real-time. As already mentioned in Section 3.3.1, I initially wrote the program to recognize gestures in real-time, so in order to test it I simply had to make the program display the images captured and see if the recognition was able to keep up with the images shown on screen.

4.3.2 Experiment 2: Will KNNRec be able to recognize gestures in real-time?

Same as with Experiment 1, I had to make sure that KNNRec could also perform gesture recognition in real-time. To test this I used the Test Set and the program described in the last paragraph of Section 3.3.2. I had my KNN algorithm go through and classify all the 67 images in the Test Set and timed how long it took from when the first image was loaded until the last one had been classified. I then divided it by the number of images in the Test Set to get the average time spent processing and classifying 1 image. From this I can say how many images I am able to classify each second.

4.3.3 Experiment 3: Classifying the Test Set

This experiment is conducted in order to answer research question 4 from Chapter 1. To do this, I had both my systems perform gesture recognition on the Test Set. In the following section I will present the results from this experiment, and in Chapter 5 I will discuss what these results mean in terms of the research question. KNNRec was primarily designed to perform gesture recognition on a test set, so this system is already good to go. With IPRec

however, I had to make a few adjustments in order to have it recognize gestures from images saved to the disk rather than images captured from a camera directly. I achieved this by using the code from KNNRec that iterates through all the images, and put my image processing and prediction from IPRec inside that loop.

4.4 Results

In this section I present the results of the experiments described in the previous section. I will not be discussing what these results mean, as I will be doing so in Chapter 5, but I will explain what the results show.

4.4.1 Experiment 1: Will IPRec be able to recognize gestures in real-time?

When running IPRec as explained in the previous section it worked exactly as it should. There was no delay between when I changed from one gesture to another and when that gesture was recognized and displayed in the console. I had the program running for 10 minutes to see if there was a small delay for each image that piled up to become a big delay after some time had passed, but there was no recognizable delay. As an addition I timed how much time went by when classifying the Test Set using IPRec from when the first image was loaded until the last image was processed. In total 950 milliseconds went by, which means an average of 14.2 milliseconds per image, which in turn means that the system is able to process 70 images each second.

4.4.2 Experiment 2: Will KNNRec be able to recognize gestures in real-time?

I executed the experiment 3 times getting quite similar processing times with the slowest being 8878 milliseconds, and the fastest being 8460 milliseconds. On average this gives 8629 milliseconds which means that on average 128.8 milliseconds was spent processing 1 image. This in turn means that the system is able to process 7 images each second.

4.4.3 Experiment 3: Classifying the Test Set

Below you will find four tables that contain the results from this experiment. In all of them each column will say which class, or which gesture an image actually was, while the rows will say which gesture the program predicted it to be. The cells with blue backgrounds are the ones that were predicted correctly. In all the runs of the KNN algorithm that I have included I have set K = 7. Table 4.1 shows the results of running the KNN algorithm and normalizing the features before calculating the distance. Table 4.2 shows the results of running the KNN algorithm without normalizing the features before calculating the distance. Table 4.3 shows the results of running the KNN algorithm by first standardizing the data by applying the Z score like they did in [20]. Finally, Table 4.4 shows the results of having IPRec recognizing the Test Set.

In addition to the tests displayed in the below tables, I also experimented by using different values for K. However, there was little to no change in how the images were classified, so I decided not to include the results of those experiments.

	Actual class					
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**
Predicted G. 0	10	0	0	0	0	0
Predicted G. 1	1	16	11	8	11	0
Predicted G. 2	0	0	0	0	1	3
Predicted G. 3	0	0	0	0	0	6
Predicted G. 4	0	0	0	0	0	0
Predicted G. 5	0	0	0	0	0	0

Table 4.1: Results of running KNN on the test set with K = 7 with normalization of the features.

	Actual class					
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**
Predicted G. 0	11	4	5	8	3	0
Predicted G. 1	0	10	6	0	0	0
Predicted G. 2	0	2	0	0	0	0
Predicted G. 3	0	0	0	0	8	2
Predicted G. 4	0	0	0	0	0	5
Predicted G. 5	0	0	0	0	0	0

Table 4.2: Results of running KNN on the Test Set with K = 7 and no normalization of the features.

	Actual class					
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**
Predicted G. 0	11	7	9	8	12	9
Predicted G. 1	0	9	2	0	0	0
Predicted G. 2	0	0	0	0	0	0
Predicted G. 3	0	0	0	0	0	0
Predicted G. 4	0	0	0	0	0	0
Predicted G. 5	0	0	0	0	0	0

Table 4.3: Results of running KNN on the Test Set with K = 7 and applying the Z score from [20].

	Actual class					
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**
Predicted G. 0	10	4	0	0	0	0
Predicted G. 1	1	12	0	0	0	0
Predicted G. 2	0	0	10	0	0	0
Predicted G. 3	0	0	1	8	1	0
Predicted G. 4	0	0	0	0	11	0
Predicted G. 5	0	0	0	0	0	9

Table 4.4: Results of gesture recognition with only image processing techniques.

Chapter 5

Evaluation and Discussion

In this chapter, I first state how my two approaches from Sections 3.3.1 and 3.3.2 will be compared. I then evaluate the results of the experiments conducted in Chapter 4. At the end of the chapter I give a discussion on the research questions presented in Chapter 1 in light of all that has been revealed throughout this thesis. Lastly, a discussion on the strengths and weaknesses of my two approaches is given.

5.1 Evaluation Metric

There are many ways to evaluate empirical data such as that shown in Tables 4.1, 4.2, 4.3, and 4.4. One of simplest and most used approach is to calculate precision and recall. Precision measures how many of the gestures classified to be Gesture X were actually Gesture X, and recall measures how many of the gestures that were actually Gesture X was classified by the system to be Gesture X [15]. These two measurements are what I will be using to evaluate the performance of my 2 approaches to hand gesture recognition. It is also how the researchers in [20] have evaluated their algorithms, so by doing it the same way, I will be able to compare my results to theirs. In addition, I will calculate the accuracy of the different results. Accuracy is calculated as the number of instances classified correctly divided by the total number of instances. It is common to confuse accuracy and precision with one another, and also to use the terms interchangeably. However, it is important to note that they are not the same. The easiest way to grasp the difference is by looking at Figure 5.1.

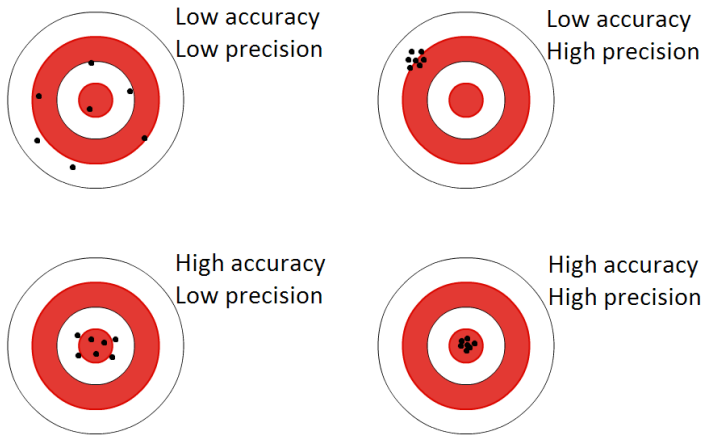

Figure 5.1: The difference between precision and accuracy.

5.2 Evaluation

In Experiments 1 and 2, I tested whether IPRec and KNNRec were able to recognize gestures in real-time. It is important that they can do so for them to be useful for SoundDream, described in Section 1.1. The results of the experiments showed that IPRec was able to process and recognize the gestures from 70 images every second, while KNNRec was able to classify 7 images each second. KNNRec was not designed to recognize gestures in real-time, but to test the performance of an algorithm. There is definitely a lot of room for improvement in terms of data representation, how the data was handled (stored into- and loaded from memory), and how intermediate variables were utilized. Optimizing KNNRec with respect to this, would probably have a huge impact on how many images it would be able to classify each second. According to [4], a response time of 0.1 second is the limit on how long a system can spend before the user does not feel like the system is reacting instantaneously. That being said, I would still argue that 7 images per second is sufficient to perform real-time gesture recognition. If the system was to be used for an action game or something along those lines, 7 actions per second would maybe not be enough, but to control SoundDream, I think 7 actions per second is more than enough. The 70 images per second from IPRec, however, would probably be enough to control any system in real-time.

As I mentioned in the previous section, I will use precision, recall, and accuracy to evaluate the results of Experiment 3. Following are the tables from Section 4.4.3, with the addition of an extra row and column that shows the recall and precision respectively. These numbers are be highlighted in green.

	Actual class						
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**	**Precision**
Predicted G. 0	10	0	0	0	0	0	100%
Predicted G. 1	1	16	11	8	11	0	34.04%
Predicted G. 2	0	0	0	0	1	3	0%
Predicted G. 3	0	0	0	0	0	6	0%
Predicted G. 4	0	0	0	0	0	0	0%
Predicted G. 5	0	0	0	0	0	0	0%
Recall	90.91%	100%	0%	0%	0%	0%	

Table 5.1: Results of running KNN on the test set with K = 7 with normalization of the features. Precision and recall highlighted in green.

	Actual class						
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**	**Precision**
Predicted G. 0	11	4	5	8	3	0	35.48%
Predicted G. 1	0	10	6	0	0	0	62.50%
Predicted G. 2	0	2	0	0	0	0	0%
Predicted G. 3	0	0	0	0	8	2	0%
Predicted G. 4	0	0	0	0	0	5	0%
Predicted G. 5	0	0	0	0	0	2	100%
Recall	100%	62.50%	0%	0%	0%	22.22%	

Table 5.2: Results of running KNN on the test set with K = 7 and no normalization of the features. Precision and recall highlighted in green.

	Actual class						
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**	**Precision**
Predicted G. 0	11	7	9	8	12	9	19.64%
Predicted G. 1	0	9	2	0	0	0	81.82%
Predicted G. 2	0	0	0	0	0	0	0%
Predicted G. 3	0	0	0	0	0	0	0%
Predicted G. 4	0	0	0	0	0	0	0%
Predicted G. 5	0	0	0	0	0	0	0%
Recall	100%	56.25%	0%	0%	0%	0%	

Table 5.3: Results of running KNN on the test set with K = 7 and applying the Z score from [20]. Precision and recall highlighted in green.

	Actual class						
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**	**Precision**
Predicted G. 0	10	4	0	0	0	0	71.43%
Predicted G. 1	1	12	0	0	0	0	92.31%
Predicted G. 2	0	0	10	0	0	0	100%
Predicted G. 3	0	0	1	8	1	0	80.00%
Predicted G. 4	0	0	0	0	11	0	100%
Predicted G. 5	0	0	0	0	0	9	100%
Recall	90.91%	75.00%	90.91%	100%	91.67%	100%	

Table 5.4: Results of gesture recognition with only image processing techniques. Precision and recall highlighted in green.

From the Tables 5.1, 5.2, 5.3, and 5.4, it is easy to see which of my approaches was most successful. IPRec, with results shown in Table 5.4, has an average precision of 90.62% and an average recall of 91.41% across the classes. In comparison the best settings of KNNRec was with K=7 and no normalization of the features, achieving an average precision of 33.00% and an average recall of 30.79%.

I was clearly unable to reproduce the success that the authors of [20] was able to find with the KNN algorithm. It might be that their gestures were more different from one another in terms of the different features than what mine were. However, if I look past the results I got from the KNN algorithm and instead compare my results from Table 5.4 with the results they achieved in [20], I can do a more reasonable comparison of the two approaches. They were able to achieve an average precision of 96.89% and an average recall of 96.94% with the KNN algorithm. This beats what I achieved with IPRec, but not by much. It is worth noting that they had a test set of 1,295 images and 10 different gestures. This test set is considerably bigger than my test set of only 67 images. Having only around 10 images for each gesture, results in every incorrectly classified image having an impact of around 10% on the recall, and also quite a lot on precision. In [20] they have a little more than 100 images for each gesture, which is exactly what I have in my training set for KNNRec. I therefore decided to use IPRec to predict the gestures in my training set to better compare it to what they achieved in [20]. The results of this additional experiment can be seen in Table 5.5.

	Actual class						
	G. 0	**G. 1**	**G. 2**	**G. 3**	**G. 4**	**G. 5**	**Precision**
Predicted G. 0	87	3	0	0	0	0	96.67%
Predicted G. 1	13	95	1	2	1	0	84.82%
Predicted G. 2	0	2	97	1	1	0	96.04%
Predicted G. 3	0	0	2	93	5	5	88.57%
Predicted G. 4	0	0	0	3	89	4	92.71%
Predicted G. 5	0	0	0	1	4	91	94.79%
Recall	87.00%	95.00%	97.00%	93.00%	89.00%	91.00%	

Table 5.5: Results of gesture recognition with only image processing techniques on the training set of KNNRec. Precision and recall highlighted in green.

With this additional experiment I was able to achieve an average precision of 92.27% and an average recall of 92.00%. This is a small improvement to what I got using the test set, but not enough to be as good as the KNN approach in [20].

In Section 3.2 I mentioned a paper [5] where they used only image processing techniques for gesture recognition, and were able to correctly predict the gesture in 82.47% of the images with their best approach. This is the same as accuracy. Calculating the accuracy from the results in Table 5.4, I get 89.55%. Doing the same calculation for the results in Figure 5.5, I get 92.00%.

5.3 Discussion

In this section, I first answer the research questions posed in Chapter 1. After that I give a discussion on how I achieved the research goal of this thesis.

5.3.1 Research Question 1: *Which image processing techniques are best suited for hand gesture recognition?*

Through my research and implementation of various image processing techniques, I have found that a combination of segmentation based on skin color, morphological filtering, extraction of the largest contour, and the properties of convexity defects gives better results on hand gesture recognition than any approach I have been able to find in my research that uses only image processing techniques. In [5] they made 4 different algorithms using only image processing techniques for hand gesture recognition and were able to successfully predict the gesture in 82.47% of the images using segmentation, a median filter, and then scanning the image horizontally and vertically to detect the number of fingers. Papers such as [24, 26, 8, 20], although using machine learning for prediction, also make use of image processing techniques such as segmentation based on skin color, finding the contour of the hand and extracting the convexity defects of the hand. The system I made using only image processing techniques combined all of this and used segmentation based on skin color, morphological filtering, extraction of the largest contour, and the properties

of convexity defects to successfully predict the gesture in 89.55% of the images in my test set, which is an improvement to what they managed in [20].

5.3.2 Research Question 2: *Which machine learning algorithms are best suited for hand gesture recognition?*

Based only on the results from [5] one could say that the best machine learning algorithm for hand gesture recognition is the KNN algorithm. However, other researchers (discussed in Section 3.2) have produced equally good results using other algorithms and other features. It would seem that the choice in algorithm and features depends on how many gestures should be recognized, and how different these gestures are. Also, some algorithms might work better than others if the system should recognize dynamic gestures in addition to static ones. In conclusion it is extremely difficult to crown one machine learning algorithm as the single best.

5.3.3 Research Question 3: *How should the different approaches be compared and evaluated?*

Precision and recall (and accuracy) was chosen as the metrics to evaluate the different approaches. The decision was based on what other researchers had used to evaluate their approaches, as well as my own experience in using these metrics successfully in previous projects. Precision and recall goes well together in describing the performance of a classification system. If we look at the column for precision in Table 5.4, we see that the precision of class/gesture 1 was 92.31%, which is quite excellent. However, this only tells us how many of the images predicted to be Gesture 1 was actually Gesture 1. It doesn't tell us anything about how many of the images that were actually of Gesture 1 were predicted to be Gesture 1. In the row for recall in the same table we see that that was only 75.00%. Combining precision and recall yields a better understanding of the performance of the system than what any of the two could do alone.

5.3.4 Research Question 4: *How does the performance of a purely image processing system compare to the performance of the best machine learning systems when it comes to hand gesture recognition?*

The two approaches both perform very well, but the machine learning approach has the edge. This question has been discussed a lot in the previous section. To sum it up; I was unable to reproduce the success that the authors of [20] had with the KNN algorithm (which in their comparative study turned out to be the best one). In order to do a proper comparison, I therefore compared my system using only image processing techniques for hand gesture recognition with the system using KNN from [20]. It turned out that both approaches were able to successfully recognize the gesture in more than 90% of the images, but with the machine learning approach being slightly better than my approach.

5.3.5 Goal: *Compare methods in Artificial Intelligence to see which is best suited for hand gesture recognition*

Throughout this thesis I have discussed the work done by other researchers in the field of hand gesture recognition. I have described and explained various techniques in image processing and machine learning. I have evaluated all my findings and built two different systems to recognize hand gestures using the AI methods that I deemed most likely to be successful. One of the systems used only image processing to recognize the gestures, and the other used image processing for feature extraction, and then KNN for gesture recognition.

Based only on the results from my experiments, the approach using only image processing techniques was without a doubt the best one. If I also take into consideration all the work that other researchers have done, then classification using machine learning techniques yields a slightly better precision, recall and accuracy. There are, however, strengths and weaknesses to both systems. IPRec using only image processing do not require a huge database of all the gestures that should be recognized. Instead it relies on each gesture being coded in the program. Having each gesture being coded, it makes the task of adding support for more gestures very difficult. To do so a programmer would have to add code to the program, and in the worst case might have to rewrite the entire system. However, with a machine learning approach, you would simply need to expand the training set with samples of the new gesture. Another weakness of IPRec is that a gesture is defined by how many fingers are being held up. This means that it would interpret a hand where only the index finger is being held up in the same way as it would if only the little finger was being held up. If this was meant to be two different gestures there would have to be made big changes to IPRec. Whereas with KNNRec one would only need to put samples of both marked as different classes into the training set.

So which approach is the best? If you have only 6 different gestures like the ones shown in Figure 3.2, you would probably be equally well off by using either of the two approaches discussed. If your system needs to support more gestures, or gestures that are more complex than simply being defined by the number of fingers shown, then a machine learning approach would be your best bet.

Chapter 6

Closing Remarks

This chapter will wrap up my thesis. I will first state what contributions I have made to the field of hand gesture recognition, then I will move on to what future work should be done to improve on my approach should be concerned with, and round it off with the conclusion of this thesis.

6.1 Contributions

With this thesis I have reviewed which image processing techniques are best suited for hand gesture recognition through studying what other researchers have done, and then written a program myself using the techniques I deemed most likely to yield good results. I then compared this system to a machine learning approach to hand gesture recognition that other researchers found success with. This kind of comparison is something I have not been able to find in other research, and I think it offers valuable insight to the topic.

6.2 Future Work

With my two approaches to hand gesture recognition I have prior knowledge about the color of the hand, which allows me to segment it. This segmentation is the foundation that the entire recognition process builds on. If the system should be used in other lighting conditions and by people with different skin color, then the segmentation should be invariant to these conditions. Papers such as [13, 9] discuss the use of the K-means clustering algorithm for segmentation, and I think using that would improve the robustness of the segmentation.

As I approached the deadline for delivering this thesis I came across the SIFT (Scale-invariant feature transform) algorithm [12]. It is an algorithm within the field of computer vision, and it is used to detect and describe local features in images. I think that the features extracted from this algorithm may prove useful for hand gesture recognition, and could be

combined with other features such as those I have looked at in this thesis. In Figure 6.1 you can see how the features extracted from a playing card on the left are matched to features of the right image in order to find and recognize a similar playing card.

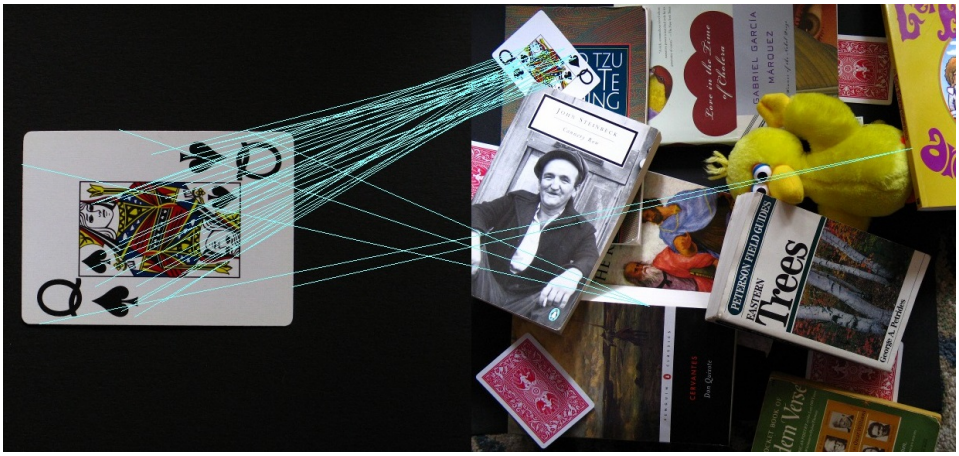

Figure 6.1: Recognition using SIFT features.

6.3 Conclusion

In my research to find the best AI approach to hand gesture recognition, I have found that a system using only image processing techniques can be quite good, but is slightly outperformed by the best machine learning algorithms. Especially if the gestures to be recognized are more complex than the ones shown in Figure 3.2, a machine learning approach would fare better than an approach using only image processing techniques. While some researcher would claim or indicate that their approach and the algorithm they have used is the best one, I have found that it is extremely difficult to crown a winner. Which algorithm performs best depends on what features are extracted from the images, and how many gestures the system should be able to recognize.

Bibliography

[1] Codelaboratories homepage. Available: `https://codelaboratories.com/`. [Accessed: 2015-03-02].

[2] OpenCV documentation for java. Available: `http://docs.opencv.org/java/`. [Accessed: 2015-06-25].

[3] OpenCV forums. Available: `http://answers.opencv.org/questions/`. [Accessed: 2015-06-25].

[4] Response times: The 3 important limits. Available: `http://www.nngroup.com/articles/response-times-3-important-limits/`. [Accessed: 2015-07-02].

[5] P. R. V. Chowdary, M. N. Babu, T. V. Subbareddy, B. M Reddy, and V. Elamaran. Image processing algorithms for gesture recognition using matlab. *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on*, pages 1511–1514, 2014.

[6] W. Du and H. Li. Vision based gesture recognition system with single camera. *Signal Processing Proceedings, 2000. WCCC-ICSP 2000. 5th International Conference on*, 2:1351–1357, 2000.

[7] S. M. Goza, R. O. Ambrose, M. A. Diftler, and I. M. Spain. Telepresence control of the nasa/darpa robonaut on a mobility platform. *Proceeding CHI '04 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 623–629, 2004.

[8] H. Hasan and S. Abdul-Kareem. Static hand gesture recognition using neural networks. *Artificial Intelligence Review*, 41(2):147–181, 2014.

[9] N. Ibraheem, R. Khan, and M. Hasan. Comparative study of skin color based segmentation techniques. *International Journal of Applied Information Systems (IJAIS)*, 5(10), 2013.

[10] W. Ke, W. Li, L. Ruifeng, and Z. Lijun. Real-time hand gesture recognition for service robot. *Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on*, 2:976–979, 2010.

[11] L. Lamberti and F. Camastra. Real-time hand gesture recognition using a color glove. *Image Analysis and Processing ICIAP 2011*, 6978:365–373, 2011.

[12] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[13] M. Panwar. Hand gesture recognition based on shape parameters. *Computing, Communication and Applications (ICCCA), 2012 International Conference on*, pages 1–6, 2012.

[14] S.S. Rautaray and A. Agrawal. Design of gesture recognition system for dynamic user interface. *Technology Enhanced Education (ICTEE), 2012 IEEE International Conference on*, pages 1–6, 2012.

[15] S. Russel and P. Norvig. *Artificial Intelligence A Modern Approach*. Pearson, 2010.

[16] J. Sklansky. Measuring concavity on a rectangular mosaic. *IEEE TRANSACTIONS ON COMPUTERS,*, C-21(12):1355–1364, 1972.

[17] T. Starner, J. Auxier, D. Ashbrook, and M. Gandy. The gesture pendant: a self-illuminating, wearable, infrared computer vision system for home automation control and medical monitoring. *Wearable Computers, The Fourth International Symposium on*, pages 87–94, 2000.

[18] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985.

[19] R. Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010.

[20] P. Trigueiros, F. Ribeiro, and L.P. Reis. A comparison of machine learning algorithms applied to hand gesture recognition. *Information Systems and Technologies (CISTI), 2012 7th Iberian Conference on*, pages 1–6, 2012.

[21] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, 1:I–511 – I–518, 2001.

[22] I. Witten, E. Frank, and M. Hall. *Data Mining Practical Machine Learning Tools and Techniques*. Elsevier, 2011.

[23] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. McLachlan, A. Ng, B. Liu, P. Yu, Z. Zhou, M. Steinbach, D. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[24] Yishen Xu, Jihua Gu, Zhi Tao, and Di Wu. Bare hand gesture recognition with a single color camera. *Image and Signal Processing, 2009. CISP '09. 2nd International Congress on*, pages 1–4, 2009.

[25] Ying Yin and R. Davis. Real-time continuous gesture recognition for natural human-computer interaction. *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pages 113–120, 2014.

[26] B. Zhang and R. Yun. Robust gesture recognition based on distance distribution feature and skin-color segmentation. *Audio Language and Image Processing (ICALIP), 2010 International Conference on*, pages 886–891, 2010.

Appendix A

Examples of feature vectors

A.1 Feature vectors without normalization

Below you find feature vectors extracted by KNNRec from four of the sample images in the training set. There is 1 of Gesture0, 1 of Gesture1, 1 of Gesture2, and 1 of Gesture4. The class of each image is marked by the last feature in each of the vectors. These features have not been normalized.

[35.0167, 771.1317, -0.7895, 55258.0, 1040.3818, 1.0, 0.0]
[35.0220, 803.6573, -0.7942, 48749.5, 1145.9382, 1.0, 1.0]
[37.9369, 1077.3910, -0.8152, 56116.5, 1544.6488, 3.0, 2.0]
[37.8960, 1115.5613, -0.7979, 53824.0, 1579.4844, 3.0, 3.0]

A.2 Feature vectors with normalization

Below you find the feature vectors from Section A.1 normalized through dividing by the largest value found for every feature.

[0.7592, 0.5023, 0.9360, 0.6943, 0.4506, 0.1667, 0.0]
[0.7593, 0.5236, 0.9416, 0.6125, 0.4963, 0.1667, 1.0]
[0.8225, 0.7019, 0.9666, 0.7051, 0.6690, 0.5000, 2.0]
[0.8216, 0.7268, 0.9460, 0.6763, 0.6841, 0.5000, 3.0]

A.3 Feature vectors with Z score applied

Below you find the feature vectors from Section A.1 after applying the Z score used in [20].

[-1.3116, -1.2844, 0.0015, -0.2897, -0.9430, -0.0726, 0.0]
[-1.3097, -1.1509, 9.9385E-4, -0.9517, -0.7574, -0.0726, 1.0]
[-0.2706, -0.0277, -0.0011, -0.2024, -0.05616, 0.0133, 2.0]
[-0.2852, 0.12885, 6.2210E-4, -0.4355, 0.0051, 0.0133, 3.0]