Static Gait Biometrics Recognition Algorithm

Tom Hoad - tdh1g19

April 2023

1 Abstract

We will explore the algorithm designed for a static gait recognition system. The main techniques used include K-means segmentation, silhouette unwrapping, pose estimation, volume estimation, principal component analysis and a nearest neighbour classifier.

2 Introduction

The task here is to create a classification algorithm for detecting stationary Gait features to identify people. This posed quite a challenge, as traditional Gait detection requires moving people, but here we only have stationary targets. This means that we only have two images per training person to learn to identify people, making this very challenging to get accurate. The algorithm and classifier I developed here rely on three main metrics: silhouette, body pose and body proportions. These three become feature vectors that we concatenate and use to classify our people.

To develop this program, I decided to use Java over the perhaps more obvious choice of Python. This is a language I am very comfortable with, and have a base familiarity with a computer vision library called OpenIMAJ [1] which I used extensively during this program. Its main purpose was to be used as a way to represent a traditional image or feature vector without the need for creating my classes and methods to do so. In addition to OpenIMAJ, I have also used a deep learning library called Deep Java Library [2]. This was used for the pose estimation [3] part of the code.

This was all developed using IntelliJ, a platform primarily for Java that allowed easy integration with libraries. It also links up well with version control systems, which I used extensively to checkpoint my code. I used a GitHub repository and made very frequent commits to not only save my code but also record the classifier results at a given time.

3 Method

3.1 Image Segmentation

The first step in the algorithm is to extract the person from the image. The background green screen provides a very beneficial contrast, but we still have to deal with a few bits of clutter. To achieve this, we use image segmentation. The process of this involves using a KMSpatialColourSegmenter to split the image into two classes (foreground and background.)

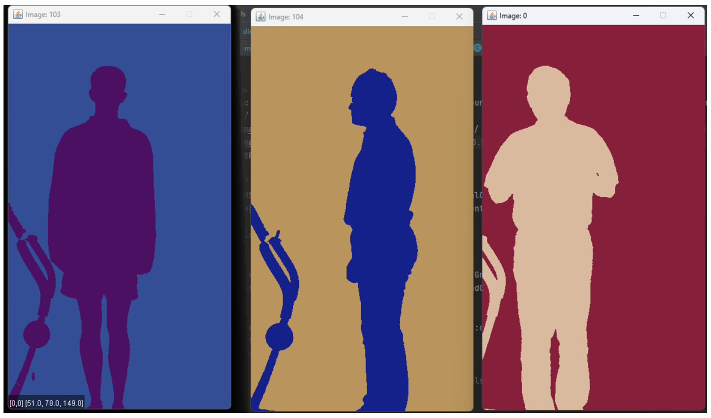

Figure 1: K=2 Segmented Images

Here we can see in the figure above what the result of this segmentation will look like. Once we have this image, we use a component labeller to identify components in each image. Knowing that the largest component will always be the background, we take the second largest to be the person. As we can see in the image above, we have two visible design decisions. Firstly, we have had to crop the image down somewhat. This largely doesn't change anything about the segmentation but does improve the performance. Secondly, we can see that the treadmill is being picked up as the foreground. This will largely be solved by being seen as a third smaller component, but where the person stands in front of it, it gets included as the person. This is not a problem I was able to resolve and has resulted in a lower than preferable final CCR.

This segmented image is used to create a new one, where the person is still visible, but the entire background has been whited out. This image is saved in a folder to record the result of segmentation and is also used for pose estimation and volumetric estimation later.

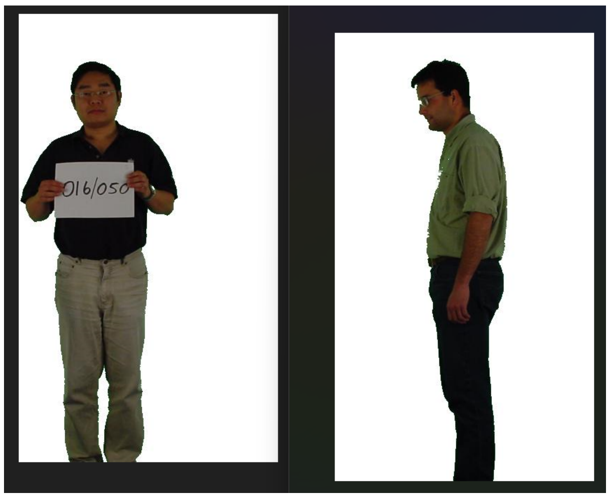

Figure 2: Whited Out Images

3.2 Silhouette Unwrapping

Whilst the white-out image is used for later, we don't need this image directly, just the connected component that represents it. The first part of creating a feature vector is to take the silhouette for each person. A property of the connected component collected is the ability to get the boundary pixels. During development, I found out that the number of pixels in each person varied quite a lot, so I had to develop a way to make a fixed-length feature vector so they could be actively compared between people.

The first step in the algorithm was to convert every pixel from cartesian to polar coordinates. This allows me to better work out the orientation and relative distances much easier. All these polar coordinates are done according to the centroid pixel (to ensure translation invariance) in the person. Once we have a list of polar coordinates, we create a histogram. The histogram used has 48 bins, each representing a range of 6.4 degrees. The pixels are placed into these bins based on their angle. One big problem I faced with the silhouette was the difference between the testing and training datasets. There was a lot of variance in hand position, so I decided to remove part of the silhouette. We have about 50 degrees of lost silhouette (25 on each side at hand level) that removes these invariant distances.

Once the histogram has its bins of distances, we take the average distance from each bin to make up a value in our feature vector. These bins should naturally be ordered by angle beforehand, so each value will line up with the same part of the body when compared to another. Finally, the feature vector is always normalised to adjust for the different sizes of people and to also ensure scale invariance of the same person.

3.3 Pose Estimation

The second part of the feature vector created, is pose estimation. The pose is a key part of Gait and can be used to tell the body proportions very well. We return to the whited-out image created earlier to detect joints on the body. This part of the algorithm makes use of Deep Java Library (DJL) an open-source deep-learning computer vision library for Java. Whilst I have been able to do much of the work by myself with OpenIMAJ tools up till now, an external library was essential for pose estimation.

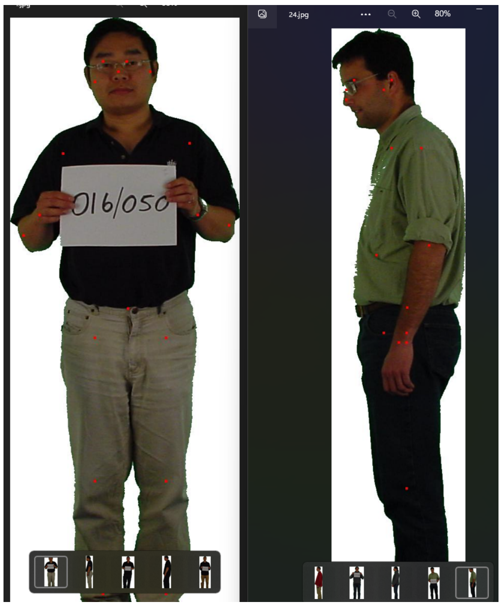

Figure 3: Pose Estimation Images

Shown here are some example results of images produced by DJL. There are several identifiable joints detected here including facial features, shoulders, elbows, wrists, hips, knees and ankles. The person's centroid is also plotted in these images.

From these joints, we find the Euclidean distances between many pairs of related points to form a general feature vector that represents the pose. These distances are split into four categories: distances to the centroid, inter-face distances, body widths and body heights. We have collected a total of 34 unique distances, that make up a feature vector length of 48 (all values from 35 to 48 are zero padding to ensure all feature vectors have a length of 48.) Once again, we normalise the feature vector to ensure scale does not have an effect.

3.4 Volumetric Feature Vector

The final part of the larger feature vector is created by measuring the volumes of an image. In the code, I have largely referred to this as a temperature feature vector/image. This is another method of pose estimation that takes into account the relative sizes of body parts a lot more.

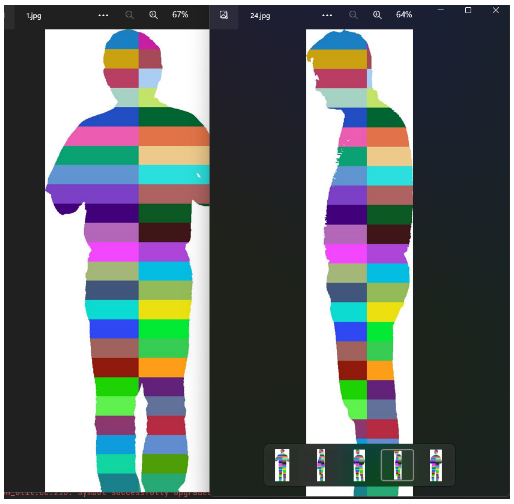

Figure 4: Volumetric 'Temperature' Images

This feature vector is relatively simple to create, when creating the white-out image earlier, we also create this image. Every pixel that represents the person is replaced with a colour that corresponds to a particular area of the image. The colour coding is designed to be invariant of overall height, these volumes will change in shape based on posture, direction, and body proportions. The feature vector is created simply by the number of pixels (the area) of each region. As with the previous two, this feature vector is normalised to help compare sizes relatively rather than absolutely.

3.5 PCA and Feature Vector Comparison

Once I have extracted these three feature vectors, I create a large feature vector by concatenating them all together. This is repeated for every single training and testing image. These have quite a lot of dimensionality as a result of this concatenation, so I have used OpenIMAJs feature vector PCA to reduce this a bit. The PCA learns a basis from the training vectors that it applies to projects to both that and the testing set afterwards. Overall, it doesn't do much to the performance, but it will allow me to graphically display this a bit better.

Unfortunately, we cannot do much to cluster the dataset as we only have two images for each subject. As a result, we have to simply use K nearest neighbours (where K = 1) to find the closest possible match to each image. For each testing image, we take the training image with the smallest Euclidean distance between the two feature vectors. We can then start producing the classification results.

4 Results

The program I developed can calculate the CCR and EER by itself. The classifier finds all the inter and intra distances calculated and uses these to first calculate CCR, but also to programmatically give a fairly accurate EER by working out the threshold that the FAR (False Accept Rate) = FRR (False Reject Rate.) Once we have this list of distances, we can also create a histogram of distances in Excel to represent the EER.

The following results and HoD are taken directly from the latest run of my classifier.

- CCR = Correct Classification Rate (CCR) = **50.0%**
- Equal Error Rate: **18.181818%**
- EER Threshold: **0.043837**
- Duration: **574147ms** ($\approx$ 9 and a half minutes)

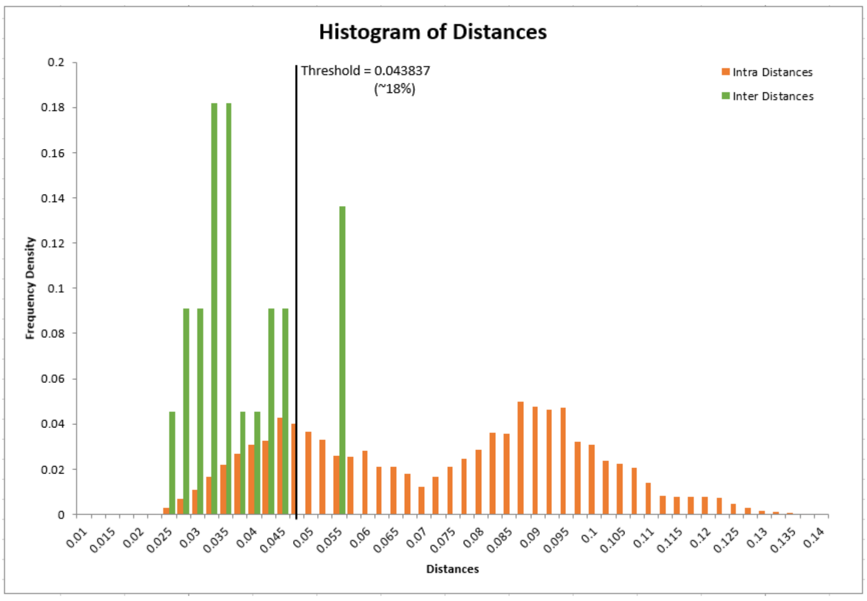

Figure 5: Histogram of Distances

These results are useful in showing that my classifier has a definite correlation, a 50% rate was quite challenging to achieve, but I feel it reflects quite well given the number of training images used. We can tell from the EER that there was room for improvement, but adding a few more bits should boost this CCR much higher.

We can see in the histogram of distances an interesting pattern emerge. Firstly, there is a definite spike in the inter distances above the threshold. We can also see a double hump produced within the intra distances. I believe both values reflect the invariant nature of the treadmill. This was a big problem with the segmentation process that I didn't overcome. These testing images that were incorrectly classified make up a fair amount of the other 50% and being able to remove the treadmill would likely boost these results much higher.

A big pain point in my code was the performance. From this latest run, I had a duration of almost 10 minutes. The process here was very slow and became quite challenging to test slight changes later on. The main problems with performance here were the constant saving of image files and the usage of the pose estimation library. This library ran quite quickly at the start but began to run very slowly late on. I couldn't diagnose if this was an issue on my end or the libraries, but late on I started to encounter external connection issues. I may test this with a better internet connection to see if this speed up the process much at all. As for the rest of the program, the feature vector creation seemed fairly quick in comparison.

5 Conclusions

Overall, I was quite happy with the results produced by the classifier. The main advantage of my solution was feature vector extraction. The variety of feature vectors and invariance to the differences in training and testing were very evident. A few subjects were very consistently classified, and I only struggled with the odd person outside of the treadmill issues. Despite many attempts to remove this, I could never remove the object from some images. This issue occurred the most in the testing images and likely contributed to a drop of 10% to 20% in CCR, maybe more, as this affects all parts of the feature vector.

In conclusion, I would have liked to keep pushing to improve the accuracy, but no amount of additional feature detection methods was going to solve my initial segmentation problem. I am very happy with 50% CCR and would like to have seen how that increased with larger data sets. Different approaches would work better, more specifically learning towards full deep learning solutions. The classification was way too slow to be practical and I'm sure there is more accuracy to get out of it.

References

[1] Open Intelligent Multimedia Analysis (2023) *OpenIMAJ* [Accessed 7/4/23] **http://openimaj.org/**

[2] DJL (2023) *Deep Java Library* [Accessed 7/4/23] **https://djl.ai/**

[3] DJL (2023) *Deep Java Library - Pose Estimation* [Accessed 7/4/23] **http://djl.ai/examples/docs/pose_estimation.html**