**CSC3060 AIDA – Assignment 1**

Adam Coyle

40178464

October-November 2019

# Introduction

The purpose of this report is to record my process of performing a statistical analysis of a set of 160 drawings, from the conversion of images to csv files, to generating a set of features for each image which I will then perform some kind of analysis, representation, and testing on to determine whether or not it might be possible for some machine to discriminate between the different groups of images.

# Section 1

I wanted to get stuck into this assignment, so I sat down for over an hour and produced all 160 doodles. Using GIMP, it was easy to get into a rhythm for exporting small images out as JPEG files. I initially tried to export a test doodle as a PGM file but found that the code I had written was unable to convert it to a csv. From placement experience, I knew how to iterate through a JPEG formatted image and check the RGB elements to determine if a pixel was white or black. Now I could convert an image to csv format, I needed the Java equivalent of the C# LINQ statements, which I could quickly find on Stack Overflow and use to iterate the doodles in a specified directory. Lo and behold, I had 160 csv files.

# Section 2

For your information, I am not in any way proud of the code written for this section. It is full of redundancies and violates a ton of programming patterns, but I’m told that I’m not assessed on how elegant it is, so it was good enough to have working code rather than pretty code. I had most of it done the week after the assignment was sent out, and after that it was just a month of resisting the urge to waste time refactoring. I commented it as best I could to explain what was happening, but if something isn’t clear then I apologise.

## Section 2.1 – nr\_pix

This feature was a nice and easy start into the nitty gritty of Section 2. Thankfully, the csv files only contained 1s and 0s where 1s were to represent black pixels. To calculate this feature, I used a double for loop to iterate through the contents of the csv file, and for every 1 I encountered, I increased the blackPixelCount by 1, finally returning the sum of black pixels in the file.

## Section 2.2 – height

I was surprised at having to use some degree of logical thinking this early into the section, and somewhat anxious about what was to come after. After seeing that the previous feature was calculated by iterating through a 2d array, I decided to stick with this as my main method of calculating features. To start, I created two int variables, topmost and bottommost and initialized them to -1 (since arrays don’t have indexes of -1).

After the creation of these variables I iterated through the array, and for every black pixel I encountered, I checked the row indexes against the values of topmost and bottommost.

* Since I iterated from the top-left to the bottom-right, I could assume that the first black pixel would be at the top of the image, so I assigned this to topmost only if topmost was less than 0 (making the assumption that the first black pixel had the lowest row index – making it the top of the image).
* The value of bottommost was simply the lowest row index encountered during the iteration, so a check to see if the row index value was greater than bottommost was enough to find this.

Finally, with the values for topmost and bottommost, I was able to calculate the height by getting the difference (absolute value) between topmost and bottommost and adding 1 to the result (since arrays begin at 0)

## Section 2.3 – width

The calculation for the width feature was identical to that of the height, just turned on its side. I had two int variables, leftmost and rightmost. The difficulty here was that leftmost needed to be initialised to the value of the array size (50) such that it could be assigned to columns that had a lower value than it (i.e. to move left in the array). The rightmost variable followed the pattern of bottommost from the previous feature, such that it had the index of the rightmost black pixel.

As with the height calculation, width was the absolute difference of the leftmost and rightmost plus 1 to account for the array offsets.

## Section 2.4 – span

This feature took some careful thought to implement. I knew that I would need to iterate over every item, *for every item*, but I realised that I only had to check for every black pixel, since white pixels were irrelevant. Therefore, I found it necessary to add a helper function to return a list of the black pixel indexes. A list became essential since using an array requires a predetermined size, which I wouldn’t know until the end. I could loop through each black pixel, checking its Euclidean distance to the other black pixels, returning the greatest distance as the span.

## Section 2.5 – rows\_with\_5

This feature was straightforward to implement. I initialised an int, numberOfRows, to 0. This would be value returned at the end of the function call. For every row in the array, I initialised another temporary int, blackPixelCount, and proceeded to iterate through the columns of the array. If a black pixel was found, the value of blackPixelCount was increased. If this value was greater than or equal to 5 then numberOfRows would could incremented. Once the iteration had finished, the value of numberOfRows was returned.

## Section 2.6 – cols\_with\_5

The implementation for this feature was identical to that of the previous feature, the only difference being that when accessing the values of the array, the column and row indexes were swapped so that the loop would go down the column instead of across the row. So, the same logic as before applied and the function returned the value of numberOfColumns.

## Section 2.7 – neigh1 and neigh5

As before, I begin the feature calculation with an iteration over the entire array. I also create two int variables, pixelsWithOneNeighbour and pixelsWithFiveNeighbours. If a black pixel was found during the loop, then I would have to check all 8 neighbours for black pixels and increment the variables appropriately. This was where I really became conscious of making reusable code for scanning the neighbours of a tile.

I wrote a helper function called CountNeighbours, that took in the row and column index of the current pixel in the loop. I would have to loop over everything from the top left (column-1, row-1) to the bottom right (column+1, row+1) to check for black pixels neighbours and increment a neighbourCount value for each one. This is where I experienced a real issue which was that the pixels at the edges of the array would throw out of bound exceptions when trying to access some of their neighbours. Now I needed yet another helper function to produce a solution – IsNeighbourValid

The IsNeighbourValid function simply checked if either the row or column to be checked was outside of the array (less than 0 or greater than or equal to the size of the array). Once this function was implemented, I could check all neighbours of a pixel without any issue, and thus the rest of the function would count the neighbours and return the result.

## Section 2.9 – 2tiles,

This section encompasses features 9 (left2tile) to 14 (horizontal) inclusive, as two features are the result of two of the 2tile features, and all 2tile features are calculated similarly. I found it quite difficult to reason about the calculation of a 2tile, and initially it boiled down to consider an iteration over all the pixels. However, I eventually realised that I would only need to iterate through the black pixels.

The algorithm for calculating a 2tile involved iterating through the black pixels, checking their non-diagonal neighbours for black pixels, and then checking the appropriate neighbours for the presence of two white tiles. If a 2tile is found, then the black pixels are marked such that all 2tiles of any variant contain 2 unique black pixels. This algorithm was effectively repeated, with the only changes to detect for a certain 2tile were the row/column indexes of neighbours. In hindsight, I see that the function is excessively long, redundant and the type of thing Robert Martin would advise against in Clean Code. However, for the purpose of calculating the feature set, it was enough.

## Section 2.15 – Custom Features using 3tiles

Regarding the creation of two custom features using 3tiles. I want to say now that I wasn’t very creative with these features and couldn’t really think of any custom features that would discriminate between images. I had every other feature completed a week after the assignment was created, but I only created these custom features the week before the deadline out of necessity.

I decided on implementing features for horizontal and vertical 3tiles.

## Section 2.17 – nr\_regions

This feature was challenging, but also enjoyable to calculate. I took up a notepad and sketched up some dummy data to simulate how I might find the number of regions in the image. When I finally found a potential solution, I opened IntelliJ IDEA and got cracking.

As with previous solutions, I created a 2d Boolean array to indicate whether I had already visited the pixel in question. I then iterated through the array in my usual fashion and only continued if I came across a black pixel that hadn’t be marked. If this was the case, I would then proceed to mark the pixel and call the recursive function MarkAllBlackNeighbours.

The feature does exactly what it sounds like it does – checks the neighbours of a black pixel for neighbouring black pixels, marks black neighbours, and then calls itself on said neighbour. The logic was that for every unmarked black pixel I came across, I would exhaust all its neighbours recursively and mark each one so that the make loop could not begin the process again. Therefore, I could safely assume that the unmarked black pixels in the main loop were the beginning of a new region, since doing so would mark their black neighbours and so on.

## Section 2.18 – nr\_eyes, hollowness and (custom) image\_fill

Despite completing the previous feature relatively easily, I couldn’t find the same success for this feature. Only by asking the Computer Science group on Facebook did I get the answer I needed (big thanks to Stephen McVeigh), which was the count the number of *white* regions in the image. So simple, yet I couldn’t think of it on my own. However, upon closer analysis of the idea, I noticed several pitfalls with the function that had to be accounted for:

* The algorithm would have to count the white space outside of the image, so the actual number of eyes in the image would always be 1 less than what the algorithm calculated.
* As per the assignment pdf, two neighbouring white pixels might not be in the same eye, so I would need to check for black pixels.
* It would also be smart to keep count of the number of white pixels in each eye for the hollowness feature, so I would have to adjust the algorithm for this.

Like the previous feature, I created a 2d Boolean array to mark white pixels which had already been visited. I then created an array list to keep count the number of eyes and the number of pixels in each eye. The function then loops through the pixels and checks for white, unmarked pixels, marks them and adds the function return of MarkEyes to the list.

Mark Eyes has several changes on the Mark Neighbours from which it was adapted. It has an int size, which is initially set to 1 to indicate the pixel from the main loop beginning the region. It then checks its neighbours for white, unmarked pixels, and does either one of two things if a candidate if found:

* If the neighbour is diagonal, then the two non-diagonal neighbours cannot both be black. If that is the case then the white diagonal neighbour is marked and the function is called again, with the result being added to the size.
* The neighbour is non-diagonal, so it is marked, and the function is called again, with the result being added to the size.

This would return the pixel size of each eye to the array list. Now I can assume that the first white pixel belongs to the white space around the image, so I stored the value of the first element before removing it. The number of eyes then was the size of the list, and to calculate the hollowness, the values for each element were added and then used to divide the number of black pixels in the image.

The image\_fill feature was calculated using the pixel count of the image (white pixels in eyes and black pixels) divided by the pixel count of the image.

# Section 3

## Section 3.1 – Histograms for nr\_pix, height and cols\_with\_5

Since this was the first question that would require the feature data, I first needed to import the data and ensure there were no issues. Following on from that, I needed to split the feature data into two distinct groups for living and non-living things, which I did so by using vectors of strings to select certain rows from the table. Then, to spare myself writing 9 lines of code for histograms, I wrote a small function, poorly named one\_to\_three\_histogram (i.e. histograms for features 1-3), to generate the required histograms.

The following page shows the calculated histograms along with my observations of the data.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| Full Set of Things | nr\_pix | height | cols\_with\_5 |
| Shape | Bi modal | Unimodal | Bi modal |
| Skew | Symmetric | Symmetric | Right Skewed |
| Normality | Not normal | Normal | Not Normal |

A picture containing screenshot

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA close up of text on a white background

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| Living Things | nr\_pix | height | cols\_with\_5 |
| Shape | Bi modal | Unimodal | Bi modal |
| Skew | Symmetric | Left Skewed | Uniform |
| Normality | Not normal | Normal | Not normal |

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| Non-living Things | nr\_pix | height | cols\_with\_5 |
| Shape | Bi modal | Unimodal | Bi modal |
| Skew | Symmetric | Symmetric | Right Skewed |
| Normality | Not normal | Normal | Not normal |

## Section 3.2 – Summary statistics for all features

For each feature in the table, I would calculate the mean, standard deviation and the variance. I wrote the function summary\_dataset (), which iterates through the datasets columns, calculating the mean , sd and var for each, before creating a table with the summary statistics as columns, and the feature names as the rows.

From left to right: both groups, living, non-living

A picture containing wall

Description automatically generatedA picture containing wall

Description automatically generatedA picture containing wall

Description automatically generated

From observation, I would have thought that the following variables might be useful for discriminating between living and non-living things:

**cols\_with\_5 (discrete), neigh5 (discrete), nr\_eyes (discrete), hollowness (continuous)**

In order to validate these assumptions, I produced histograms for each variable and each feature set (living and non-living).

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

A close up of a map

Description automatically generatedA close up of a map

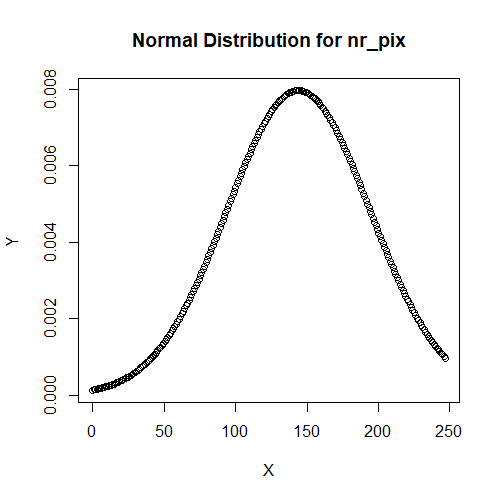
Description automatically generated

To my surprise, the distributions were all similar in shape, but the range of values and the frequency of occurrences for the nr\_eyes and neigh5 features suggest a difference between the two groups which could be used to discriminate between them.

## Section 3.3 – Theoretical Normal Distribution for nr\_pix

As per the task description, I could assume that the nr\_pix variable was sampled from a normally distributed population. I calculated the variance and mean for all 160 objects. Afterwards I produced the normal distribution and the histogram on separate graphs, and then a separate graph with the normal curve overlaid on the histogram.

A screenshot of a social media post

Description automatically generatedA screenshot of a cell phone

Description automatically generated

## Section 3.4 – Cut-off value for the nr\_pix

I made use of the qnorm () function here to determine the cut-off value, and therefore needed the p value which was obtained from 1 – 0.05 = 0.95 (since we are looking for the smallest value of the last 5%). The result of the function yielded a cut-off value of 1.65 (result of qnorm – mean / Sd).

A screenshot of a cell phone

Description automatically generated

## Section 3.5 – Assessing the Normality of all features

I iterated through all the required features (nr\_pix – horizontalness) and calculated their corresponding histograms. For the most part, the results showed most features had either a normal distribution or a slight skew in normality. There were exceptions with the histograms for both the horizontalness, verticalness and both neighbour features. I made use of the “e0171” library which had a skewness function. If the skewness of a given feature was greater than or equal to 1, then the index of that feature was added to a vector of indexes which were subsequently transformed. I ran into issue using log transformations on the neighbour features (non-finite numbers don’t work well for plotting) so I added a check for finite values, if this was the case I could safely use a log base-10 transformation, otherwise I used a square root transformation.

The bottom2tile feature was able to be transformed using a log base-10 transformation, and the strength of the transformation clearly shows a transition from a strong right skew to a much-reduced skewness. The distribution now takes on a far better symmetrical shape.

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

The neighbour features below threw errors as mentioned above, so I had to change the type of transformation to a square root transform. It was clearly a much weaker transformation than the log base-10 and halved the skewness of the original distribution by roughly half. Therefore, the neigh5 feature failed to sit within the margin of +/- 1 required to be normally distributed.

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

## Section 3.6 – Investigating Linear Association between height and span

I used a scatterplot to visualize the relationship between the height and span features. At a first glance, there isn’t anything interesting about the result, with no distinct correlation observed between the features. It did look as if there was a clustering towards the bottom of the distribution, which then progressed to more sparsely plotted values.

Before checking the actual correlation between the features, I needed to clarify the null and alternative hypotheses:

* H0: There is no correlation between the height and span.
* H1: There is some correlation between the height and span.

To get quantitative data about the relationship between the features, I used the cor.test () function using the Pearson method to get a value of 0.07842437. Therefore, I could reject the null hypothesis and conclude that there was a positive correlation (albeit very weak) between the height and span.

A screenshot of a cell phone

Description automatically generated

## Section 3.7 – Usefulness of nr\_pix to discriminate between non-living objects

As the question required me to work with four different groups, ANOVA was the obvious choice of statistical test. I could also assume that there was no need to take a sample of the data, since I had the population data for the 80 non-living things.

For this test, I had decided on the following hypotheses:

* H0 – There is no significant difference between the four objects
* H1 – There is a difference between at least two groups

To begin the test, I produced a box plot for each group to visually assess the variations in the number of pixels. The results were interesting in that they already suggested the alternative hypothesis would be accepted as there was a large variation in the number of pixels for the non-living things.

Initially, I used the aov function to produce the desired results table. However, I had observed a significant F value (and thereby small p value) which would disprove H0. Concerned for the validity of the results, I carried out the test manually.

1. Generate a data frame of statistics for the non-living things (means, Sd and length (n))
2. Calculate the df values df1 and df2, from the length of number of row names in the table minus 1, and the sum of the length (n) of each group minus 1 minus df1.
3. Calculate the variance between groups (n (mean – total mean) ^2) for each group and summing the results.
4. Calculate the tot variance (nr\_pix – mean(nr\_pix) ^2) by iterating through the full table and summing the results
5. Get the Group and Error means from the values for the two variances and the df values
6. Compute the F value and the corresponding p values using pF ()

To my surprise, I calculated the exact same result as the aov() function, so I could assume there was no error in the calculations, reject the null hypothesis H0 based on the large F/small p value and conclude that nr\_pix is a useful discriminating feature for non-living things.

A screen shot of a person

Description automatically generated

## Section 3.8 – Usefulness of hollowness to discriminate between non-living objects

I understood the description of the question was that we had to test to see if the hollowness feature was useful for discriminating between the non-living things using randomization. I had a difficult time understanding how to use randomization with ANOVA, but I eventually came to find a solution and produce some result.

Hypotheses

* H0 – There is no significant difference in means for the four groups of non-living things.
* H1 – There is a significant difference in means for at least two of the non-living things.

I decided to compute ten-thousand randomization tests. This was mainly to strike a balance between quick computation (approx. 10 seconds) and conducting enough tests to produce an accurate set of results. I also experimented with one hundred thousand tests; however, it took considerably longer (approx. 1-2 minutes) and I discovered that the distribution of F-values was identical to the smaller test sample.

A picture containing screenshot

Description automatically generatedThe randomization test sampled the hollowness feature values from the non-living things data frame. From there it assigned every 20 sample values (out of 80) to the name of one of the non-living things (Pencil, Envelope, Wineglass and Golf club) and then computed the F-value for the test using the aov function. This result was then stored in a vector which, after the tests had completed, would produce a normal histogram of the distribution of F-values for the ten thousand tests.

The test results produced an extremely right skewed distribution, with the average F-value being roughly 1.02 (3 s.f) compared to the actual, non-randomized F-value of 1201.

A picture containing screenshot

Description automatically generatedWhen trying to visualize the positioning of the observed F-value, the line plotted at this value, rather unsurprisingly, would not display on the graph. For the sake for representation, I plotted a line representing the observed value using qnorm (p = 1) to indicate its off-plot position, since the result of pnorm (observation, random data) calculated a value of 1.

A close up of a screen

Description automatically generated

## Section 3.9 – Features that may discriminate between living and non-living things

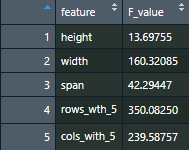
A picture containing text

Description automatically generatedI had decided to conduct a t-test on each of the feature variables 1-22, comparing the two groups living and non-living things, and place the results (means, differences and t-values) into a table.

The neigh5 feature revealed the highest difference in mean of all the features, and its t-value was somewhere middling among the others. However, I would be cautious of taking these results at face value, since as we have seen previously, the distribution for the neigh5 feature was extremely skewed to the right and had to be square root transformed since it threw errors with the log transformation. Even with this transformation, the feature remained too skewed to pass any assumptions of normality required for statistical tests. So this would not be a valuable feature for discriminating between the two groups.

After the neigh5 feature, the highest difference in means was the left2tile feature, which, although its visualisation was not included in this report, was normally distributed. It could be argued that any feature which produced a difference in means greater than 5 could be considered a useful feature, but one would need to assert the normality of the feature before jumping to such a conclusion, since as I have just stated, the neigh5 feature looked to be significant, but its distribution was not normal.

## Section 3.10 – Calculating differences in means for features 2-6

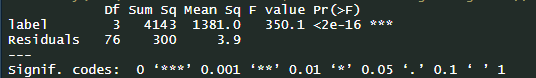
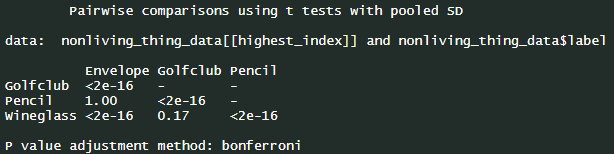
Replicating the same test in (7), I managed to build a table containing the F-values of each ANOVA test on the features 2-6 for the non-living things.

The table shows that the rows\_with\_5 feature clearly has the largest F-value by a large margin, so I had completed the rest of the task using this feature.

From a side-by-side comparison of the two test results (ANOVA and pairwise t test – using Bonferroni correction), I observed that for the most part, the p-values for the t test produced the same value as that of the ANOVA test, with a minor exception in the comparison between Wineglasses and Pencils, and a large value for the comparison between Pencils and Envelopes.

A screenshot of a cell phone

Description automatically generatedTherefore, the result of the ANOVA test confirms at least one of the means is different, however as we can see in the results table for the pairwise comparison, this is largely the case, there is no difference between the means of then Pencils and Envelopes.



# Conclusions

From observation of the statistics reported in section 3, it is possible to discriminate between different groups of things, living and non-living as well as their respective subgroups. However, I would question the validity of my own results since I am in no way a statistical analyst of any sort, and the tests I have conducted on the feature set will be prone to minor or major errors in calculation, in addition to the feature set being base off of my own doodles and code. That said, I will conclude that I have reported as best I can on the doodle features and presented an interesting analysis of the results.