A-Level Computer Science Programming Project

Luke Williams, Candidate 2276
The Cherwell School, Centre 62309
OCR Computer Science A-Level — H446

Contents

1. Analysis

1.1 Introduction

1.1.1 The Problem

For my project, I have decided to train a neural network to predict the Human Development Index (HDI) of a given area. HDI is a number between 0 and 1, which represents how developed a certain country is. It is officially calculated each year by the United Nations, based on factors such as mean years of schooling or life expectancy, by finding the geometric mean of the Life Expectancy Index, the Education Index and the Income Index, as shown in the formula

$$\begin{aligned} \text{HDI} &= \sqrt[3]{\text{LEI} \times \text{EI} \times \text{II}} \\ &\approx \sqrt[3]{\frac{L-20}{85-20} \times \frac{\frac{S_M}{15}+\frac{S_E}{18}}{2} \times \frac{\ln G - \ln 100}{\ln 75000 - \ln 100}} \\ &= \sqrt[3]{\frac{(L-20)\,(18S_M+15S_E)\,(\ln G - \ln 100)}{35100 \ln 750}} \end{aligned} \tag{1.1}$$

where L is the life expectancy at birth, S_M is the mean years of schooling, S_E is the expected years of schooling and G is the gross national income per capita.

However, these factors are very difficult to determine, for many reasons. For example, life expectancy at birth is defined as how long a newborn is expected to live if current death rates do not change. However, the actual death rate for any particular birth cohort cannot be known in advance, and therefore must be estimated on the basis of surveys and historical data, which makes it unreliable. Gross national income per capita also has limitations in its calculation, its accuracy depending on the availability and reliability of income and population data.

Motivated by this, I will instead be investigating how more geographical factors (which can be found by only looking at a map of the area) can affect HDI. The data for which these factors are based on will be pulled from OpenStreetMaps (OSM). OpenStreetMaps is a large data set that contains ~ 1.9 terabytes of information about the Earth. For example, it contains the location of every (registered) park bench, cafe, parking space, ATM machine, post office, school and many others. An API can be used to access this data called Overpass Turbo, in which you can specify an area and what you are looking for, and it will return a list of latitude and longitude coordinates correct to 7 decimal places, along with any extra information.

Therefore, the problem I am solving is that of trying to predict the HDI of a given area based only on physical features of that area (which may be natural or man-made). This is useful because once the HDI has been predicted, suggestions can be made to improve it, which will involve building tangible objects (such as schools), instead of vaguely suggesting "increase the mean years of schooling".

1.1.2 Amenable to Computational Approach

This problem is amenable to a computational approach, as it involves inputs (various geographical factors), outputs (a predicted HDI) and calculations (calculating the predicted HDI from the set of geographical factors through a neural network, which involves many matrix multiplications).

This problem can also be solved in a finite, reasonable number of steps. Most algorithms that solve problems in a finite but unreasonable number of steps often involve a brute-force approach of trying all possibilities for a given problem, which I do not intend to implement.

This problem will also involve a vast amount of data, gathered by OpenStreetMaps, in the form of ordered latitude and longitude pairs. The storage requirements for this data however will not be so large, as I will only be dealing with (relatively) small amounts of data at any given time.

1.1.3 Computational Methods

Abstraction

I intend to use abstraction in my project to remove any unnecessary detail from it, in order to make problem solving easier. For example, I will abstract away specific details for what my neural network is going to do, to focus on writing the infrastructure for a general neural network, which can be tuned to do anything (including what I want it to do).

I will also be using abstraction to make models when solving particular problems, in order to make them simpler. For example, if the problem requires some geometry in the Earth, it helps to abstract the Earth into a perfectly smooth sphere, instead of considering its rough topography, or its slight eccentricity.

The data that will be fetched from OpenStreetMaps will also be abstracted, as all I will be considering is the object's latitude and longitude coordinates, as well as what type of object it is. There is no need to consider any other attribute about the object, such as its size, its material, or its history.

I will also use abstraction to remove unnecessary details from the map the users will be selecting their areas from. Details which are given by satellite imagery are not required for the user to be able to recognise and select the area they want to analyse. Therefore I will only be using a digital map for users to select from.

Decomposition

Decomposition is the process of breaking down a problem into smaller subproblems, each of which are easier to solve. I will use decomposition to break down my project into 3 distinct

and independent modules: the collection of data, the neural network, and the user interface. These modules have almost nothing to do with each other, and can be considered completely separately. The only times they interact are when the neural network reads the gathered data, and when the user interface queries the neural network.

Furthermore, the purpose of hidden layers in a neural network (further described later in section 2.3.1) is inherently to represent decomposition. Calculating the HDI from a set of seemingly unrelated geographical factors is very difficult, and so the hidden layers attempt to calculate nebulous intermediate values in between the geographical factors and the HDI.

In other words, making use of a neural network with n hidden layers decomposes the problem of calculating HDI from a set of geographical factors into

- Calculating nebulous intermediate value 1 from set of geographical factors
- Calculating nebulous intermediate value 2 from nebulous intermediate value 1

$$\vdots$$

- Calculating nebulous intermediate value n from nebulous intermediate value $n-1$
- Calculating HDI from nebulous intermediate value n

with each nebulous intermediate value representing something closer and closer to HDI with each one. What the nebulous intermediate values actually represent are too abstract for us humans to understand.

Divide & Conquer

Once the HDI has been successfully predicted, I would like to make suggestions on how to improve it. The HDI will then be predicted again based on the improvements, and so to determine the best one, the different improvements must be sorted. An algorithm such as quicksort is suitable, as it makes use of the divide & conquer method. This is because it considers a problem smaller in scope but of the same format each iteration, which makes for an efficient and intuitive algorithm.

Data Mining

Data mining is a technique used to identify patterns and/or anomalies in large data sets (big data), by means of statistics or machine learning. This most certainly fits the description of my project, as the large data set I will be using is OpenStreetMaps ($\sim$ 1.9 TB), and I hope to recognise patterns within the locations of geographical objects, and how that may correlate with HDI. I plan to do this by making use of a neural network, which falls under machine learning.

Heuristics

It is impossible to *actually* calculate HDI from a set of geographical factors, as how you *actually* calculate it is with more social factors (as shown in equation 1.1). Therefore a

heuristic "good enough" approach with neural networks must be used. I don't strictly plan on using heuristics, however in order to train a neural network, you must have a heuristic sense of how much each particular neuron will affect the output of the system (further described later in section 2.3.3).

Visualisation

For the user, I will make use of visualisation to show them various statistics. For example, I may implement a bar graph showing which suggestions will increase the HDI the most, or a map to show where new buildings should be placed in order to have the biggest effect.

For me, I will make use of visualisation in order to make solving problems easier. I plan on using diagrams to depict the various processes going on inside a neural network, and to further my understanding. Diagrams will also help in deriving different geometric equations, which will be useful throughout this project.

1.2 Stakeholders

My stakeholders described themselves as:

1. **Umanga Shrestha** – An A-Level Student interested in moving out of the UK
2. **Joel Tew** – An A-Level Student interested in Simulating Countries
3. **Victor Faustino** – A Local Council Member
4. **Kiefer Keyworth** – An A-Level student who has not studied Geography since Year 9

When asked how they would make use of my proposed solution, and why it is appropriate for them, they responded with:

No.	How they would make use of my Proposed Solution	Why it is appropriate to their needs
1	He will use it to understand the HDI within different areas. He will then use this information to decide where to move.	It is appropriate because he is very interested in HDI in different areas of the world and would love to provide help to increase it.
2	He will use it to estimate the HDI of a hypothetical country, Flabbistan (which the UN does not recognise). He believes that this will be the most developed "country" in the world.	It is appropriate because it will make his simulations involving Flabbistan more realistic, by providing an insight into how it may develop in the future.

3	As a local council member, he would love to use it to suggest changes to his local area that would raise the HDI.	It is appropriate because it would suggest changes to increase HDI while analysing the importance of each of the factors. This would help him to choose between different changes based on cost.
4	He will use the solution to gain a better understanding of what HDI is. He will be able to see the relevance of many factors (in particular, the number of park benches) to the development of a country.	It is appropriate because he is interested in geography but did not take it at GCSE or A-Level and is therefore in need of a program related to it. This solution will be very useful in showing how various factors correlates with development.

Table 1.1: Stakeholders

Each stakeholder's more specific needs and ideas will be discussed in the Stakeholder Interview (Section 1.3.2).

1.3 Research

1.3.1 Existing Solutions

sandeepponduru's human-development-index [1]

This solution makes use of a Jupyter Notebook (`.ipynb`) to predict the HDI (Figure 1.1), which allows for small python code snippets to be run independently, along with shell commands. They have used this to train the model as they go, and then use it for predictions and data visualisations in the same workflow.

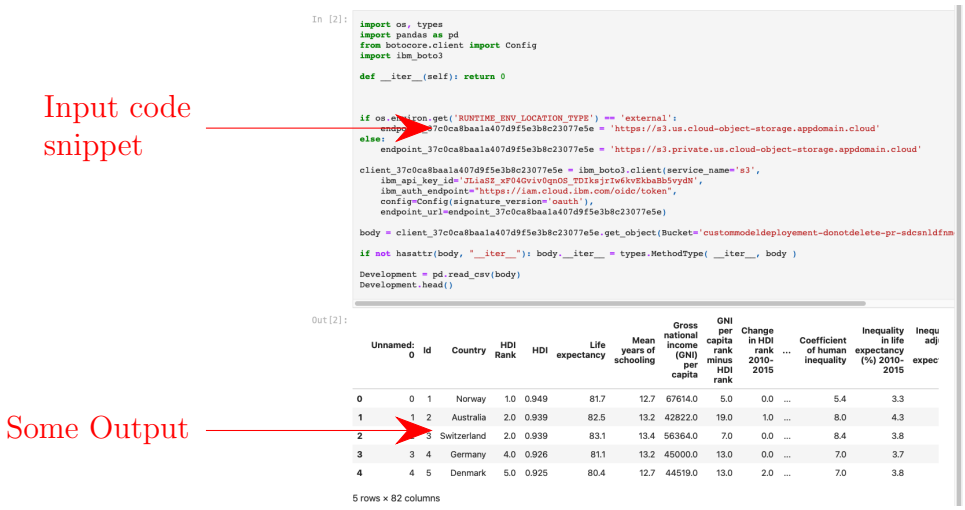


Figure 1.1: Jupyter Notebook Format (Not Suitable)

However, for my project, my model needs to be pre-trained and then accessed via a GUI, so a Jupyter Notebook is not suitable.

This solution also has a scatter plot showing the HDI of various countries (Figure 1.2). It includes a select few countries, which have an HDI ≥ 0.90, which are each represented by a coloured dot. (The specific colour seems to carry no meaning)

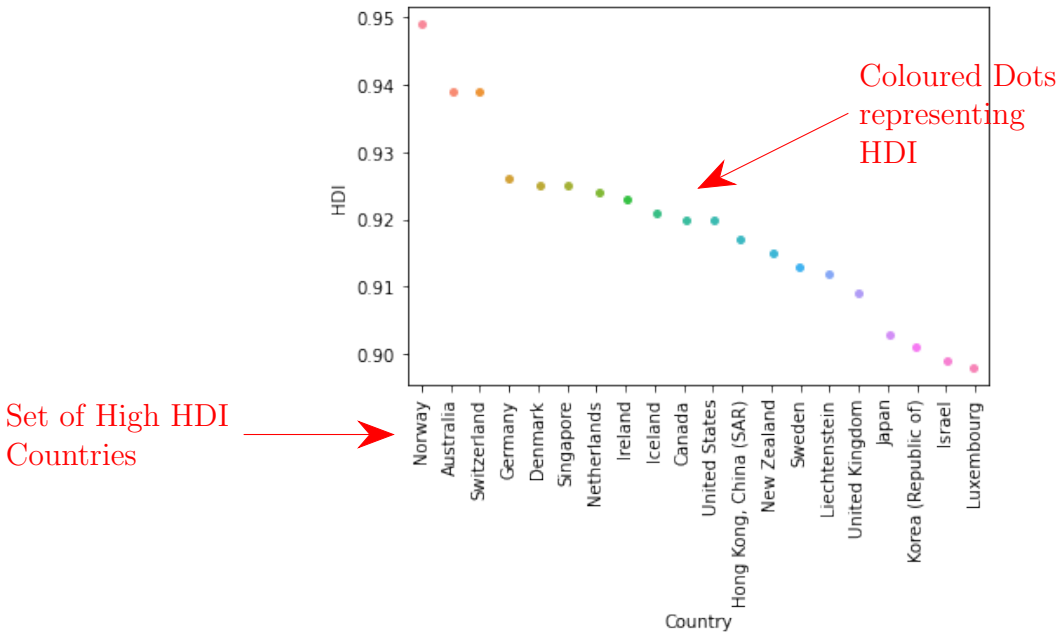


Figure 1.2: HDI Scatter Plot By Country

I believe that a bar graph would be more appropriate, as the x-axis is a discrete variable (which country) and the y-axis is continuous (HDI). This would be good to implement in my project as the HDI in the user's area could be compared to the HDI of other countries or

areas on a similar graph to this, so that if they don't fully understand the HDI scale, they can easily compare with what they know.

This solution also includes the use of a heatmap, describing the relationship between each factor they consider (Figure 1.3). Brighter cells represent strong positive correlation between the factors, and darker cells represent strong negative correlation.

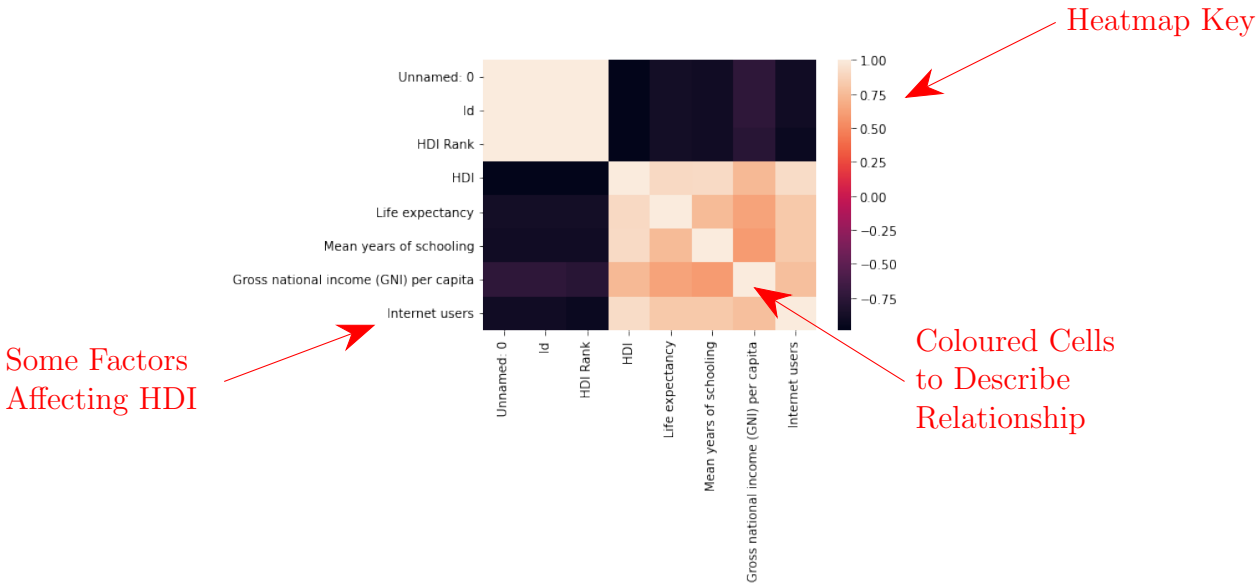


Figure 1.3: HDI Factor Heatmap

I think it would be useful to implement this into my solution, as it would inform which changes the user needs to make to their area, in order to not only increase the general HDI, but also perhaps will improve some other things, which will even further boost the HDI.

Both of the suggested features here (Comparison of predicted HDI to other countries & Analysis of how much each factor affects the others) will be discussed in the Stakeholder Interview (Section 1.3.2).

julieanneco's predictingHDI [2]

This solution also has a similar heatmap feature, but here each cell has a pie chart showing the correlation coefficient (Figure 1.4). The proportion of the pie that is filled carries the same meaning as the colour of the pie, with more filled in pies indicating stronger correlation. This chart also omits half of the possible combinations, as they have already been accounted for (for example, correlation between birth rate and life expectancy will be equal to the correlation between life expectancy and birth rate).

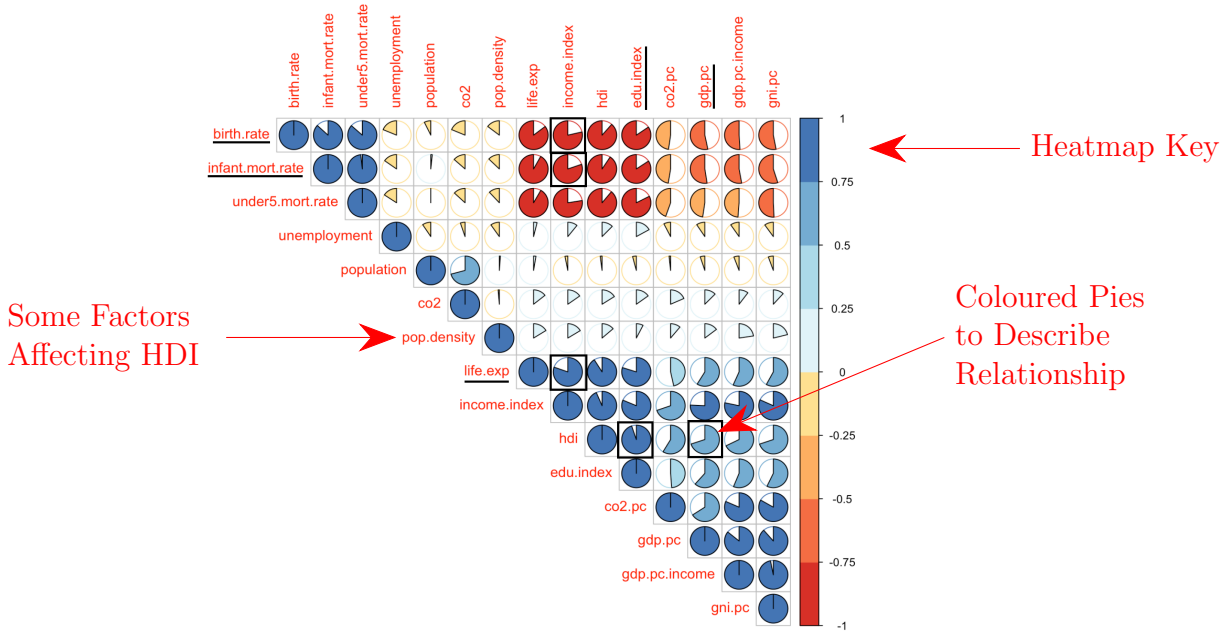


Figure 1.4: HDI Factor Pie Chart Heatmap

While omitting half of the combinations of factors is efficient, I believe the use of pie charts in each cell only makes this chart unnecessarily more complex and harder for someone to understand. If I were to add a feature similar to this, I would only include half of it (as shown in this solution), and fill the cells with solid colour (as shown in the previous solution).

The factors that this solution take into account are more social, economic and environmental factors (Figure 1.5). They have also taken into account multiple time periods (years).

Social, Economic & Environmental Factors

Multiple Years

```
'data.frame':   6293 obs. of  25 variables:
 $ iso2c          : chr  "AF" "AF" "AF" "AF" ...
 $ country        : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
 $ population     : num  37172386 36296400 35383128 34413603 33370794 ...
 $ year           : int  2018 2017 2016 2015 2014 2013 2012 2011 2010 2009 ...
 $ pop.density    : num  56.9 55.6 54.2 52.7 51.1 ...
 $ gdp.pc         : num  568 572 571 574 584 ...
 $ gdp.pc.income  : num  2242 2203 2129 2087 2069 ...
 $ co2            : num  NA NA 8672 9035 8467 ...
 $ co2.pc         : num  NA NA 0.245 0.263 0.254 ...
 $ birth.rate     : num  32.5 33.2 34 34.8 35.7 ...
 $ life.exp       : num  64.5 64.1 63.8 63.4 63 ...
 $ infant.mort.rate: num  48 49.6 51.3 53.2 55.2 57.2 59.5 61.7 64.1 66.5 ...
 $ unemployment   : num  11.1 11.2 11.3 11.4 11.4 ...
 $ under5.mort.rate: num  62.5 64.9 67.6 70.4 73.6 76.8 80.3 83.9 87.6 91.4 ...
 $ iso3c          : chr  "AFG" "AFG" "AFG" "AFG" ...
 $ region         : chr  "South Asia" "South Asia" "South Asia" "South Asia" ...
 $ capital        : chr  "Kabul" "Kabul" "Kabul" "Kabul" ...
 $ longitude      : chr  "69.1761" "69.1761" "69.1761" "69.1761" ...
 $ latitude       : chr  "34.5228" "34.5228" "34.5228" "34.5228" ...
 $ income         : chr  "Low income" "Low income" "Low income" "Low income" ...
 $ lending        : chr  "IDA" "IDA" "IDA" "IDA" ...
 $ gni.pc         : num  1746 1767 1766 1783 1796 ...
 $ hdi            : num  0.496 0.493 0.491 0.49 0.488 0.485 0.479 0.465 0.464 0.447 ...
 $ edu.index      : num  0.413 0.408 0.406 0.405 0.403 0.398 0.39 0.374 0.372 0.352 ...
 $ income.index   : num  0.432 0.434 0.434 0.435 0.436 0.438 0.435 0.421 0.426 0.409 ...
```

Figure 1.5: Social, Economic & Environmental Factors Across Years (Not Suitable)

While these are important factors in calculating HDI, this is not suitable for my project, as all of my factors are going to come from OpenStreetMaps, which only has geographical

data. However, some of this geographical data will somewhat represent these factors (for example, school density will be loosely proportional to the education index (`edu.index`)). The use of multiple years in this solution is also not suitable for my project, as I will only be accessing the most recent version of OpenStreetMaps. Any previous versions will just be more inaccurate, as new data is constantly being added.

This solution makes use of a Random Forest Classification Algorithm to determine the HDI Class of each country at a particular time, which can either be Low, Mid or High (Figure 1.6).

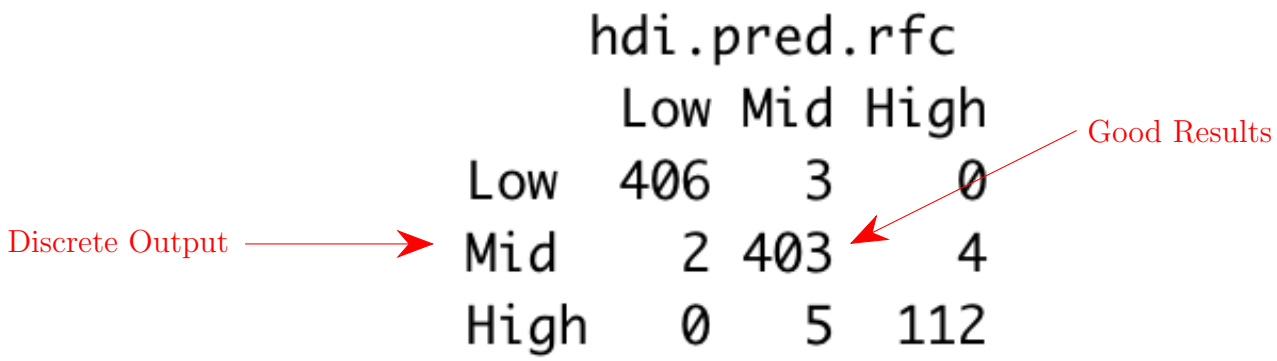


Figure 1.6: Random Forest Classification (Not Suitable)

While this does seem to produce good results (the HDI Class has been correctly predicted most of the time), this is not suitable for my solution, as I would like the output of my model to be continuous (the actual HDI), so I will instead be using a Multilayer Perceptron model (described later, in Section 2.3).

Human Development Reports' Country Insights [3]

This solution does not predict HDI, it only visualises it in different graphs. One of those is this ranking chart, which shows the HDI Rank of a particular country compared with the rest (Figure 1.7).

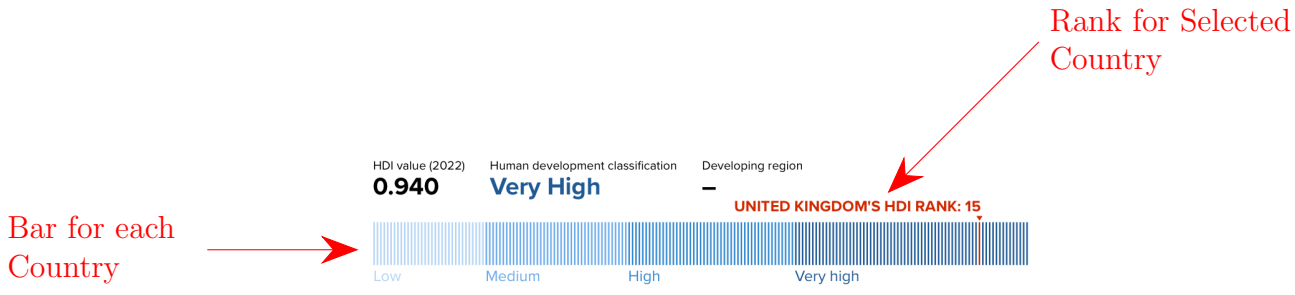


Figure 1.7: HDI Ranking Chart

You can also hover over this chart to reveal additional information for each of the other countries (Figure 1.8).

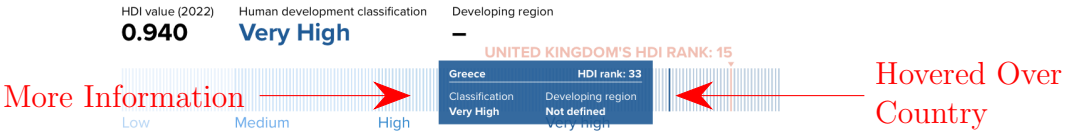


Figure 1.8: Hovering Over HDI Ranking Chart

I could implement a similar chart to this, except instead of a selected country, it would be whatever region the user is predicting the HDI of, so that they can compare it to the HDI of different countries.

This solution also contains other charts, such as this waterfall chart showing HDI of a particular country over time (Figure 1.9), however I believe none of these are suitable for my project, as I will only be dealing with present-day data.

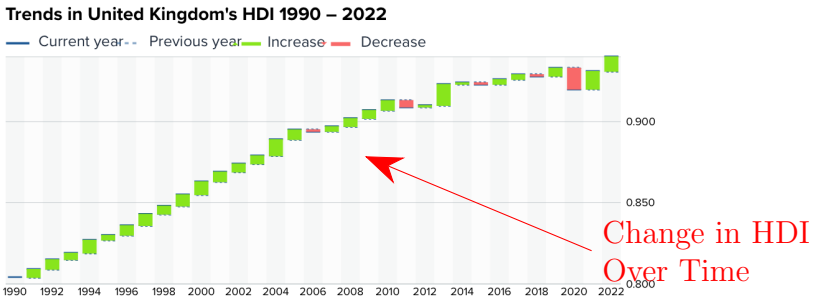


Figure 1.9: HDI Over Time Waterfall Chart (Not Suitable)

1.3.2 Stakeholder Interview & Feedback

For this interview, coloured quotes will represent each stakeholder, as shown in the Stakeholders Table (Table 1.1)

Question 1

1. *"Are you familiar with the HDI scale for development?"*

"Yes"
"Yes"
"Yes"
"Yes"

For each of the following questions, each stakeholder was given the options in a random order, and was made aware that they don't need to rank all of them.

Question 2

2. *"Rank the following end-features by how important you believe they are."*

Analysis of how much each factor affects the others
Comparison of predicted HDI to other countries
Ranking of suggested changes to increase development

"1. Ranking of suggested changes to increase development
2. Analysis of how much each factor affects the others
3. Comparison of predicted HDI to other countries"
"1. Analysis of how much each factor affects the others
2. Comparison of predicted HDI to other countries
3. Ranking of suggested changes to increase development"
"1. Ranking of suggested changes to increase development
2. Analysis of how much each factor affects the others
3. Comparison of predicted HDI to other countries"
"1. Ranking of suggested changes to increase development
2. Analysis of how much each factor affects the others
3. Comparison of predicted HDI to other countries"

To decide which feature is the most wanted from this data, each option will be awarded $(4 - p)$ points for each ranking, where p is the place that it was ranked (for example, if it was ranked in first place, it would gain 3 points).

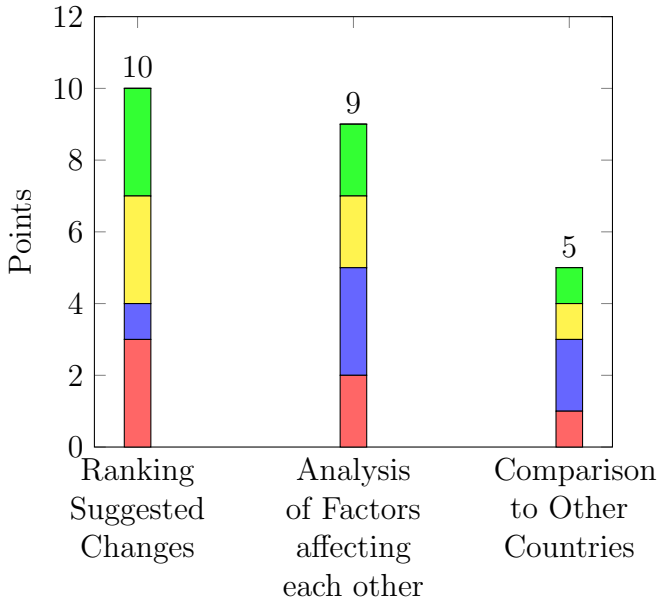


Figure 1.10: Score for each end-feature

Both the Ranking of suggested changes to increase development and the Analysis of how much each factor affects the others seem to be popular.

Notation Explanation

The following 2 questions are about various geographical factors which may have an impact on HDI.

As many of the following factors have a similar format, I will use the notation $D(x)$ to represent "x Density", for some object x (the number of x per unit area), and $A(x, y)$ to represent "Average Distance from x to nearest y", for some objects x, y (Note that $A(x, y)$ might not be the same as $A(y, x)$). The functions A and D will be implemented later, in Section 2.2.

There are many factors to consider which are directly linked to HDI, such as D(School) or D(Hospital), which I believe are essential factors to be considered. Therefore, I have decided to only ask about more ridiculous factors, such as D(Bench) or D(Post Box) which will have less of an effect on HDI. I have also included D(School) as a test to ensure that the stakeholders fully understand the meaning of HDI (it should always rank highly).

Question 3

3. *"Rank the following density factors affecting development in a particular area by how much you would like them to be implemented."*

D(Bench)	D(Fast-Food Place)	D(Lake)	D(Place of Worship)
D(Pond)	D(Post Box)	D(Restaurant)	D(Rock)
D(School)	D(Toilet)	D(Tree)	D(Vending Machine)

"1. D(Toilet), 2. D(School), 3. D(Place of Worship), 4. D(Restaurant)."
"1. D(School), 2. D(Toilet), 3. D(Restaurant), 4. D(Vending Machine), 5. D(Bench), 6. D(Post Box), 7. D(Place of Worship), 8. D(Fast-Food Place), 9. D(Pond), 10. D(Lake), 11. D(Rock), 12. D(Tree)."
"1. D(School), 2. D(Restaurant), 3. D(Post Box), 4. D(Tree), 5. D(Place of Worship), 6. D(Vending Machine), 7. D(Toilet), 8. D(Fast-Food Place)."
"1. D(School), 2. D(Bench), 3. D(Post Box), 4. D(Toilet), 5. D(Vending Machine), 6. D(Fast-Food Place), 8. D(Restaurant), 9. D(Place of Worship), 10. D(Tree)" (the abscense of an option at number 7 is not a mistake, he did not seem to put anything there.)

These rankings will be scored similarly, with each option being awarded $(13 - p)$ points, where p is the place that it was ranked, as there are now 12 options instead of 3. If the option was not ranked, then it won't be awarded any points. The points here are not comparable with the points from the previous question, as a different system is being used.

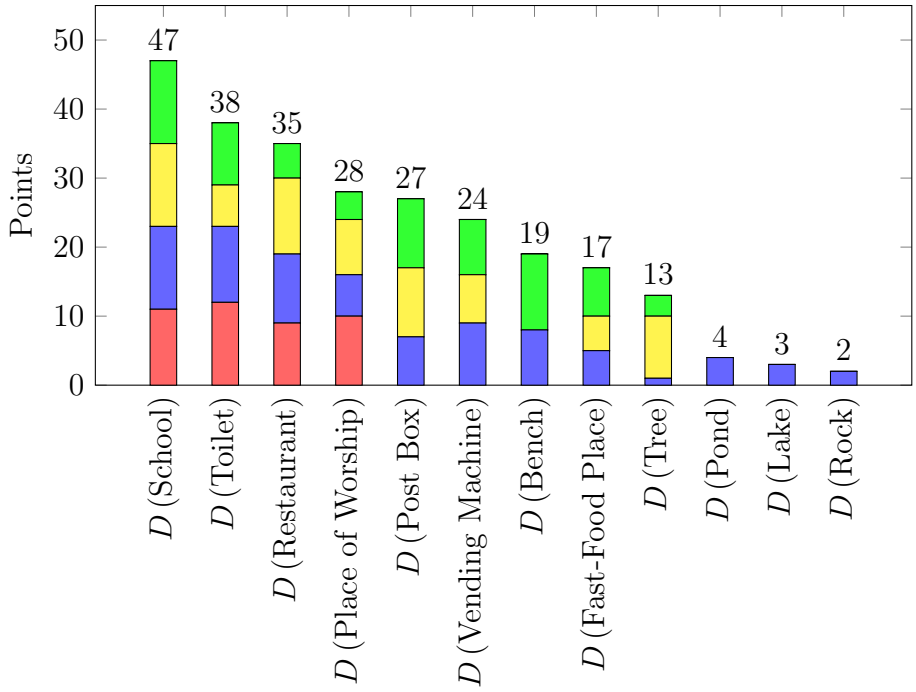


Figure 1.11: Score for each density factor

As Expected, the most popular answer was D (School). Among the other answers, D (Toilet), D (Restaurant), D (Place of Worship), D (Post Box) and D (Vending Machine) all proved to be very popular, with a score ≥ 20.

Question 4

4. *"Rank the following distance factors affecting development in a particular area by how much you would like them to be implemented."*

A (Bank, Slot Machine)	A (Bench, Tree)	A (Fast-Food Place, Toilet)
A (House, Fast-Food Place)	A (House, Place of Worship)	A (House, School)
	A (House, Tree)	

"1. A (House, School), 2. A (House, Place of Worship), 3. A (Fast-Food Place, Toilet)."

"1. A (House, School), 2. A (Bank, Slot Machine), 3. A (Fast-Food Place, Toilet), 4. A (House, Place of Worship), 5. A (House, Fast-Food Place), 6. A (Bench, Tree), 7. A (House, Tree)"

"1. A (Bank, Slot Machine), 2. A (House, School), 3. A (Bench, Tree), 4. A (House, Tree), 5. A (House, Place of Worship), 6. A (House, Fast-Food Place), 7. A (Fast-Food Place, Toilet)"

"1. A (House, School), 2. A (House, Tree), 3. A (House, Fast-Food Place), 4. A (Bench, Tree), 5. A (House, Place of Worship), 6. A (Fast-Food Place, Toilet), 7. A (Bank, Slot Machine)"

The scoring again will work the same as the previous 2 questions, except each option will be awarded $(8 - p)$ points, where p is the place that it was ranked, as there are now 7 options instead of 3 or 12.

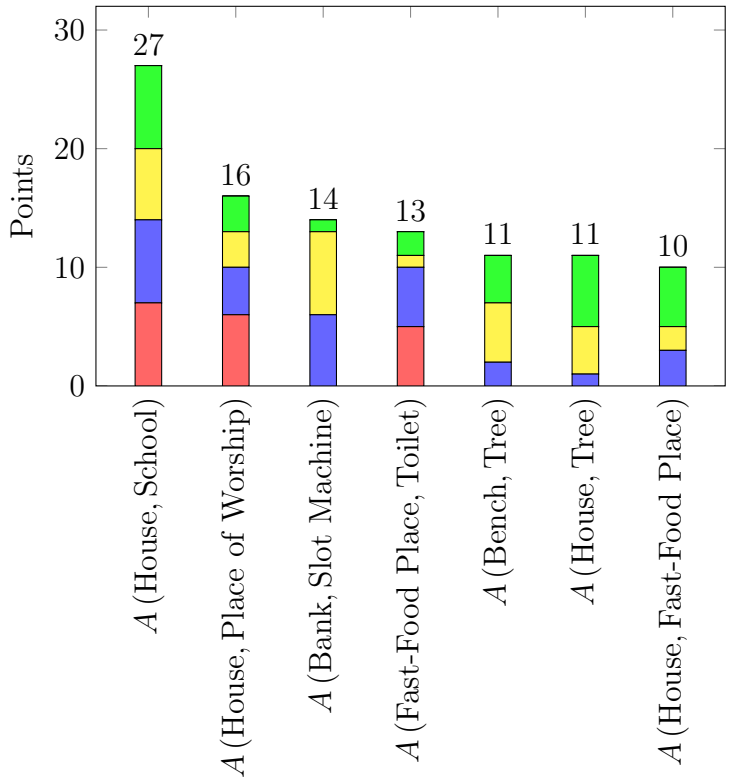


Figure 1.12: Score for each distance factor

Again, the serious factor $A\,(\text{House}, \text{School})$ was the most popular by far, and will be included among others such as $A\,(\text{House}, \text{Shop})$ or $A\,(\text{House}, \text{Pharmacy})$. Among the other answers, $A\,(\text{House}, \text{Place of Worship})$, $A\,(\text{Bank}, \text{Slot Machine})$ and $A\,(\text{Fast-Food Place}, \text{Toilet})$ proved to be very popular, with a score ≥ 12

I may ask further questions to my stakeholders as required, when I am developing my coded solution.

1.4 Essential Features

Feature	**Explanation**
Area Selection	The user must be able to select the area to predict the HDI of. They should be able to do this by either drawing on a map, or entering the data manually.
Multilayer Perceptron	The HDI must be predicted based on a range of scalar factors, which the user has just inputted.

Ranking of Suggested Changes to increase Development	This will allow for users to make informed decisions on what needs to be done to increase the development of their area. This feature proved popular in the Stakeholder Interview.
Analysis of how much each factor affects the others	Users can determine which factors may affect others, which will further increase the development. This feature was in many of the existing solutions and was also proved popular in the Stakeholder Interview.

Table 1.2: Essential Features

All of the above (except for the Multilayer Perceptron) will be implemented and accessed using a GUI.

1.5 Limitations

- **Only 1 Time Period** – As I am using OpenStreetMaps, I will only be able to access the data as it is now, so I won't be able to train the model on previous year's HDI. Older versions of OpenStreetMaps are only more inaccurate, so the training data will just be dirty and the final product will have a smaller chance of working.
- **Only Geographical Factors** – Factors such as life expectancy will not be accounted for, as I would like all the data to come from the same place (OpenStreetMaps), such that the user can circle an area on a map and the HDI will be estimated from that.

Figure 1.13: Map of all shelters from OpenStreetMaps

- **Skew Towards More Developed Countries** – OpenStreetMaps is not perfect. There will be a skew towards more developed countries because in those places, there are more people with computers who know about OpenStreetMaps, and are contributing to it. The map in Figure 1.13 shows this, as it suggests that there are many more shelters in Europe than in Africa, however in reality it is likely to be the other way around.

- **Potential Lack of Training Data** – HDI is only officially calculated for countries or sub-national regions (for example, in the United States, the sub-national regions are the 50 states). Therefore, I only have $\sim$ 2000 pieces of data to train on (which is equal to the total number of sub-national regions), which may not be enough to get accurate results.

1.6 Requirements

1.6.1 Hardware Requirements

- **Standard Input Devices (Keyboard & Mouse)** – I need to be able to write code and the user needs to be able to enter their details.
- **Monitor** – The user and I need to be able to see the output of the program.
- **Main Memory** – The program will be loaded into here when booted up.
- **Secondary Storage** – The pre-trained model must be stored in secondary storage so that it can be called upon at any time.
- (For me only) **Graphical Processing Unit** – While no complex graphics need to be processed, a GPU would decrease the time taken for the model to be trained, as it would involve lots of the same type of calculation (matrix multiplication) to be carried out simultaneously.

I will be using a Mac Pro, 2.66GHz Quad-Core Intel Xeon Processor to develop this app.

1.6.2 Software Requirements

- (For me only) **IDE (Visual Studio Code)** – I will write the code for this project (and the documentation) in this IDE, as it provides many useful features such as syntax highlighting, automatic bracket completion and version control (git integration).
- (For me only) **Python 3.10** – The programming language I intend to write in.
- (For me only) **Python Libraries** Including but not limited to:
  - **Gradio** – A GUI Library, that will initialise a GUI on a local server, so that the app can be accessed from a web browser.
  - **Numpy** – A Maths Library, which contains many useful mathematical functions and data structures, such as matrices or multidimensional arrays, as well as the operations that come with them.
  - **Folium** – A Mapping Library, which will allow me to create various maps within the Gradio interface, for both input and output.

- **Web Browser** – The Gradio GUI must be accessed through a web browser. I need to be able to test the code, and the user needs to be able to use it. The user will only require a web browser as I plan to deploy this app once I have finished development.

I will be using macOS Big Sur 11.7.9 to develop this app.

1.7 Success Criteria

No.	Success Criterion	Justification
S_1	Find suitable HDI training data for the Neural Network	The neural network must train on pre-calculated HDI values for certain areas, along with their respective density and average distance factors.
S_2	Retrieve OSM data for a given area using Overpass Turbo	This will allow me to calculate useful geographical factors to predict the HDI from instead of more human ones (which are harder to determine).
S_3	Calculate average distance factors from a given area	The average distance from *some object* to *some other object* is also a good measure to consider. For example $A\,(\text{House}, \text{School})$ is a useful factor as school children shouldn't have to travel long distances for education.
S_4	Calculate density factors from a given area	The number of *some object* per unit area is a good way of measuring how many of that object are in a particular area while removing the bias of larger areas.
S_5	Successfully Implement the Feedforward Algorithm	This algorithm will allow us to make predictions on the HDI given a set of factors.
S_6	Successfully Implement the Backpropogation Algorithm	This algorithm will allow us to train the neural network base on a set of training examples.
S_7	Successfully train the Neural Network	In order to accurately predict the HDI of a given area, the neural network must train on existing examples to see what makes a high HDI.
S_8	User can select an area on the map for which they want to analyse	This will allow the user to choose which area they want to predict the HDI of by drawing on the map.
S_9	Neural Network accurately predicts HDI from those factors	Once the user has selected their area, the HDI must be predicted based on factors calculated from that area.

S_{10}	Changes are suggested to increase the HDI	This will allow the user to make an informed decision on what to do in order to improve their area.
S_{11}	Accurately calculate where a new building would need to be in order to increase the HDI the most	The specific placement of new buildings (as suggestions to increase HDI) is important to consider, so a suitable place must be chosen in order to decrease the average distance to it as much as possible.
S_{12}	User can see how each factor can affect the others	This will allow the user to consider the secondary effects of their descisions, as the suggested changes may not only increase the HDI, but also some of the other factors.
S_{13}	Predicted HDI is ranked among other countries with a similar HDI	This will allow users who are not that familiar with HDI to get an idea of the HDI scale by comparing it to the gist of what they know.
S_{14}	App has a clean, simple and presentable GUI	This will make it easier for the user to use.
S_{15}	App has been deployed on Hugging Face	This will allow anyone with a web browswer to use this app, without having to install anything.

Table 1.3: Success Criteria

2. Design of the 1st Prototype

2.1 Structure Diagram

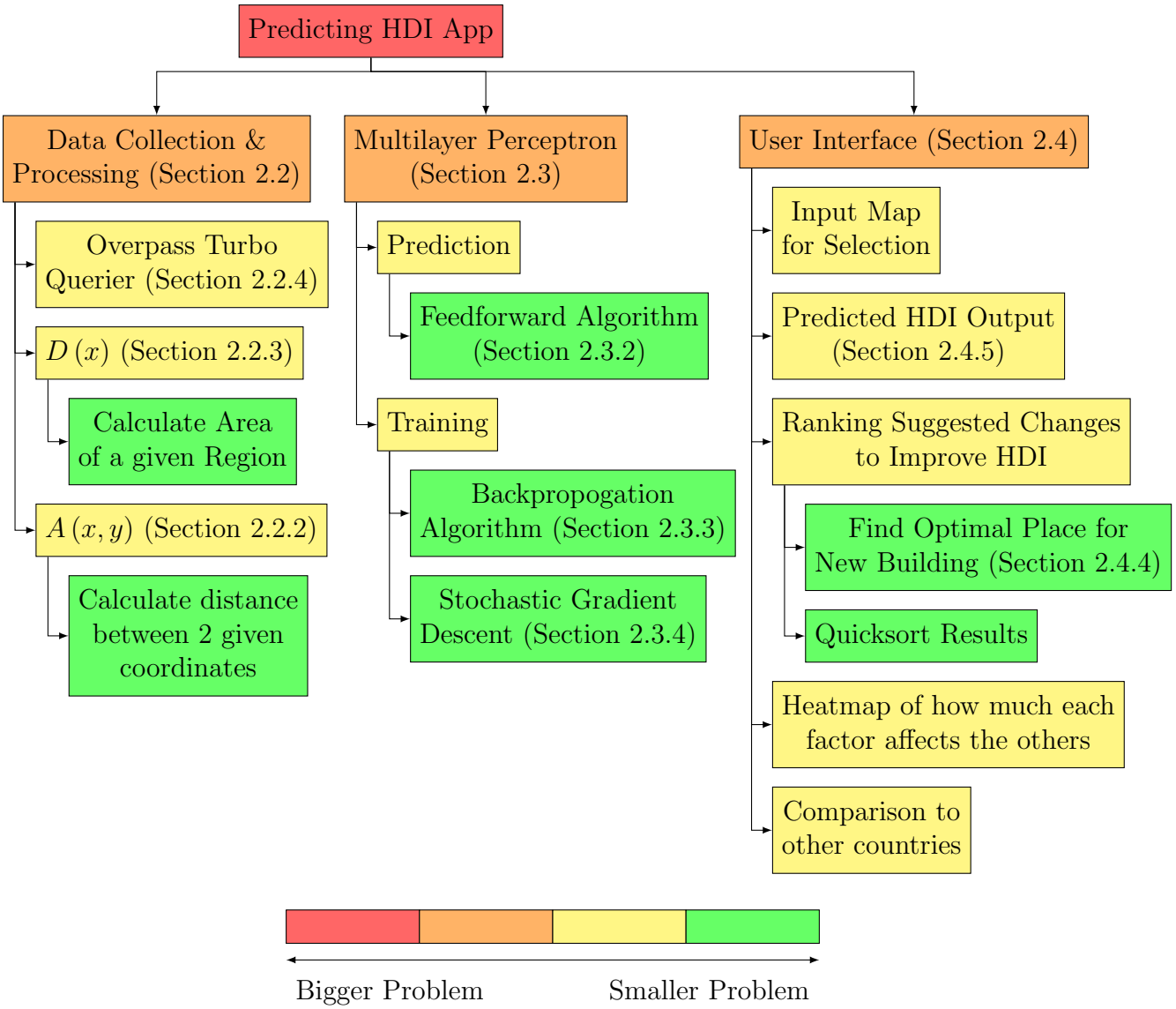


Figure 2.1: Structure Diagram

Figure 2.1 shows this project's structure, breaking down the larger problem of the entire app, into a series of smaller problems which are easier to consider. I will go into more detail for each of the smaller problems above (further decomposing them), in the approximate order I would be implementing them.

In this first prototype, I will be going through the algorithms involved in Data Collection & Processing (Section 2.2) and the Multilayer Perceptron (Section 2.3), with only the most essential features in the GUI. Various data visualisations such as the Factor Heatmap or Comparisons to Other Countries will be implemented in later prototypes.

2.2 Data Collection & Processing

2.2.1 Full List of Factors to Include

After considering the responses from the stakeholder interview (Section 1.3.2), as well as research on what contributes to a higher HDI, the full list of factors I would like to consider are (in no particular order):

1. $A\,(\text{House}, \text{School})$
2. $A\,(\text{House}, \text{Hospital})$
3. $A\,(\text{House}, \text{Pharmacy})$
4. $A\,(\text{House}, \text{Restaurant})$
5. $A\,(\text{School}, \text{Hospital})$
6. $A\,(\text{Police}, \text{Hospital})$
7. $A\,(\text{House}, \text{Place of Worship})$
8. $A\,(\text{Bank}, \text{Slot Machine})$
9. $A\,(\text{Fast-Food Place}, \text{Toilet})$
10. $A\,(\text{House}, \text{Police})$
11. $A\,(\text{University}, \text{Library})$
12. $A\,(\text{House}, \text{Library})$
13. $D\,(\text{School})$
14. $D\,(\text{Hospital})$
15. $D\,(\text{Pharmacy})$
16. $D\,(\text{Police})$
17. $D\,(\text{Library})$
18. $D\,(\text{Toilet})$
19. $D\,(\text{Restaurant})$
20. $D\,(\text{Place of Worship})$
21. $D\,(\text{Post Box})$
22. $D\,(\text{Vending Machine})$
23. $D\,(\text{Bench})$
24. $D\,(\text{Tree})$

Now we must figure out how to calculate $A\,(x, y)$, for some objects x, y, and $D\,(x)$, for some object x.

2.2.2 Finding the Average Distance between 2 objects x, y, $A(x, y)$

Finding the distance between 2 given points

This is a difficult problem to solve, so instead decompose it and consider a simple one, finding the distance between any 2 given points on the Earth's surface (latitude longitude pairs).

First, abstract the earth into a perfectly smooth sphere (which is fairly accurate anyway as $\frac{\text{height difference between Mt. Everest and the Mariana Trench}}{\text{Earth's average radius}} = 0.0031$), with centre O and radius r, and rotated such that the plane of the equator lies on the xy-plane and that the point with 0 latitude and 0 longitude lies on the positive x-axis. I am not considering the altitude at each point on the Earth's surface, nor the slight eccentricity in the shape of the Earth.

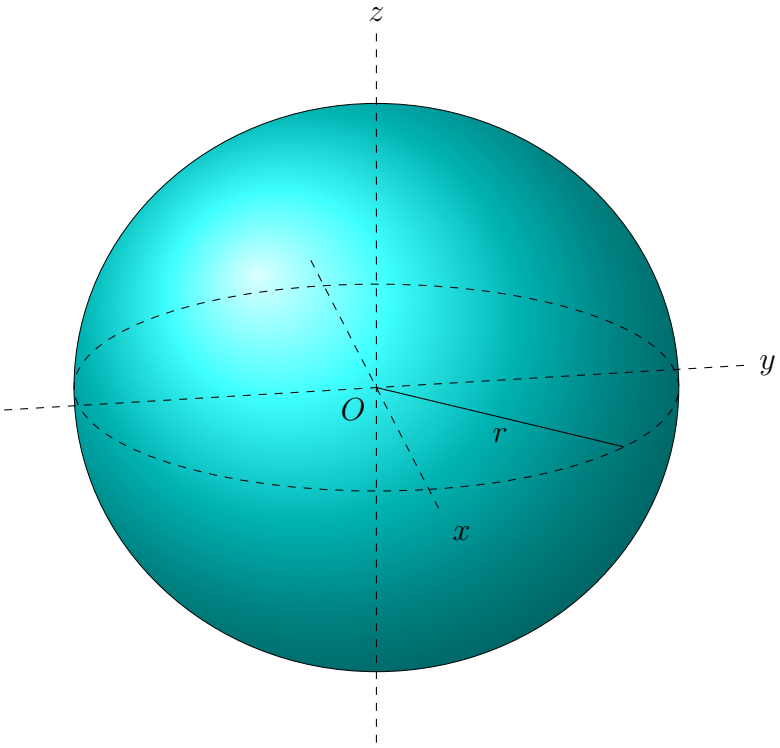


Figure 2.2: Abstracted Model of the Earth

The 2 points on its surface we are trying to find the distance between are P_1 and P_2, each with a latitude and longitude (La_1, La_2, Lo_1, Lo_2), given in radians.

Construct points A and B, such that they lie directly below P_1 and P_2 respectively, and sit on the plane of the equator. By definition of latitude, $\angle P_1OA = La_1$ and $\angle P_2OB = La_2$. From right-angled trigonometry, $OA = r\cos(La_1)$ and $OB = r\cos(La_2)$

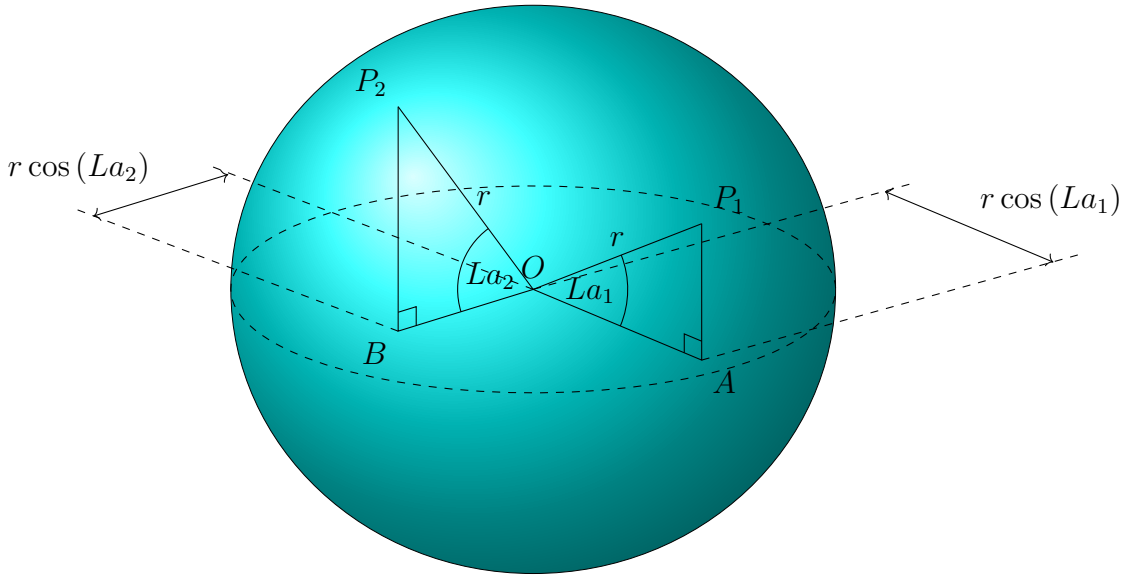


Figure 2.3: P_1 and P_2

Construct the line AB, which lies on the plane of the equator. By definition of longitude, $\angle AOB = |Lo_1 - Lo_2|$ (the difference in longitudes). Define a to be the length of the line segment AB, which can be found using the cosine rule.

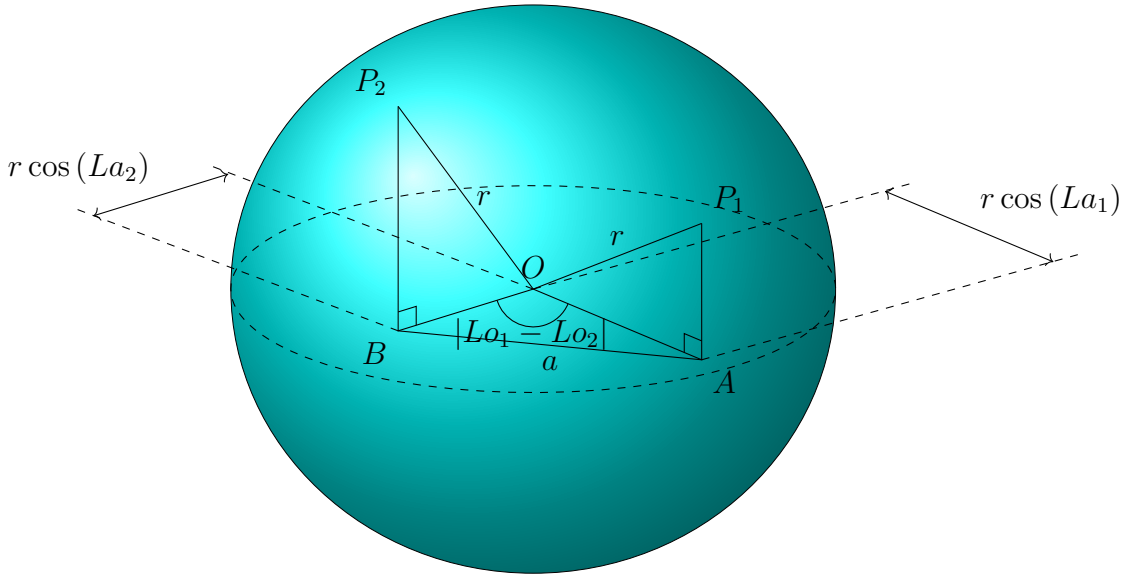


Figure 2.4: Line segment AB

$$
\begin{aligned}
a = AB &= \sqrt{OA^2 + OB^2 - 2 \times OA \times OB \times \cos(\angle AOB)} \\
&= \sqrt{(r\cos(La_1))^2 + (r\cos(La_2))^2 + 2 \times r\cos(La_1) \times r\cos(La_2) \times \cos(|Lo_1 - Lo_2|)} \\
&= \sqrt{r^2\cos^2(La_1) + r^2\cos^2(La_2) - 2r^2\cos(La_1)\cos(La_2)\cos(Lo_1 - Lo_2)}
\end{aligned}
\tag{2.1}
$$

From right-angled trigonometry, $P_1A = r\sin(La_1)$ and $P_2B = r\sin(La_2)$. Construct the line P_1P_2, which will lie in the same vertical plane as AB (as P_1 is directly above A and P_2 is directly above B). If P_1 and P_2 are in the same hemisphere (north or south), then the shape P_1ABP_2 will be a right-angled trapezium. Define b to be the length of the line segment P_1P_2.

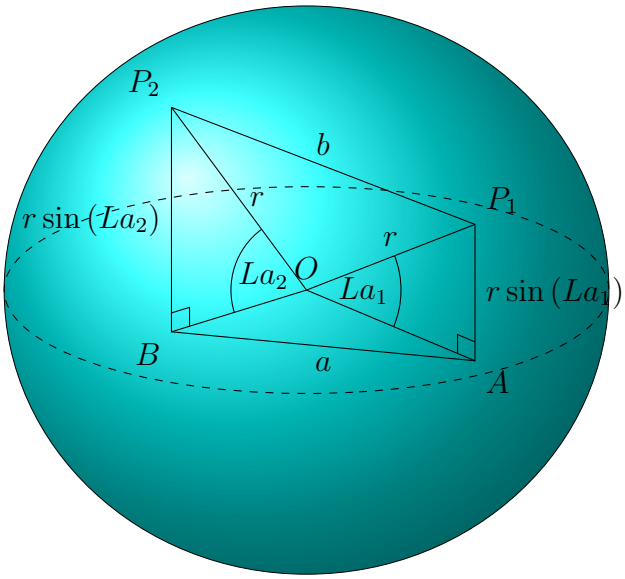


Figure 2.5: Line segment P_1P_2

b can be found using Pythagoras, by considering moving AB up until it meets either P_1 or P_2. We don't know which one it will meet first, so the difference in heights is $|P_1A - P_2B|$. This will still be true even if P_1ABP_2 is not a trapezium (as a result of P_1 and P_2 being in opposite hemispheres), as the one in the Southern Hemisphere will have a negative height.

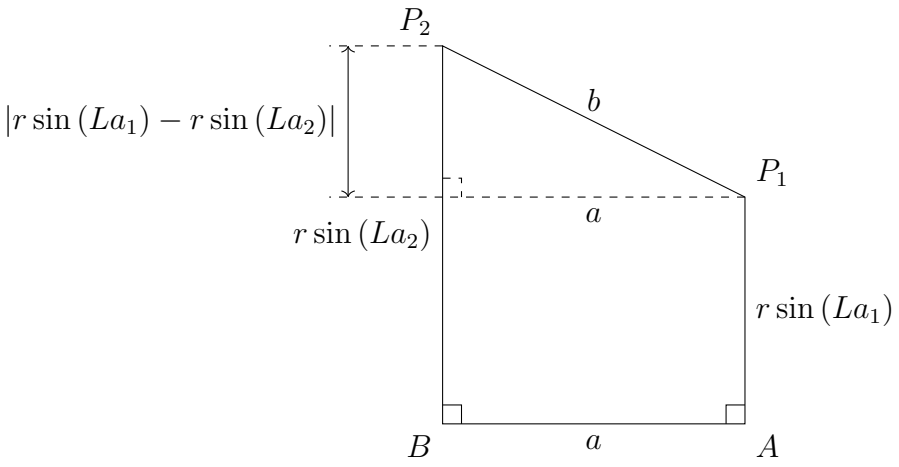


Figure 2.6: Finding b

$$\begin{aligned} b = P_1P_2 &= \sqrt{|P_1A - P_2B|^2 + AB^2} \\ &= \sqrt{(|r\sin(La_1) - r\sin(La_2)|)^2 + a^2} \\ &= \sqrt{(r\sin(La_1) - r\sin(La_2))^2 + a^2} \end{aligned} \tag{2.2}$$

This is not quite what we are after, as this is the straight line distance from P_1 to P_2, but we want the distance along the surface of the Earth, which will be curved. To find this, we must find the angle θ between P_1 and P_2, which we can do by using the cosine rule in reverse.

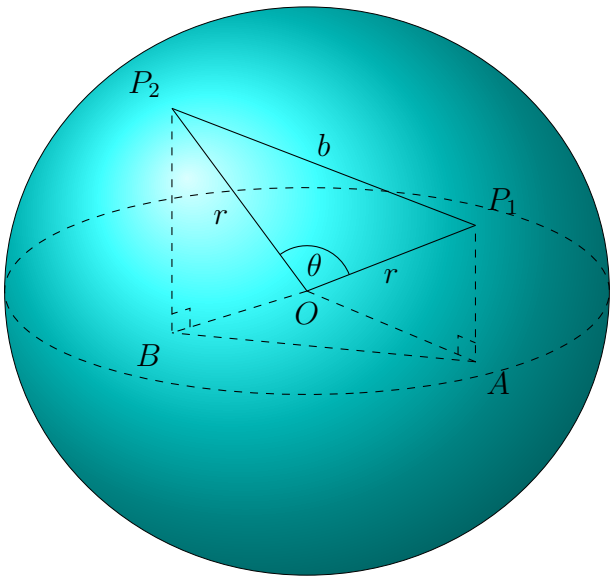


Figure 2.7: Angle θ between P_1 and P_2

$$\begin{aligned}\theta = \angle P_1OP_2 &= \arccos\left(\frac{(OP_1)^2 + (OP_2)^2 - (P_1P_2)^2}{2 \times OP_1 \times OP_2}\right) \\ &= \arccos\left(\frac{r^2 + r^2 - b^2}{2 \times r \times r}\right) \\ &= \arccos\left(\frac{2r^2 - b^2}{2r^2}\right) \\ &= \arccos\left(1 - \frac{b^2}{2r^2}\right)\end{aligned} \tag{2.3}$$

The arclength between 2 points on a circle (in this case, the great circle on the surface of the Earth containing P_1 and P_2) separated by an angle of θ is given by arclength $= r\theta$.

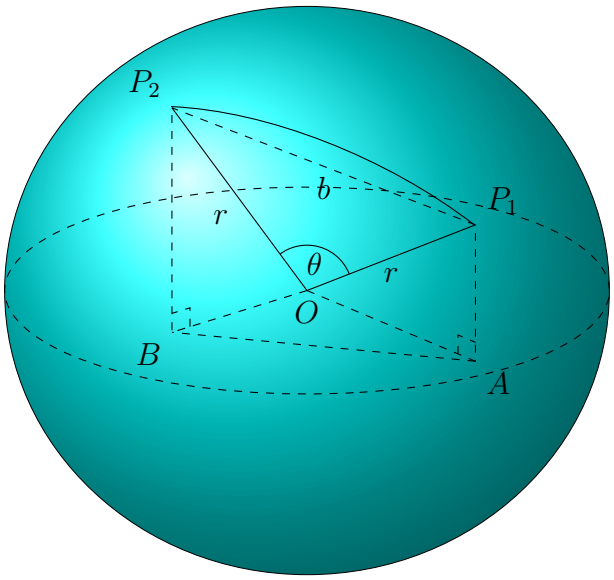


Figure 2.8: Arc between P_1 and P_2

Substituting θ for equation 2.3 gives

$$\text{arclength} = r \arccos\left(1 - \frac{b^2}{2r^2}\right) \tag{2.4}$$

Substituting b for equation 2.2 gives

$$\begin{aligned}\text{arclength} &= r \arccos\left(1 - \frac{\left(r\sin\left(La_1\right) - r\sin\left(La_2\right)\right)^2 + a^2}{2r^2}\right) \\ &= r \arccos\left(1 - \frac{r^2\sin^2\left(La_1\right) - 2r^2\sin\left(La_1\right)\sin\left(La_2\right) + r^2\sin^2\left(La_2\right) + a^2}{2r^2}\right)\end{aligned} \tag{2.5}$$

Substituting a for equation 2.1 gives

$$
\begin{aligned}
\text{arclength} &= r \arccos\left(1 - \frac{r^2\begin{pmatrix}\sin^2(La_1) - 2\sin(La_1)\sin(La_2) + \sin^2(La_2) + \cos^2(La_1) \\ + \cos^2(La_2) - 2\cos(La_1)\cos(La_2)\cos(Lo_1 - Lo_2)\end{pmatrix}}{2r^2}\right) \\
&= r \arccos\left(1 - \frac{\begin{matrix}\sin^2(La_1) + \cos^2(La_1) + \sin^2(La_2) + \cos^2(La_2) \\ - 2\sin(La_1)\sin(La_2) - 2\cos(La_1)\cos(La_2)\cos(Lo_1 - Lo_2)\end{matrix}}{2}\right) \\
&= r \arccos\left(1 - \frac{2 - 2\sin(La_1)\sin(La_2) - 2\cos(La_1)\cos(La_2)\cos(Lo_1 - Lo_2)}{2}\right) \\
&= r \arccos\left(1 - (1 - \sin(La_1)\sin(La_2) - \cos(La_1)\cos(La_2)\cos(Lo_1 - Lo_2))\right) \\
&= r \arccos\left(\sin(La_1)\sin(La_2) + \cos(La_1)\cos(La_2)\cos(Lo_1 - Lo_2)\right)
\end{aligned}
\tag{2.6}
$$

Therefore, the distance in km between 2 latitude and longitude coordinates along the surface of the Earth is given by:

$$
\text{distance} = r \arccos\left(\sin(La_1)\sin(La_2) + \cos(La_1)\cos(La_2)\cos(Lo_1 - Lo_2)\right) \tag{2.7}
$$

where r is the radius of the Earth in km and La_1, La_2, Lo_1 and Lo_2 are the 2 points' latitudes and longitudes in radians.

However, the latitude and longitude coordinates returned by OpenStreetMaps are not measured in radians, but instead measured in degrees, so we must use the conversion factor (angle in radians) $= \frac{\pi}{180} \times$ (angle in degrees) on each one before they are passed into the formula.

Finding the Average Distance between 2 objects x, y, $A(x, y)$

Now that we have the means to find the distance between any 2 points along the Earth's surface, we can consider the average distance between 2 particular types of object. Recall that $A(x, y)$ is specifically defined as "Average Distance from x to nearest y" (for example, x might be House, and y might be School), so the input for this function will be 2 sets of lists of latitude and longitude coordinates, corresponding to all the x-objects and y-objects in a particular area.

As there are likely to be hundreds of thousands of each object (depending on how large of an area the user selects), it will take too long to calculate the average distance over all of these data points. This is important to consider, as this function will be run thousands of times, both in preparing the data (for me only) and when users are using the app.

Therefore, instead of considering every single data point, randomly sample an arbitrary 500 of the x-objects (and if there are less than 500, consider all of them), and then find out which of the y-objects is closest to each of them. We must consider every y-object for each considered x-object as we don't know which one will be closest to that x. Considering all the y-objects should be fine, as for all the factors in the form $A(x, y)$, there will be many more x-objects than y-objects (for example, there are many more houses than schools).

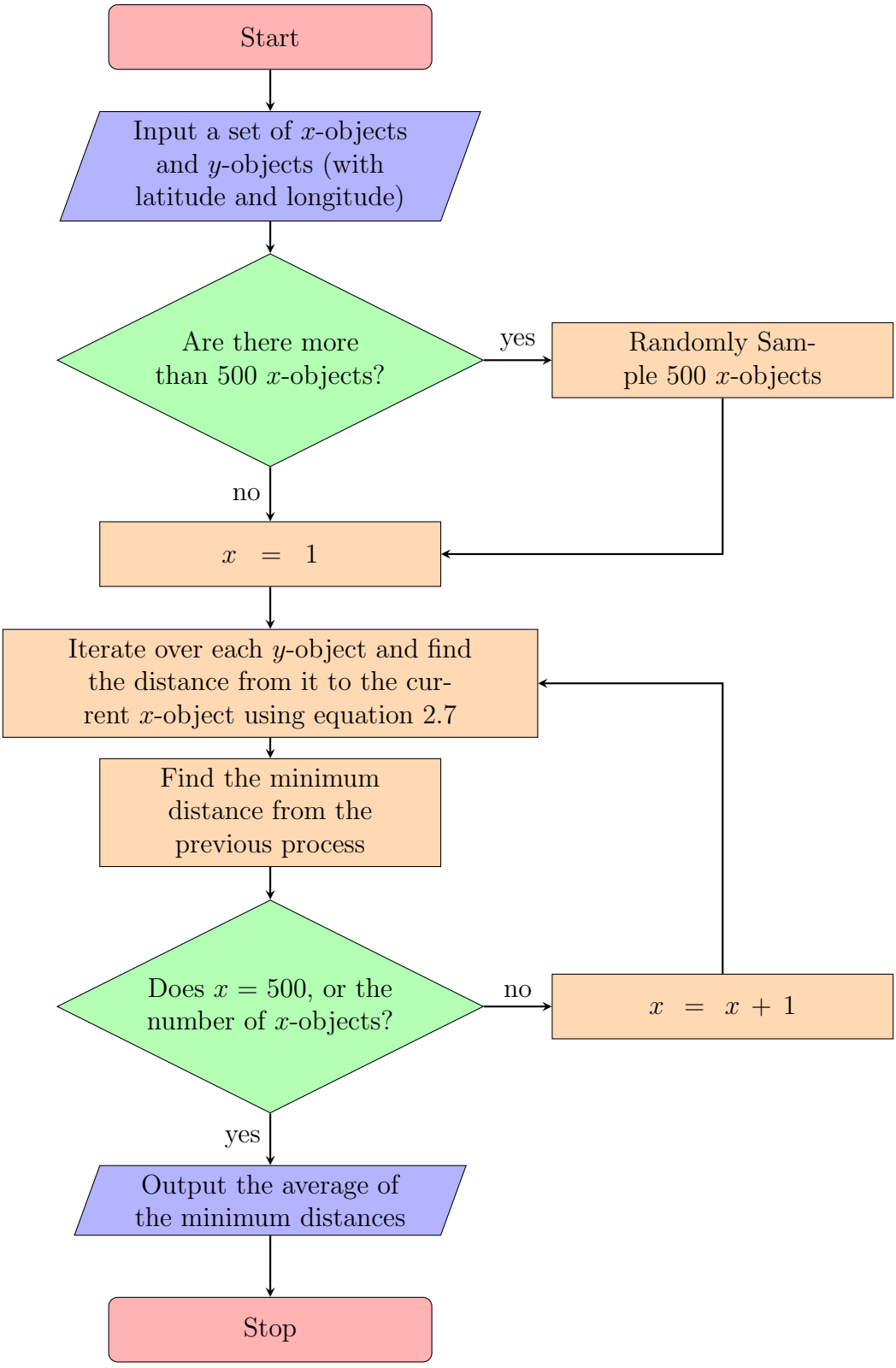


Figure 2.9: $A(x, y)$ Flowchart

For validation, if the entered list of x-objects is empty (in other words, there does not

exist any x-objects in the region), the average distance returned should be infinity.

2.2.3 Finding the Density of some object x, $D(x)$

Definition of $D(x)$

The density of some object is defined as "The number of that object per unit area". Therefore, $D(x)$ is

$$\frac{\text{Total number of } x\text{-objects in a given region}}{\text{Area of that region}} \tag{2.8}$$

Finding the total number of x-objects in a region is easy. As they will come in a list, we just need to find the length of that list. However, finding the area of that region is much harder, as the only information we have about it is a list (of unknown size) of its bounding latitude and longitude coordinates.

To do this, I will be using the same abstraction for the Earth, as shown in Figure 2.2.

Area of a Polygon on the Surface of a Sphere

First, assume that any polygon on the surface of a sphere (any region that the user may draw) can be cut into a series of internal spherical triangles (in fact, a spherical polygon with n sides can always be broken up into $n-2$ spherical triangles).

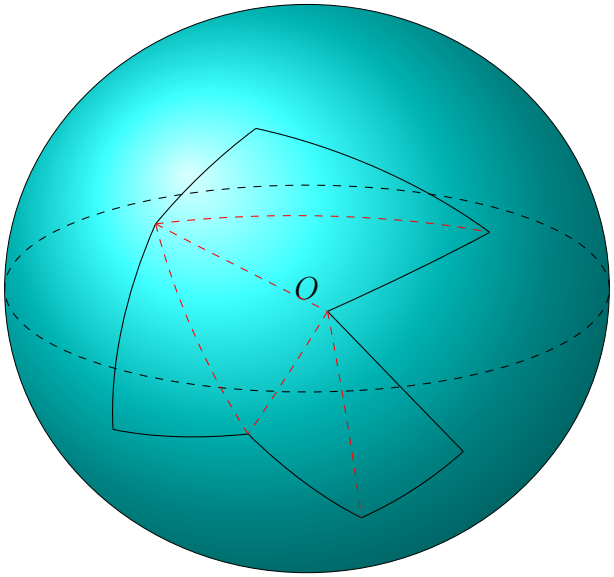


Figure 2.10: Cutting into Triangles

To find the area of the polygon, we can individually find the area of each triangle and then add them up.

To find the area of a triangle on the surface of the sphere, consider extending the sides of the triangle to make great circles around the sphere.

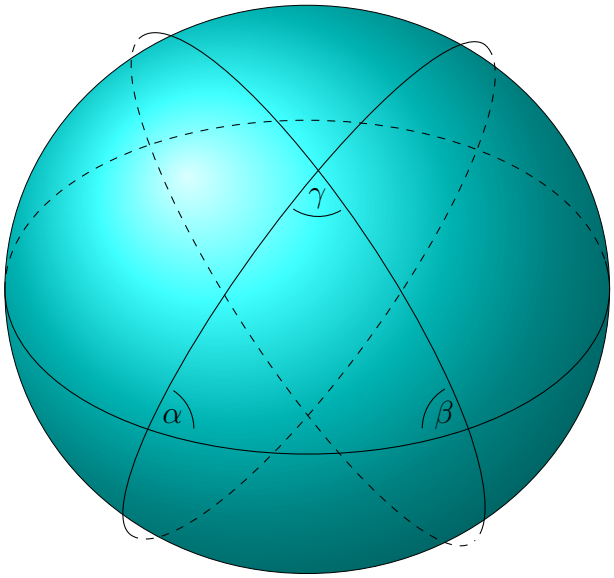


Figure 2.11: Triangle Edges Extended

As you can see from Figure 2.11, the total surface area of the sphere can be given by the sum of these 6 "spherical lune" shapes, with an excess of 4 of the triangles we want (at either end of the sphere, the triangle is formed by the intersection of 3 spherical lunes, and so therefore we must subtract 2 for each side to get the full area of the sphere with no excess, which is why we must subtract 4 triangles overall).

$$A_{\text{Sphere}} = \sum A_{\text{Lune}} - 4A_{\text{Triangle}} \tag{2.9}$$

The surface area of a sphere is $4\pi r^2$, and the area of a spherical lune of an angle θ can be given as a fraction of the full area of the sphere.

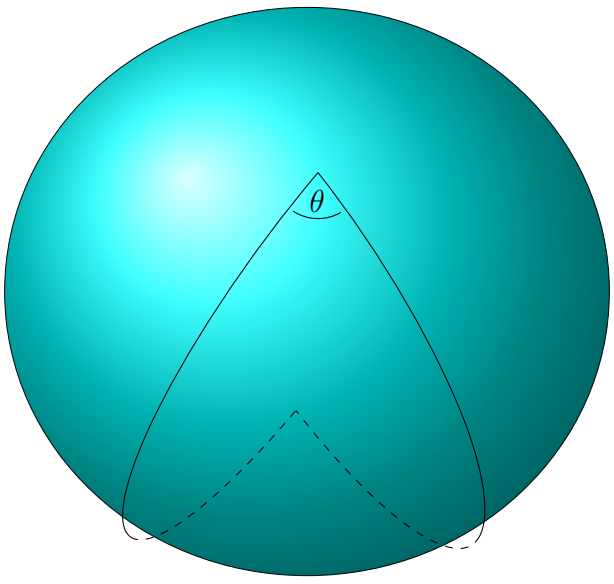


Figure 2.12: Area of a Spherical Lune

$$\begin{aligned} A_{\text{Lune}} &= \frac{\theta}{2\pi} \times 4\pi r^2 \\ &= 2\theta r^2 \end{aligned} \tag{2.10}$$

The 6 lunes shown in Figure 2.11 can be broken down into 3 sets of 2 identical lunes, with angles α, β and γ. Therefore, from equation 2.9

$$\begin{aligned} A_{\text{Triangle}} &= \frac{\sum A_{\text{Lune}} - A_{\text{Sphere}}}{4} \\ &= \frac{2 \times (2\alpha r^2 + 2\beta r^2 + 2\gamma r^2) - 4\pi r^2}{4} \\ &= \frac{4r^2 (\alpha + \beta + \gamma - \pi)}{4} \\ &= r^2 \left(\sum \theta - \pi \right) \end{aligned} \tag{2.11}$$

where $\sum \theta$ represents the sum of the angles in the spherical triangle.

As our original region (an n-sided spherical polygon) was split up into $n-2$ spherical triangles, we can see that the area of the region R is given by

$$\begin{aligned} R &= \sum_{i=1}^{n-2} \left(A_{\text{Triangle}}\right)_i \\ &= \sum_{i=1}^{n-2} r^2 \left(\left(\sum \theta \right)_i - \pi \right) \\ &= r^2 \left(\sum_{i=1}^{n-2} \left(\sum \theta \right)_i - (n-2)\,\pi \right) \end{aligned} \tag{2.12}$$

where $\left(\sum \theta \right)_i$ represents the sum of the angles in spherical triangle i.

As the sum of the angles in a polygon equals the sum of all the sums of angles in the triangles it was broken up into, this simplifies to

$$R = r^2 \left(\sum \theta - (n-2)\,\pi \right) \tag{2.13}$$

where $\sum \theta$ represents the sum of the interior angles in the spherical polygon, and n is the number of sides of the spherical polygon.

Finding the Interior Angles

To find the interior angles of our polygon, we could use the cosine rule in reverse, as before. However, as $\cos\theta \equiv \cos(360 - \theta)$, this will essentially result in both the interior angle and exterior angle for all the vertices, with no indication of which ones which. Therefore, we must introduce some directionality, by considering vectors.

The vectors we want to consider are those in the plane tangent to the point in which we want to find the interior angle of.

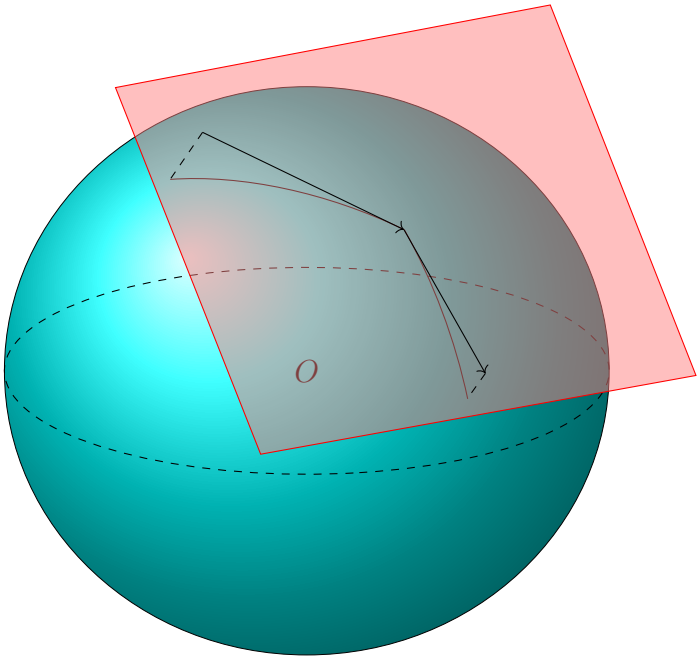


Figure 2.13: Considering Vectors

To find these vectors, we first must convert each of the coordinates from their latitude-longitude forms into their 3D position vector forms (xyz coordinates).

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\cos(La)\cos(Lo) \\ r\cos(La)\sin(Lo) \\ r\sin(La) \end{pmatrix} \tag{2.14}$$

where La is the latitude of the point, Lo is the longitude of the point, and r is the radius of the Earth. As we are only using this to find angles, the radius of the Earth is actually irrelevant, and so to simplify things, we can let $r = 1$.

Finding these vectors on a plane which is rotated strangely is difficult, so we can instead rotate the whole system such that this plane lies on the xy-plane. This can be done by rotating it about the z-axis anticlockwise by an angle of $-Lo$, by using the matrix

$$\begin{pmatrix} \cos(-Lo) & -\sin(-Lo) & 0 \\ \sin(-Lo) & \cos(-Lo) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.15}$$

followed by rotating it about the y-axis anticlockwise by an angle of $La - 90°$, by using the matrix

$$\begin{pmatrix} \cos(La-90°) & 0 & \sin(La-90°) \\ 0 & 1 & 0 \\ -\sin(La-90°) & 0 & \cos(La-90°) \end{pmatrix} \tag{2.16}$$

finally followed by translating it down by 1 unit (as, for now, $r = 1$), by adding the vector

$$\begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \tag{2.17}$$

This will have the effect of moving the point we want to find the angle at, to $(0, 0, 0)$, while transforming the other points in the same way.

As the plane with the vectors we want to find is now the xy-plane, all we need to do to project the other points onto that plane is set their z-coordinates to 0.

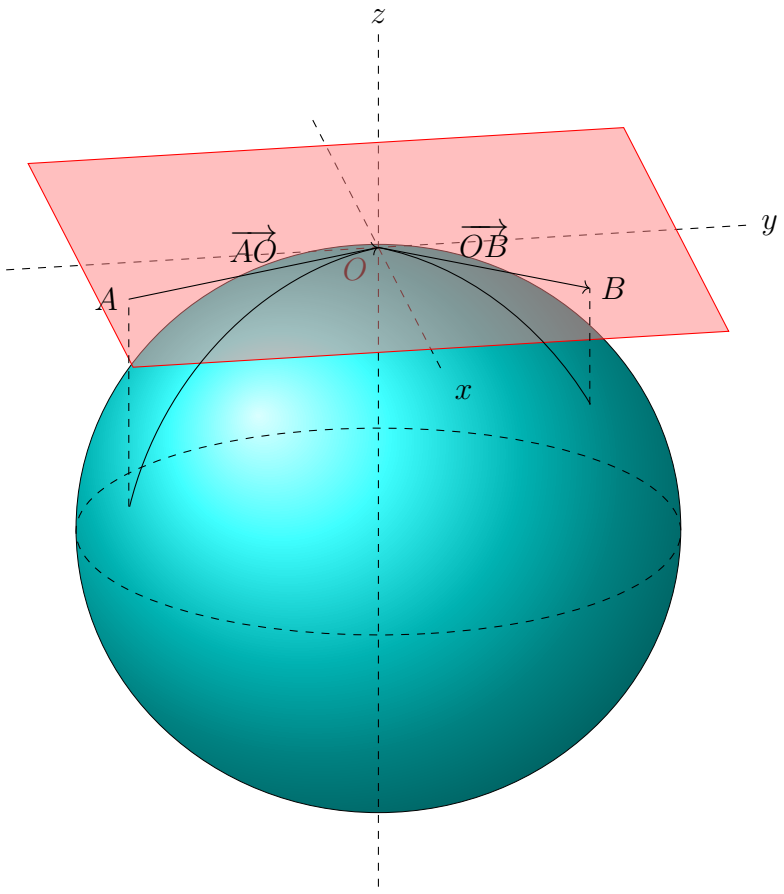


Figure 2.14: Projecting the Other Points onto the xy-plane

As the points will be given in a list, in the other the user has drawn them in the polygon, $\overrightarrow{AO}$ (the vector from the projection of the previous point to the current point) will be given by the coordinates of A times -1, and $\overrightarrow{OB}$ (the vector from the current point to the projection of the next point) will be given by the coordinates of B.

Imagine travelling along the line segment AO. When you get to O, you must turn some angle θ anticlockwise (between 180° and −180°) to be facing in the right direction to travel along OB. Note that this angle is not the angle between AO and OB, but the angle between $\overrightarrow{AO}$ and $\overrightarrow{OB}$. The *anticlockwise* angle θ between $\overrightarrow{AO}$ and $\overrightarrow{OB}$ is given by

$$\theta = \arctan 2\left((a_2b_3 - a_3b_2) - (a_3b_1 - a_1b_3) + (a_1b_2 - a_2b_1)\,, a_1b_1 + a_2b_2 + a_3b_3\right) \tag{2.18}$$

where

$$\overrightarrow{AO} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}, \overrightarrow{OB} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{2.19}$$

Therefore, if the user has drawn the region in an anticlockwise-winding order, then the interior angle of this particular point will be given by $180° - \theta$.

However, the user may decide to draw their region in a clockwise-winding order, which would currently result in the wrong area being calculated. To detect if the user has drawn their shape in a clockwise or anticlockwise winding order, we can find the sum of the "anticlockwise" angles. If it is positive, they have drawn it anticlockwise, and if it is negative, they must have drawn it clockwise. If it is 0, that means that the shape must self-intersect at some point, which should not be allowed.

If the shape was drawn clockwise, then instead of calculating the anticlockwise angles, this process will have calculated the clockwise angles between the vectors (as the vectors are pointing the other way), which is why I put "anticlockwise" in quotation marks. In order to turn them into the correct anticlockwise angles, we must multiply each of them by -1, before calculating the interior angles.

Final Algorithm

Now that we have all the interior angles, all we need to do is plug them into the formula for the area of the region (equation 2.13), and then divide by the number for x-objects to find the density.

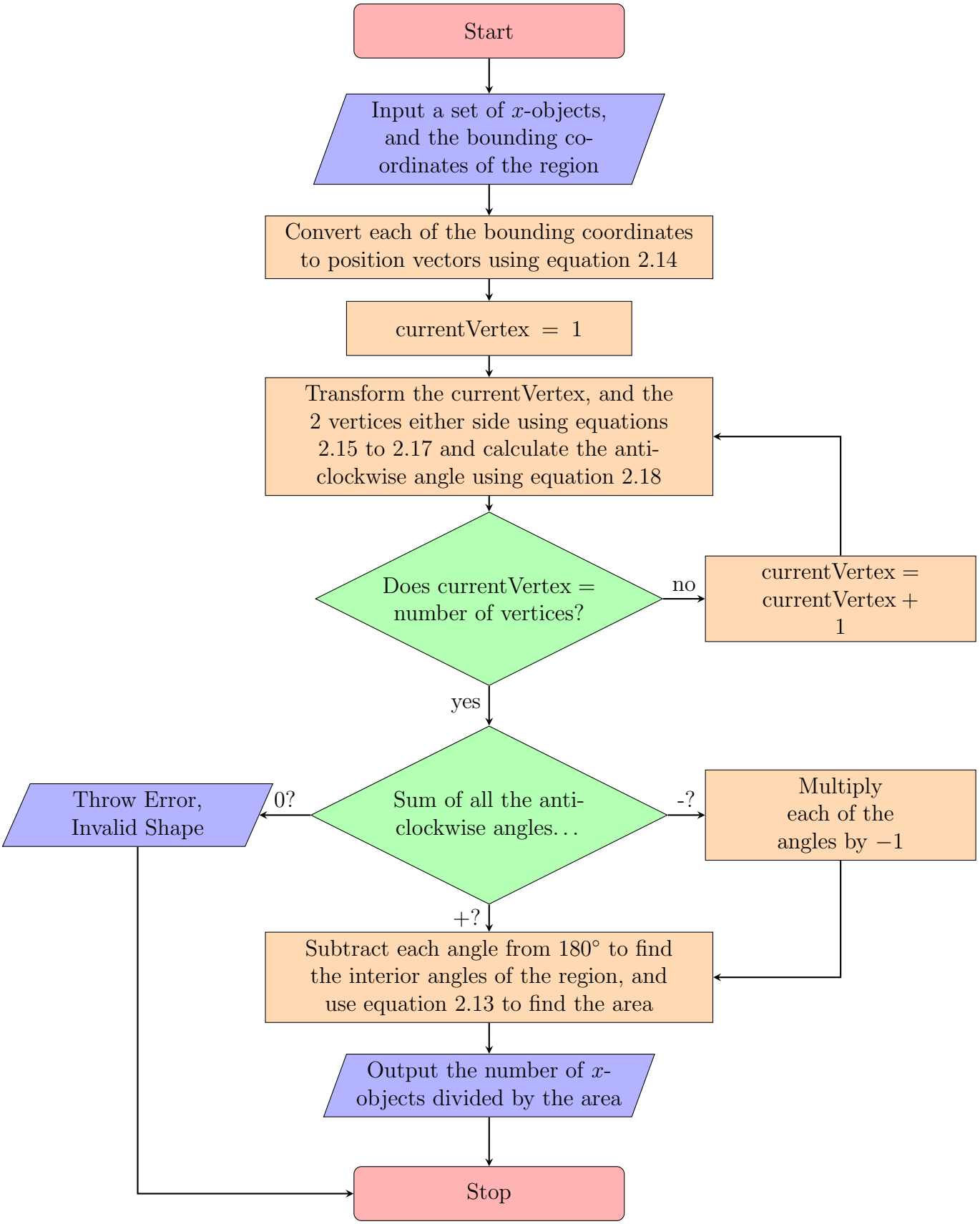


Figure 2.15: $D(x)$ Flowchart

2.2.4 Overall Algorithm for Data Collection & Processing

In order to train a neural network, I need both input and output data. The input data will be 24 factors listed above, in section 2.2.1, and the output data will be the true HDI of the region.

For this, I will be using the subnational region data from the Global Data Lab [4], in order to maximise the number of training examples. In order to calculate the 24 factors for each of these regions, I will need the bounding coordinates of all the regions. I will also get this data from the Global Data Lab [5], which contains shapefile data on all of the subnational regions.

Therefore, all we need to do is for each subnational region, calculate each factor and store it in an external file (e.g a `.csv` file) along with the correct HDI, ready for the neural network to train on.

In order to calculate each factor however, we must first query Overpass Turbo, in order to get the data about the locations objects.

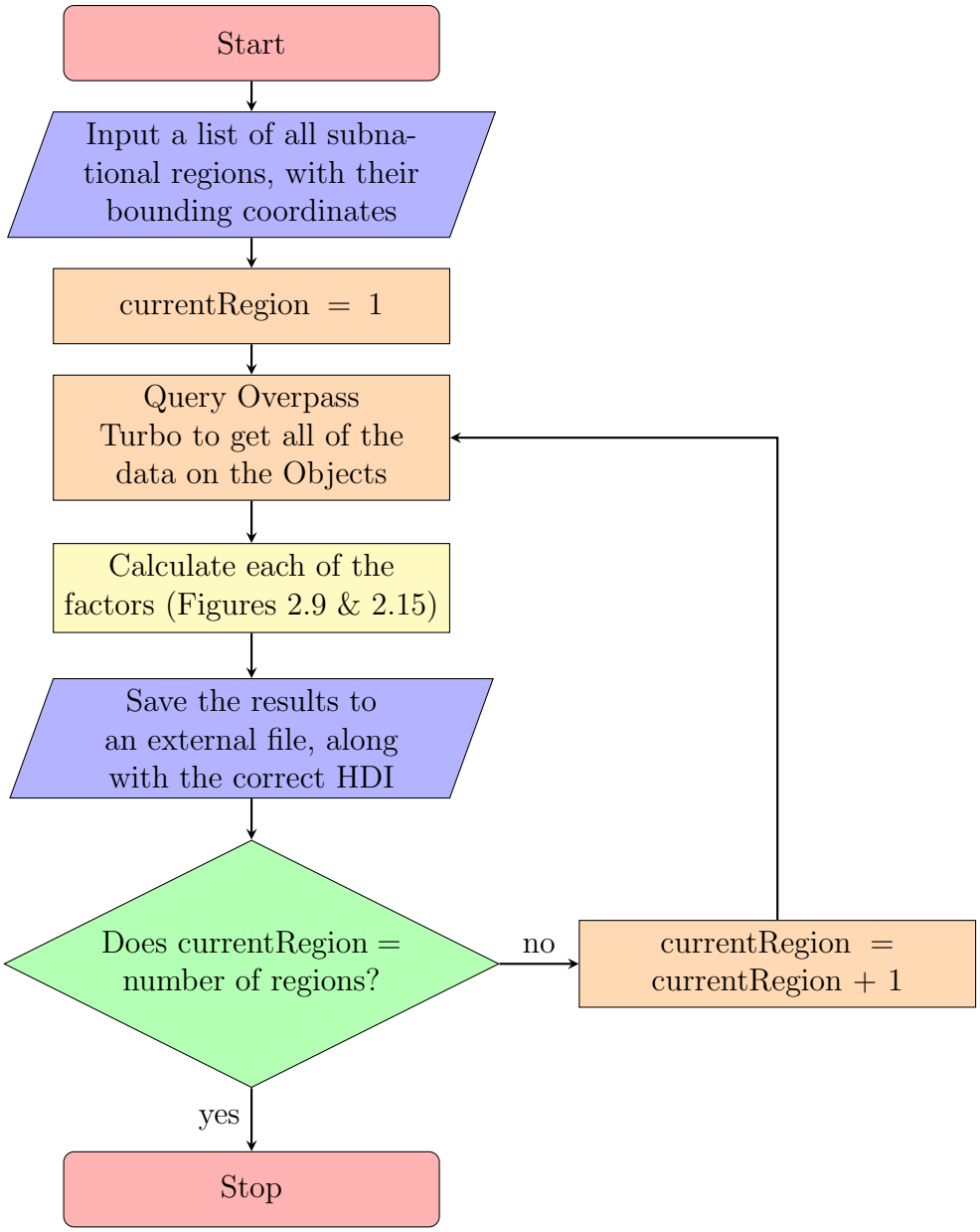


Figure 2.16: Overall Data Collection & Processing Algorithm Flowchart

2.2.5 List of Key Variables

Variables in $A(x, y)$

Identifier	Data Type	Explanation	Justification
`x_objects`	`list[tuple[float]]`	Parameter of the function, holds a list of the coordinates of all the x-objects.	Each coordinate is a tuple as the latitudes and longitudes will never change.

`y_objects`	`list[tuple[float]]`	Parameter of the function, holds a list of the coordinates of all the y-objects.	Each coordinate is a tuple as the latitudes and longitudes will never change.
`max_x_objects`	`int`	Parameter of the function, holds the maximum number of x-object to consider. Was 500 in the explanation above.	Considering every x-object would take a long time, and so some random sampling must be done.
`x_object`	`tuple[float]`	Iterator variable for when looping through `x_objects`.	Is the singular of `x_objects`.
`y_object`	`tuple[float]`	Iterator variable for when looping through `y_objects`.	Is the singular of `y_objects`.
`dists_to_ys`	`list[float]`	Holds the distances to each y-object for a particular x-object.	We must use this to find the minimum distance out of the distances stored.
`min_dists`	`list[float]`	Holds the minimum distances from each x-object to the nearest y-object.	$A(x, y)$ is given by the average of this list.

Table 2.1: List of Key Variables in $A(x, y)$

Variables in $D(x)$

Identifier	**Data Type**	**Explanation**	**Justification**
`x_objects`	`list[tuple[float]]`	Parameter of the function, holds a list of the coordinates of all the x-objects.	Each coordinate is a tuple as the latitudes and longitudes will never change.
`bounding_coords`	`list[tuple[float]]`	Parameter of the function, holds a list of the bounding coordinates of the region.	Each coordinate is a tuple as the latitudes and longitudes will never change.

bounding_vectors	list[column_vector]	Holds the bounding_coords once converted into column vectors.	This is so that we can use matrix multiplication to rotate them.
current_vertex	int	Iterator variable to keep track of which vertex I am dealing with.	Use it to get the vertex from the bounding_vectors as well as the previous and next one.
transformed_vertices	list[column_vector]	Holds the transformed current_vertex as well as the prevous and next one.	We must compute the vectors and anticlockwise angles from this.
anticlockwise_angles	list[float]	Holds all of the anticlockwise angles at each vertex.	We need to sum this to determine if they are actually anticlockwise angles or not.
interior_angles	list[float]	Holds the interior angles of the region the user has drawn.	This is one of the terms in the formula to calculate the area of the region.
area	float	Holds the area of the region the user has drawn.	$D(x)$ is given by the length of x_objects divided by this.

Table 2.2: List of Key Variables in $D(x)$

The data structure column_vector does not come with python, but can be implemented with the numpy module.

Other Variables

Identifier	Data Type	Explanation	Justification
current_region	int	Iterator variable for iterating through the regions.	We must calculate all of the factors for each region.

`dataset_path`	`str`	Stores the file path to the external file for which the neural network will read the data from.	The neural network may need to be trained more than once, due to mistakes, so storing this in an external file would be beneficial.

Table 2.3: List of Other Key Variables in Data Collection & Processing

As the user never interacts with this part of the program, there is no validation that needs to be done (e.g. as stuff is automatically in the right type).

2.3 Multilayer Perceptron

All the information in this section, I learned from 3blue1brown's videos [6] and Micheal Nielsen's e-book [7] on the topic.

2.3.1 Structure

Layers & Neurons

The structure of a multilayer perceptron is similar to that of a graph, in the sense that it has a series of nodes (neurons) and edges (weights). The neurons are arranged in layers, the first of which being the input layer, the last being the output layer, and any in between being hidden layers.

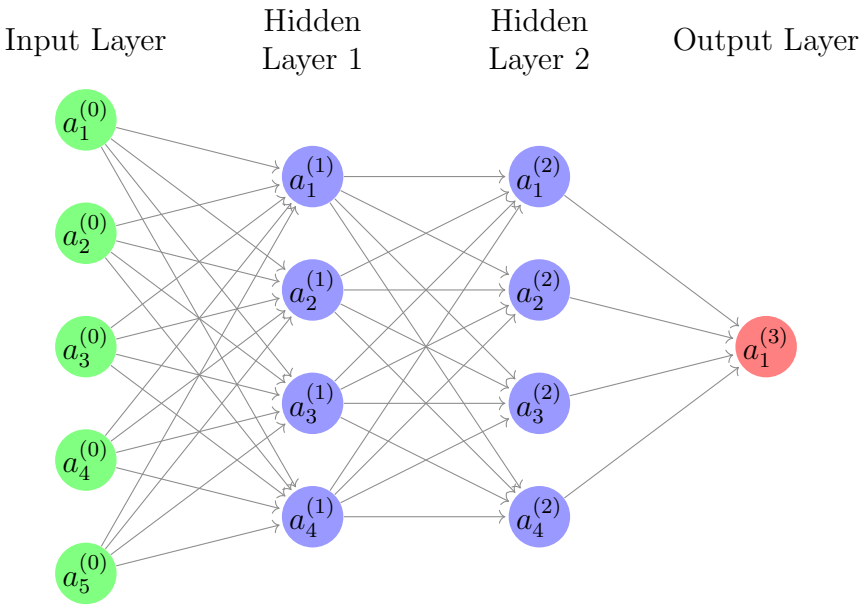


Figure 2.17: Neural Network Structure

Figure 2.17 shows an example neural network, with 5 neurons in the input layer (shown in green), 1 neuron in the output layer (shown in red), and 2 hidden layers each with 4 neurons (shown in blue). "Multilayer Perceptron" is just the fancy official name for this type of machine learning model.

Each neuron will hold a number, which will represent something. In the input layer, the neurons represent the various inputs to the system. In my case, they will be the various factors that may affect the HDI, so my neural network will have 24 neurons in the input layer. In the output layer, the neuron(s) will represent whatever output the system should have. For example, I only want 1 output, the HDI, so there is 1 neuron in the output layer which will hold the predicted HDI.

In the hidden layers, the hope is that the neurons will represent something intermediate in between the input and the output. For example, if the input neurons represent geographical factors, and the output neuron represents the HDI, the neurons in hidden layer 1 might represent the *gist* of how good the schooling is, or how good the healthcare is, and the neurons in hidden layer 2 might represent the *gist* of how these things work together to result in a higher HDI. This means that the original problem of calculating HDI from a set of factors has been decomposed into a series of smaller tasks.

As shown in Figure 2.17, I will use the notation $a_i^{(l)}$ to denote a particular neuron, where l is the index of the layer that it is in, and i is its index within the layer. In general, a network will have L layers, not including the input layer (as the layers are 0-indexed).

Weights

The value of each neuron (that isn't in the input layer) should be based on all the neurons of the previous layer, as shown by the arrows in Figure 2.17.

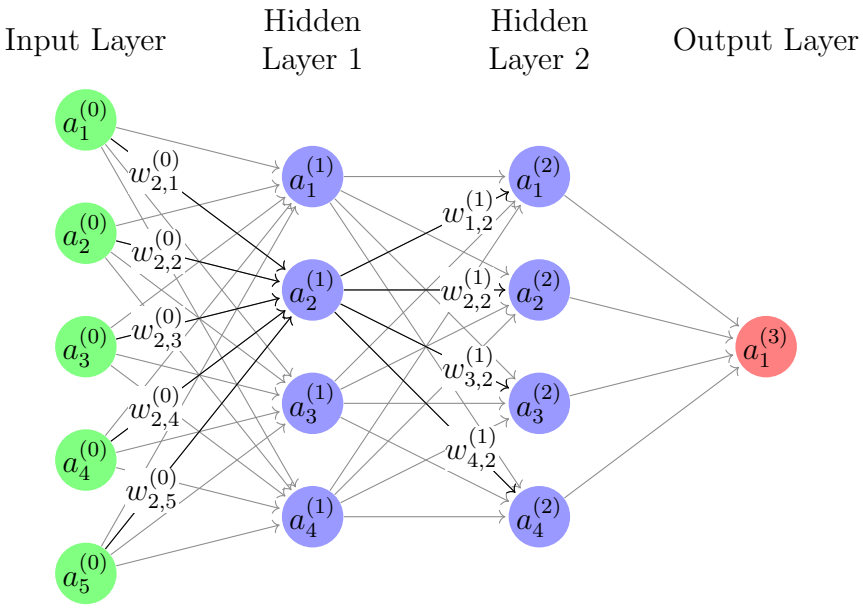


Figure 2.18: Neural Network Weights

I will use the notation $w_{i,j}^{(l)}$ to denote a particular weight, where l is the index of the layer it is pointing from, i is the neuron's index within its layer that the weight is pointing to, and j is the neuron's index within its layer that the weight is pointing from, as shown in Figure 2.18.

Each weight will also hold a value (any real number), which will not be calculated during prediction (like neurons' values are), and so the weights are parameters of the system. Initially, they will be set to random values, which will change during training.

2.3.2 Feedforward Algorithm & Prediction

Weighted Sum

Again, the value of each neuron (that isn't in the input layer) should be based on all the neurons of the previous layer. This can be done using a weighted sum. For example, the equation to calculate the value of $a_1^{(1)}$ in this example neural network would be:

$$a_1^{(1)} = w_{1,1}^{(0)} a_1^{(0)} + w_{1,2}^{(0)} a_2^{(0)} + w_{1,3}^{(0)} a_3^{(0)} + w_{1,4}^{(0)} a_4^{(0)} + w_{1,5}^{(0)} a_5^{(0)} \tag{2.20}$$

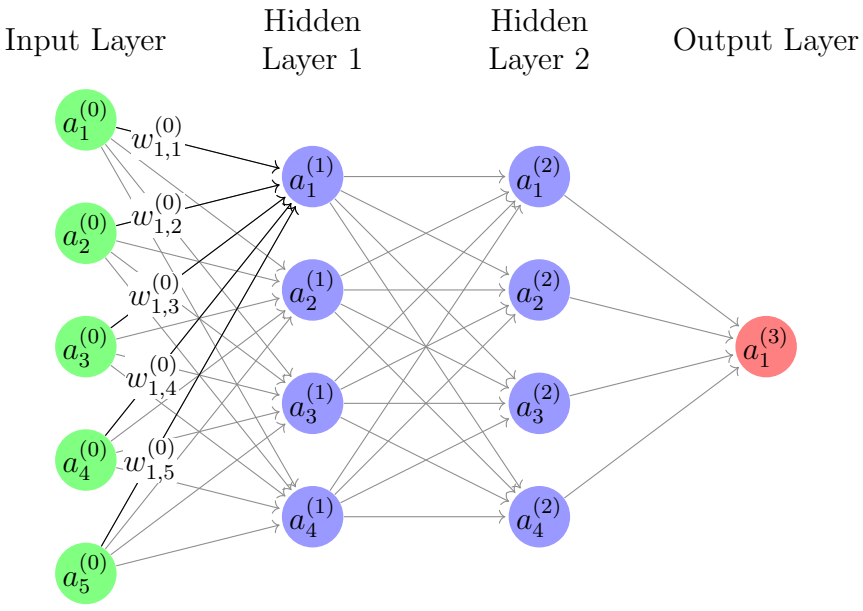


Figure 2.19: Calculating $a_1^{(1)}$ Example

In general, the equation to calculate the value of some neuron $a_i^{(l)}$ in any given network would be:

$$a_i^{(l)} = w_{i,1}^{(l-1)} a_1^{(l-1)} + w_{i,2}^{(l-1)} a_2^{(l-1)} + w_{i,3}^{(l-1)} a_3^{(l-1)} + \cdots + w_{i,n}^{(l-1)} a_n^{(l-1)} \tag{2.21}$$

where n is the number of neurons in layer $l-1$ (the previous layer).

Activation Function

The value of each neuron should also represent how "activated" that neuron is, ideally on a continuous scale from 0 to 1. This is to match what happens in biological neural networks (brains), as this is effectively a model for that. The 0 to 1 range is also suitable in this particular instance, as HDI is also measured on a continuous scale from 0 to 1.

However, the result of this weighted sum could be any real number (as the weights can be any real number), so in order to squish it into the 0 to 1 range, it must be passed through some function, the activation function.

There are many activation functions that could be used, but the one I will use is $\sigma(x)$ ("sigmoid of x"), which is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.22}$$

Why this is suitable is best shown in the graph of $y = \sigma(x)$:

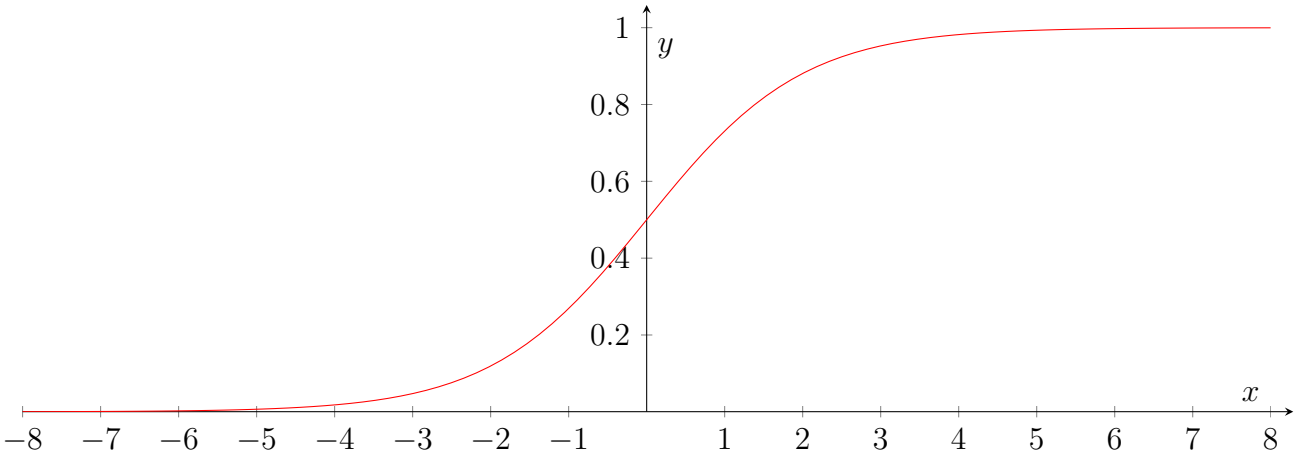


Figure 2.20: Graph of $y = \sigma(x)$

As shown in Figure 2.20, the domain of $\sigma(x)$ is $\mathbb{R}$ (all real numbers), which is what the evaluation of the weighted sum could be, and the range is $0 < \sigma(x) < 1$, which is what we want it to be. Therefore, if the weighted sum is negative, then the activation of the neuron (its value) will be between 0 and 0.5, and if the weighted sum is positive, its activation will be between 0.5 and 1.

As the graph has asymptotes at $y = 0$ and $y = 1$ (which means that the function approaches it but never reaches it), very negative weighted sums will result in activations ≈ 0, and very positive weighted sums will result in activations ≈ 1. The smooth curve in the middle will also mean that the change from 0 to 1 is gradual, and there are no sudden jumps. (The function is also differentiable on its entire domain, which will be helpful in the Backpropogation Algorithm).

Therefore, the equation to calculate the activation of some neuron $a_i^{(l)}$ in any given network would now be:

$$a_i^{(l)} = \sigma\left(w_{i,1}^{(l-1)}a_1^{(l-1)} + w_{i,2}^{(l-1)}a_2^{(l-1)} + w_{i,3}^{(l-1)}a_3^{(l-1)} + \cdots + w_{i,n}^{(l-1)}a_n^{(l-1)}\right) \tag{2.23}$$

where n is the number of neurons in layer $l-1$.

Bias

A neuron may be considered "activated" if its value is ≥ 0.5. For now, this is where the weighted sum is ≥ 0, as $\sigma(0) = 0.5$. But what if a particular neuron should be activated when the weighted sum is (for examle) ≥ 5, or ≥ -87.2?

To achieve this, each neuron (that isn't in the input layer) has a bias, which is another real number which is added onto the weighted sum before the activation function. I will use the notation $b_i^{(l)}$ to denote the bias of neuron $a_i^{(l)}$. The biases (like the weights) are also parameters of the system.

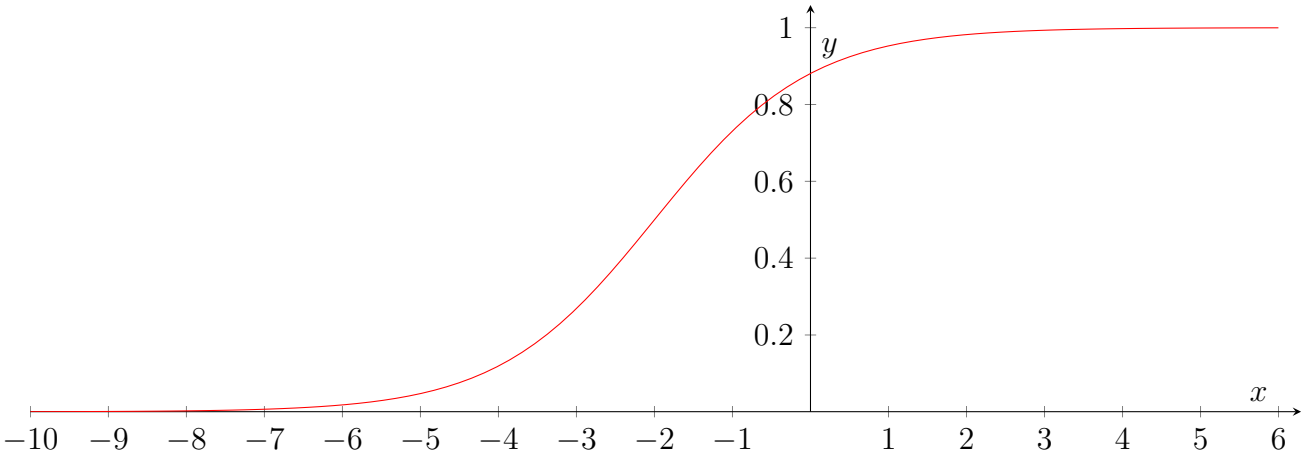


Figure 2.21: Graph of $y = \sigma(x + 2)$

Figure 2.21 shows the graph of $y = \sigma(x + 2)$, which shows how a particular neuron would be activated if its weighted sum is ≥ -2.

Therefore, if a particular neuron $a_i^{(l)}$ should be activated if its weighted sum is $\geq p$, then $b_i^{(l)} = -p$. However, we have no way of knowing what this value of p should be for any of the neurons in the network, as what the neurons represent are too abstract for us to understand. Like the weights, they will be initially set to random values, and will change during training.

The final equation to calculate the activation of some neuron $a_i^{(l)}$ in any given network is:

$$a_i^{(l)} = \sigma\left(w_{i,1}^{(l-1)}a_1^{(l-1)} + w_{i,2}^{(l-1)}a_2^{(l-1)} + w_{i,3}^{(l-1)}a_3^{(l-1)} + \cdots + w_{i,n}^{(l-1)}a_n^{(l-1)} + b_i^{(l)}\right) \tag{2.24}$$

where n is the number of neurons in layer $l - 1$.

Rearranging into Matrices & Column Vectors

Instead of calculating the activation of each neuron 1 by 1, consider calculating an entire layer of neurons at once.

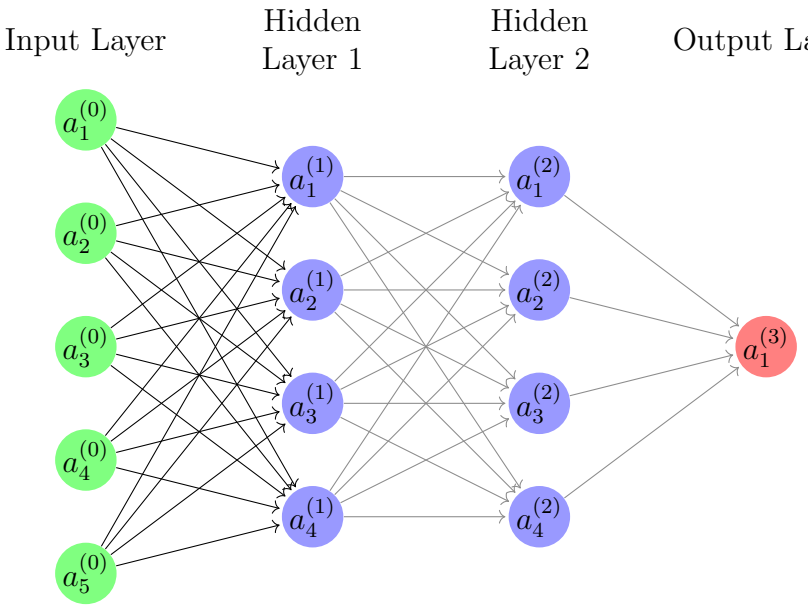


Figure 2.22: Calculating an entire layer of neurons

To do this, we can arrange each layer of neurons and their biases into column vectors (transposed arrays), $\mathbf{a}^{(l)}$ and $\mathbf{b}^{(l)}$ respectively, where l is the index of their layer:

$$\mathbf{a}^{(l)} = \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_n^{(l)} \end{pmatrix}, \mathbf{b}^{(l)} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_n^{(l)} \end{pmatrix} \tag{2.25}$$

where n is the number of neurons in layer l.

Arrange the weights in between each layer into a matrix (2D array), $\mathbf{W}^{(l)}$, where l is the layer the weights are pointing from:

$$\mathbf{W}^{(l)} = \begin{pmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(l)} & w_{m,2}^{(l)} & \cdots & w_{m,n}^{(l)} \end{pmatrix} \tag{2.26}$$

where n is the number of neurons in layer l and m is the number of neurons in layer $l+1$.

Consider the matrix multiplication, $\mathbf{W}^{(l)}\mathbf{a}^{(l)}$:

$$\begin{pmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(l)} & w_{m,2}^{(l)} & \cdots & w_{m,n}^{(l)} \end{pmatrix} \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_n^{(l)} \end{pmatrix} = \begin{pmatrix} w_{1,1}^{(l)}a_1^{(l)} + w_{1,2}^{(l)}a_2^{(l)} + \cdots + w_{1,n}^{(l)}a_n^{(l)} \\ w_{2,1}^{(l)}a_1^{(l)} + w_{2,2}^{(l)}a_2^{(l)} + \cdots + w_{2,n}^{(l)}a_n^{(l)} \\ \vdots \\ w_{m,1}^{(l)}a_1^{(l)} + w_{m,2}^{(l)}a_2^{(l)} + \cdots + w_{m,n}^{(l)}a_n^{(l)} \end{pmatrix} \tag{2.27}$$

The result is a column vector, where each item is a weighted sum of all the neurons in that layer, which is almost the correct column vector for the activations of the neurons in the next layer! To make it correct, we just need to add the bias column vector for the next layer, and pass it through the activation function.

Therefore, the equation to calculate the activations of all the neurons in the next layer, $l+1$ is:

$$\begin{pmatrix} a_1^{(l+1)} \\ a_2^{(l+1)} \\ \vdots \\ a_m^{(l+1)} \end{pmatrix} = \sigma \left(\begin{pmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(l)} & w_{m,2}^{(l)} & \cdots & w_{m,n}^{(l)} \end{pmatrix} \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_n^{(l)} \end{pmatrix} + \begin{pmatrix} b_1^{(l+1)} \\ b_2^{(l+1)} \\ \vdots \\ b_m^{(l+1)} \end{pmatrix} \right) \tag{2.28}$$

where n is the number of neurons in layer l, m is the number of neurons in layer $l+1$ and σ of a column vector $= \sigma$ of all the items in the column vector.

This can be simply written as:

$$\mathbf{a}^{(l+1)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)}\right) \tag{2.29}$$

where σ of a column vector $= \sigma$ of all the items in the column vector, which is the formula that I will be using in my program.

Final Algorithm

In order to fully make a prediction, we just need to iterate this process over all the layers:

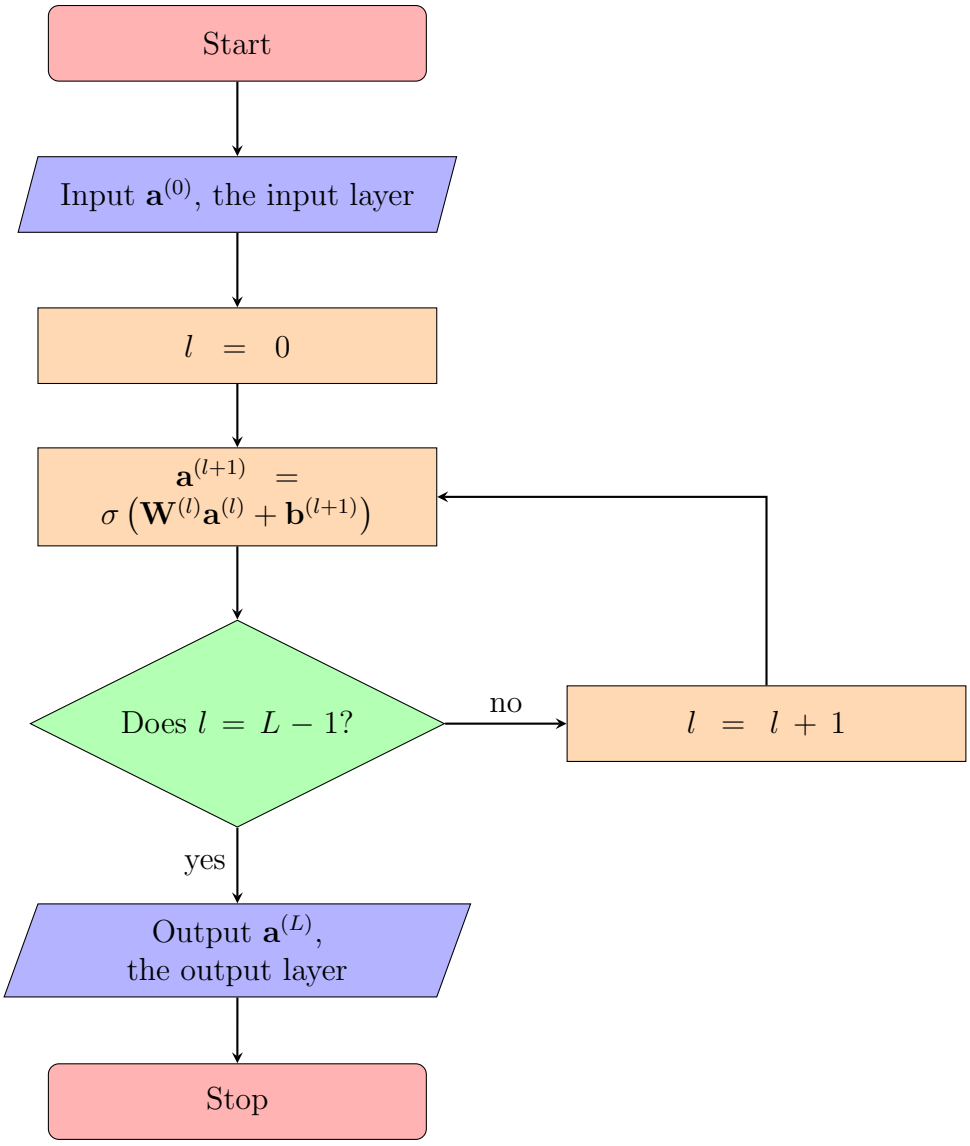


Figure 2.23: Feedforward Algorithm Flowchart

2.3.3 Backpropogation Algorithm & Training

The Cost Function & Gradient Descent

This feedforward algorithm to make predictions works, but if you were to test it right now it would return absolute nonsense. This is because all the weights and biases of the network are initially set to random values, so in reality the network isn't doing anything in particular. In order for it to recognise patterns within factors which may contribute to HDI, it must be trained on a set of examples. The network should also know after a prediction how far off it was from the correct answer.

Therefore, define the "cost" of a single example, c to be:

$$c = \sum_{i=1}^{n} \frac{1}{2}\left(a_i^{(L)} - y_i\right)^2 \tag{2.30}$$

where $a_i^{(L)}$ is the activation of the i^{th} neuron in the output layer, y_i is the "correct" activation, as given by the training example, and n is the number of neurons in layer L. As I will only have 1 neuron in the output layer, this simplifies to:

$$c = \frac{1}{2}\left(a_1^{(L)} - y\right)^2 \tag{2.31}$$

where $a_1^{(L)}$ is the predicted HDI and y is the true HDI. The difference between the true value and the predicted value is squared to always make it positive, as some predictions may be less than the true value and others may be greater than the true value.

Similarly to the activation function, there are many ways of defining the cost, c. I have chosen this one as it is simple to understand how it works, it is very quick to compute (as all you have to do is subtract 2 values, square and half) and it is differentiable on its entire domain, which will be helpful later.

Now define the cost function, C, to be the average cost over many pieces of training data, in terms of all the weights and biases:

$$C\left(w_{1,1}^{(0)}, \ldots, w_{i,j}^{(L-1)}, b_1^{(1)}, \ldots, b_i^{(L)}\right) = ??? \text{ (depends on what the network does)} \tag{2.32}$$

The goal is to find a minimum to this function, as when this function returns a small value, that means that all the predictions that the network gave were close to the true values.

We can do this by using gradient descent. This is where you find the gradient of the function, and then take a small step in the direction such that the gradient will decrease the most. This will work as the gradient at minimums of functions = 0. In other words, we must find out how much each of the weights and biases affects the cost, and then using this information, change the weights and biases such that the cost will decrease.

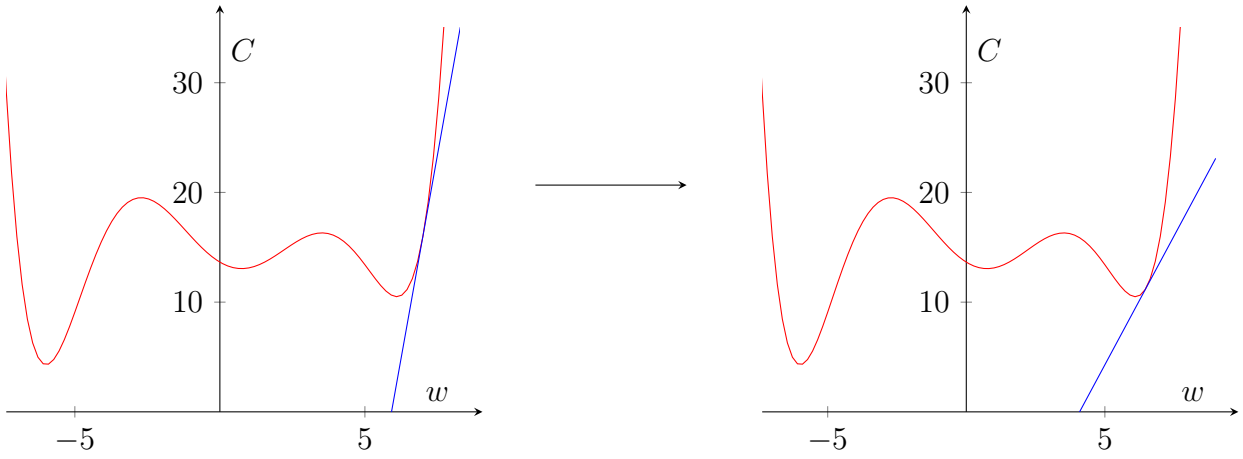


Figure 2.24: Gradient Descent Example

Figure 2.24 shows how C may vary with some weight w, and how gradient descent will be used to find a minimum of C with respect w. Note that it may not converge on the absolute minimum, but it will converge onto some local minimum.

Error of a Neuron, $\delta_i^{(l)}$

To do this, it will help to find out how much each neuron affects the output of the network, or the cost. This will be called the error of a neuron.

First, define $z_i^{(l)}$ to be neuron $a_i^{(l)}$'s activation before it was passed into the activation function:

$$z_i^{(l)} = w_{i,1}^{(l-1)} a_1^{(l-1)} + w_{i,2}^{(l-1)} a_2^{(l-1)} + w_{i,3}^{(l-1)} a_3^{(l-1)} + \cdots + w_{i,n}^{(l-1)} a_n^{(l-1)} + b_i^{(l)} \tag{2.33}$$

Now we can define the error of a neuron $a_i^{(l)}$, $\delta_i^{(l)}$ to be:

$$\delta_i^{(l)} = \frac{\mathrm{d}C}{\mathrm{d}z_i^{(l)}} \tag{2.34}$$

which represents the quantity answered by the question: *"If I make a tiny change to $z_i^{(l)}$, $\Delta z_i^{(l)}$ what is the resulting change in C?"*.

If $\delta_i^{(l)}$, is close to 0, there isn't much we can do to $z_i^{(l)}$ to have a big effect on C. On the other hand, if $\delta_i^{(l)}$ is large, then we only have to change $z_i^{(l)}$ a little bit to have a big effect on C. Therefore, there is a heuristic sense of how $\delta_i^{(l)}$ measures the sensitivity, or error, of a neuron.

The plan is to find the error of the output layer, then backpropogate that to find the error of previous layers, and finally find the change in the cost function C with respect to the weights and biases in terms of these errors.

Error of the Output Layer, $\boldsymbol{\delta}^{(L)}$

The error of the neurons in the output layer will therefore be given by:

$$\delta_i^{(L)} = \frac{\mathrm{d}C}{\mathrm{d}z_i^{(L)}} \tag{2.35}$$

as L is the number of layers in the network (not including the input layer), and therefore represents the output layer. However, this is not easily computable, and should be rearranged into something that is.

We can break up this expression using the chain rule for differentiation:

$$\delta_i^{(L)} = \frac{\mathrm{d}C}{\mathrm{d}a_i^{(L)}} \frac{\mathrm{d}a_i^{(L)}}{\mathrm{d}z_i^{(L)}} \tag{2.36}$$

As $a_i^{(l)} = \sigma\left(z_i^{(l)}\right)$ by definition (from equations 2.24 and 2.33),

$$\frac{\mathrm{d}a_i^{(L)}}{\mathrm{d}z_i^{(L)}} = \sigma'\left(z_i^{(L)}\right) \tag{2.37}$$

where $\sigma'(x)$ is the derivative of the sigmoid function.

As the output of C is given by $\sum_{i=1}^{n} \frac{1}{2}\left(a_i^{(L)} - y_i\right)^2$ by definition (from equation 2.30),

$$\frac{\mathrm{d}C}{\mathrm{d}a_i^{(L)}} = a_i^{(L)} - y_i \tag{2.38}$$

Therefore, from equations 2.36, 2.37 and 2.38,

$$\delta_i^{(L)} = \left(a_i^{(L)} - y_i\right)\left(\sigma'\left(z_i^{(L)}\right)\right) \tag{2.39}$$

which is all easily computable.

As done before with $\mathbf{a}^{(l)}$ and $\mathbf{b}^{(l)}$, arrange each of the z-values and errors into column vectors, $\mathbf{z}^{(l)}$ and $\boldsymbol{\delta}^{(l)}$, where l is the index of their layer:

$$\mathbf{z}^{(l)} = \begin{pmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_n^{(l)} \end{pmatrix}, \boldsymbol{\delta}^{(l)} = \begin{pmatrix} \delta_1^{(l)} \\ \delta_2^{(l)} \\ \vdots \\ \delta_n^{(l)} \end{pmatrix} \tag{2.40}$$

where n is the number of neurons in layer l.

$\boldsymbol{\delta}^{(L)}$ must therefore be given by:

$$\boldsymbol{\delta}^{(L)} = \left(\mathbf{a}^{(L)} - \mathbf{y}\right) \odot \left(\sigma'\left(\mathbf{z}^{(L)}\right)\right) \tag{2.41}$$

where $\mathbf{y}$ is the column vector of the expected outputs and $\odot$ represents the operation to compute the element-wise product of 2 column vectors with the same dimension. For example,

$$\begin{pmatrix} 2 \\ 6 \\ 3 \end{pmatrix} \odot \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix} = \begin{pmatrix} 6 \\ 24 \\ 15 \end{pmatrix} \tag{2.42}$$

Error of the Previous Layer in terms of the current one

Now that we have the error of the output layer, $\boldsymbol{\delta}^{(L)}$, we can attempt to backpropogate by finding the error of each layer before it. In other words, we want $\boldsymbol{\delta}^{(l-1)}$ in terms of $\boldsymbol{\delta}^{(l)}$ and other easily computable terms. Consider $\delta_i^{(l-1)}$:

$$\delta_i^{(l-1)} = \frac{\mathrm{d}C}{\mathrm{d}z_i^{(l-1)}} \tag{2.43}$$

We can again break this up using the chain rule:

$$\delta_i^{(l-1)} = \sum_{j=1}^{n} \frac{\mathrm{d}C}{\mathrm{d}z_j^{(l)}} \frac{\mathrm{d}z_j^{(l)}}{\mathrm{d}z_i^{(l-1)}} \tag{2.44}$$

except this time we must sum over each j from 1 to n, where n is the number of neurons in layer l. This is because the activation of all the neurons in layer l is based on $z_i^{(l-1)}$, so we must add up all the small changes to C from layer l to calculate the overall effect from $z_i^{(l-1)}$.

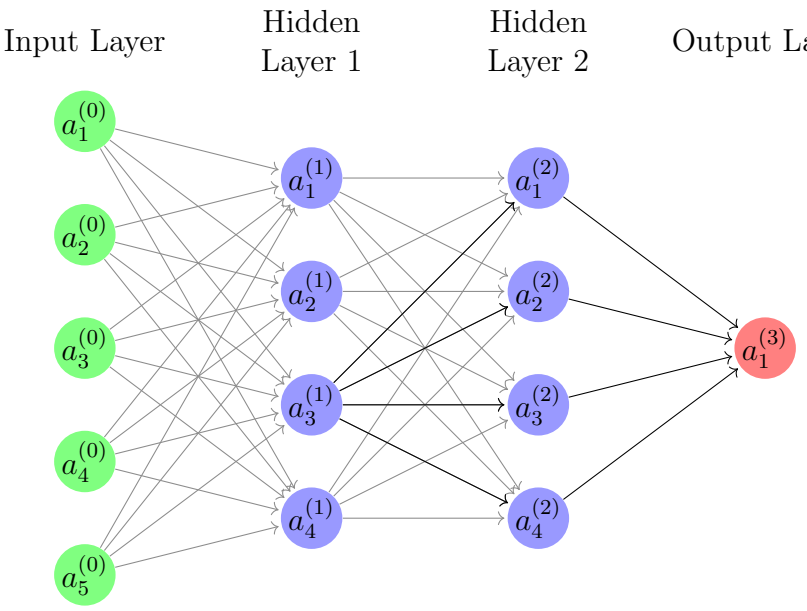


Figure 2.25: $z_i^{(l-1)}$ Affects all neurons in layer l, which all affect C

By definition, $\dfrac{\mathrm{d}C}{\mathrm{d}z_j^{(l)}} = \delta_j^{(l)}$, so equation 2.44 simplifies to:

$$\delta_i^{(l-1)} = \sum_{j=1}^{n} \frac{\mathrm{d}z_j^{(l)}}{\mathrm{d}z_i^{(l-1)}} \delta_j^{(l)} \tag{2.45}$$

Using equation 2.33, we can write $z_j^{(l)}$ as:

$$z_j^{(l)} = w_{j,1}^{(l-1)} a_1^{(l-1)} + w_{j,2}^{(l-1)} a_2^{(l-1)} + \cdots + w_{j,i}^{(l-1)} a_i^{(l-1)} + \cdots + w_{j,n}^{(l-1)} a_n^{(l-1)} + b_j^{(l)} \tag{2.46}$$

where n is the number of neurons in layer $l-1$ and i is some index of a particular neuron in the previous layer which we care about.

As $a_i^{(l-1)} = \sigma\left(z_i^{(l-1)}\right)$ by definition (from equations 2.24 and 2.33),

$$z_j^{(l)} = w_{j,1}^{(l-1)} a_1^{(l-1)} + w_{j,2}^{(l-1)} a_2^{(l-1)} + \cdots + w_{j,i}^{(l-1)} \sigma\left(z_i^{(l-1)}\right) + \cdots + w_{j,n}^{(l-1)} a_n^{(l-1)} + b_j^{(l)} \tag{2.47}$$

Therefore,

$$\frac{\mathrm{d}z_j^{(l)}}{\mathrm{d}z_i^{(l-1)}} = w_{j,i}^{(l-1)} \sigma'\left(z_i^{(l-1)}\right) \tag{2.48}$$

Substituting that into equation 2.45 gives:

$$\delta_i^{(l-1)} = \sum_{j=1}^{n} w_{j,i}^{(l-1)} \sigma' \left(z_i^{(l-1)} \right) \delta_j^{(l)} \tag{2.49}$$

Consider the matrix multiplication, $\left(\mathbf{W}^{(l-1)}\right)^T \boldsymbol{\delta}^{(l)}$, where $\mathbf{M}^T$ represents the transposed matrix of matrix $\mathbf{M}$:

$$\begin{pmatrix} w_{1,1}^{(l-1)} & w_{2,1}^{(l-1)} & \cdots & w_{n,1}^{(l-1)} \\ w_{1,2}^{(l-1)} & w_{2,2}^{(l-1)} & \cdots & w_{n,2}^{(l-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,m}^{(l-1)} & w_{2,m}^{(l-1)} & \cdots & w_{n,m}^{(l-1)} \end{pmatrix} \begin{pmatrix} \delta_1^{(l)} \\ \delta_2^{(l)} \\ \vdots \\ \delta_n^{(l)} \end{pmatrix} = \begin{pmatrix} w_{1,1}^{(l-1)}\delta_1^{(l)} + w_{2,1}^{(l-1)}\delta_2^{(l)} + \cdots + w_{n,1}^{(l-1)}\delta_n^{(l)} \\ w_{1,2}^{(l-1)}\delta_1^{(l)} + w_{2,2}^{(l-1)}\delta_2^{(l)} + \cdots + w_{n,2}^{(l-1)}\delta_n^{(l)} \\ \vdots \\ w_{1,m}^{(l-1)}\delta_1^{(l)} + w_{2,m}^{(l-1)}\delta_2^{(l)} + \cdots + w_{n,m}^{(l-1)}\delta_n^{(l)} \end{pmatrix} \tag{2.50}$$

Again, this is almost the result that we want as we just need to multiply each term by $\sigma' \left(z_i^{(l-1)} \right)$ to get a column vector of elements in the form of equation 2.49. Therefore, the equation to backpropogate the error from one layer to the previous is given by:

$$\boldsymbol{\delta}^{(l-1)} = \left(\left(\mathbf{W}^{(l-1)}\right)^T \boldsymbol{\delta}^{(l)} \right) \odot \left(\sigma' \left(\mathbf{z}^{(l-1)} \right) \right) \tag{2.51}$$

which again, is all easily computable.

Gradient of the Cost Function

Now that we have $\delta_i^{(l)}$ for any given l and i in the network in terms of easily computable terms, we can try and find the change in the cost function C with the respect to all the weights and biases, which is the gradient of the cost function, which is what we need to train the model. In other words, we are trying to find $\dfrac{\mathrm{d}C}{\mathrm{d}w_{i,j}^{(l)}}$ and $\dfrac{\mathrm{d}C}{\mathrm{d}b_i^{(l)}}$ for any l, i, j in the network, in terms of errors, in order to take a small step in the opposite direction (as to decrease the gradient and find a minimum).

We can again break up these expressions using the chain rule into:

$$\frac{\mathrm{d}C}{\mathrm{d}z_i^{(l+1)}} \frac{\mathrm{d}z_i^{(l+1)}}{\mathrm{d}w_{i,j}^{(l)}} \text{ and } \frac{\mathrm{d}C}{\mathrm{d}z_i^{(l)}} \frac{\mathrm{d}z_i^{(l)}}{\mathrm{d}b_i^{(l)}} \text{ respectively} \tag{2.52}$$

By definition, $\dfrac{\mathrm{d}C}{\mathrm{d}z_i^{(l)}} = \delta_i^{(l)}$ so this simplifies to:

$$\frac{\mathrm{d}z_i^{(l+1)}}{\mathrm{d}w_{i,j}^{(l)}} \delta_i^{(l+1)} \text{ and } \frac{\mathrm{d}z_i^{(l)}}{\mathrm{d}b_i^{(l)}} \delta_i^{(l)} \text{ respectively} \tag{2.53}$$

From equation 2.33,

$$\frac{\mathrm{d}z_i^{(l+1)}}{\mathrm{d}w_{i,j}^{(l)}} = a_j^{(l)} \text{ and } \frac{\mathrm{d}z_i^{(l)}}{\mathrm{d}b_i^{(l)}} = 1 \tag{2.54}$$

Therefore, from equations 2.53 and 2.54,

$$\frac{\mathrm{d}C}{\mathrm{d}w_{i,j}^{(l)}} = a_j^{(l)}\delta_i^{(l+1)} \text{ and } \frac{\mathrm{d}C}{\mathrm{d}b_i^{(l)}} = \delta_i^{(l)} \tag{2.55}$$

which is what we need.

Therefore, for a given training example index x, we must change $\mathbf{b}^{(l)}$ by an amount $-\eta\boldsymbol{\delta}_x^{(l)}$ where $\boldsymbol{\delta}_x^{(l)}$ the error for the neurons in layer l for the specific training example x (this notation also follows for $\mathbf{a}$ and $\mathbf{z}$) and η is some constant, the learning rate. The negative sign is involved as to carry out gradient descent we need to take a step in the direction opposite to the gradient.

Similarly, we must change $\mathbf{W}^{(l)}$ by an amount $-\eta\boldsymbol{\delta}_x^{(l+1)}\left(\mathbf{a}_x^{(l)}\right)^T$. The activations column vector has been transposed so that when the error column vector is multiplied by it the result is a matrix with the same dimensions of the weight matrix.

Final Algorithm

Now all we need to do is iterate this process over multiple training examples and then change the gradient by an average over all the pieces of training data.

Therefore, the equations to update the weights and biases are:

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \frac{\eta}{m}\sum_{x=1}^{m}\boldsymbol{\delta}_x^{(l+1)}\left(\mathbf{a}_x^{(l)}\right)^T \tag{2.56}$$

and

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \frac{\eta}{m}\sum_{x=1}^{m}\boldsymbol{\delta}_x^{(l)} \tag{2.57}$$

where m is the number of training examples.

Each training example will be given in the format $(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ is a column vector of activations for the input layer and $\mathbf{y}$ is a column vector for the expected outputs for the output layer.

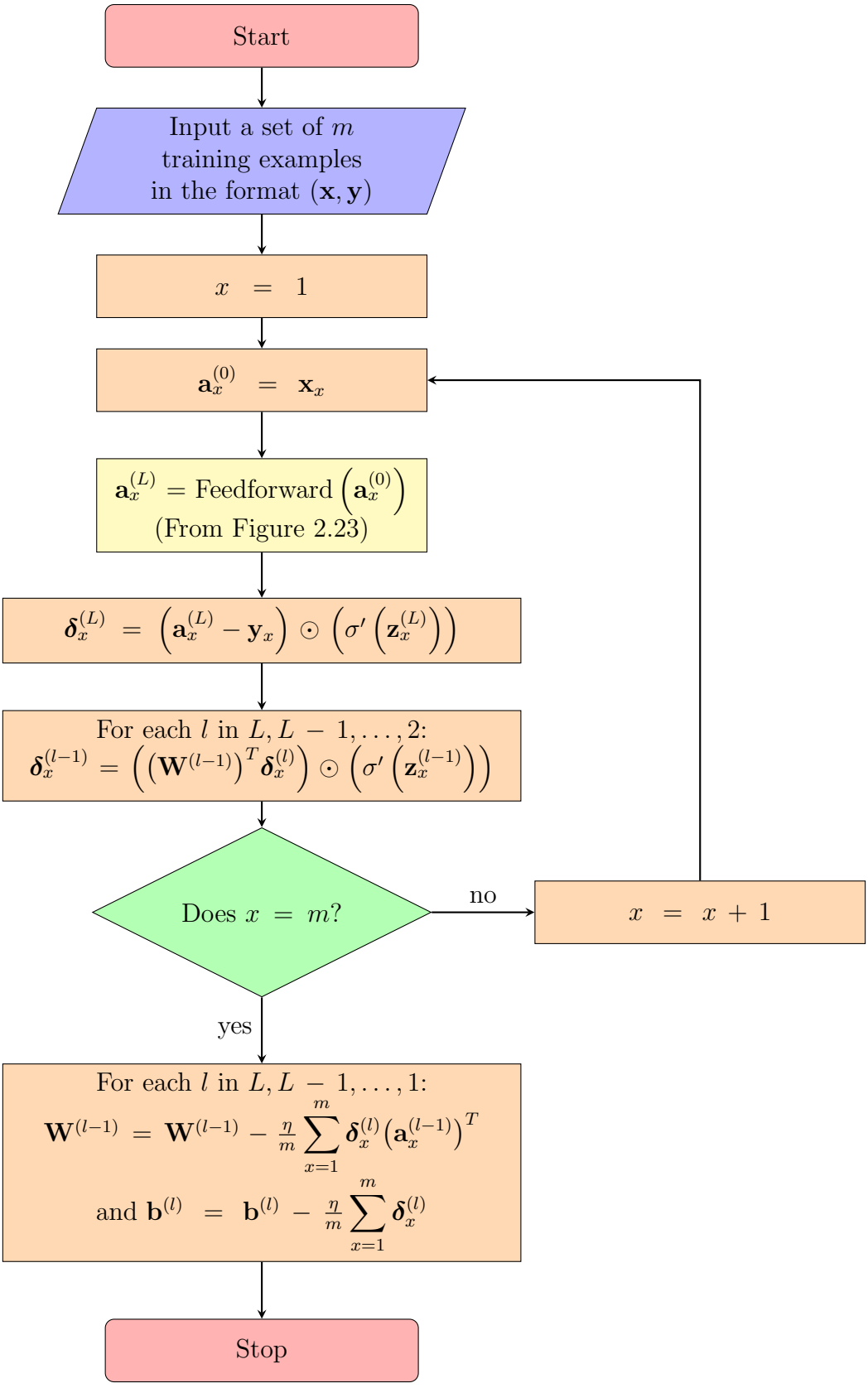


Figure 2.26: Backpropogation Algorithm Flowchart

2.3.4 Stochastic Gradient Descent

In practice, averaging over all the training examples is not very efficient, as you have to create a copy of the column vectors $\mathbf{a}$, $\mathbf{z}$ and $\boldsymbol{\delta}$ for each layer and training example, which can get very large very fast.

To combat this, instead break up the training data into mini batches which are easier to handle, while still having enough to represent a range of things that the network may want to look for. The training examples are also randomly shuffled before being sorted into mini batches to ensure that the contents of each one are different every time.

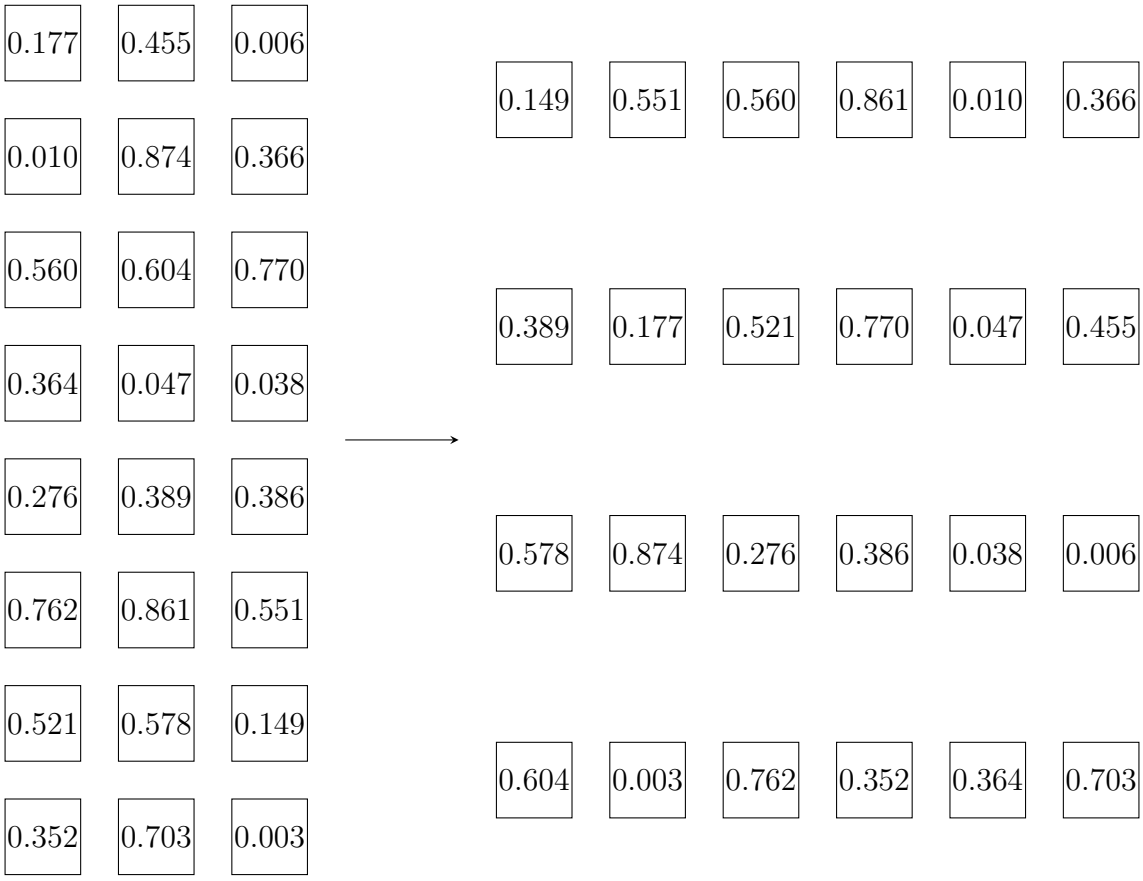


Figure 2.27: Shuffling into Mini Batches

When carrying out the gradient descent, this will have the effect of taking a step in some direction close to that which decreases the gradient by the most, instead of going directly in that direction. This is called Stochastic Gradient Descent.

Finally, this entire process should be repeated multiple times, over a number of "epochs", in order to ensure that it has enough time to learn.

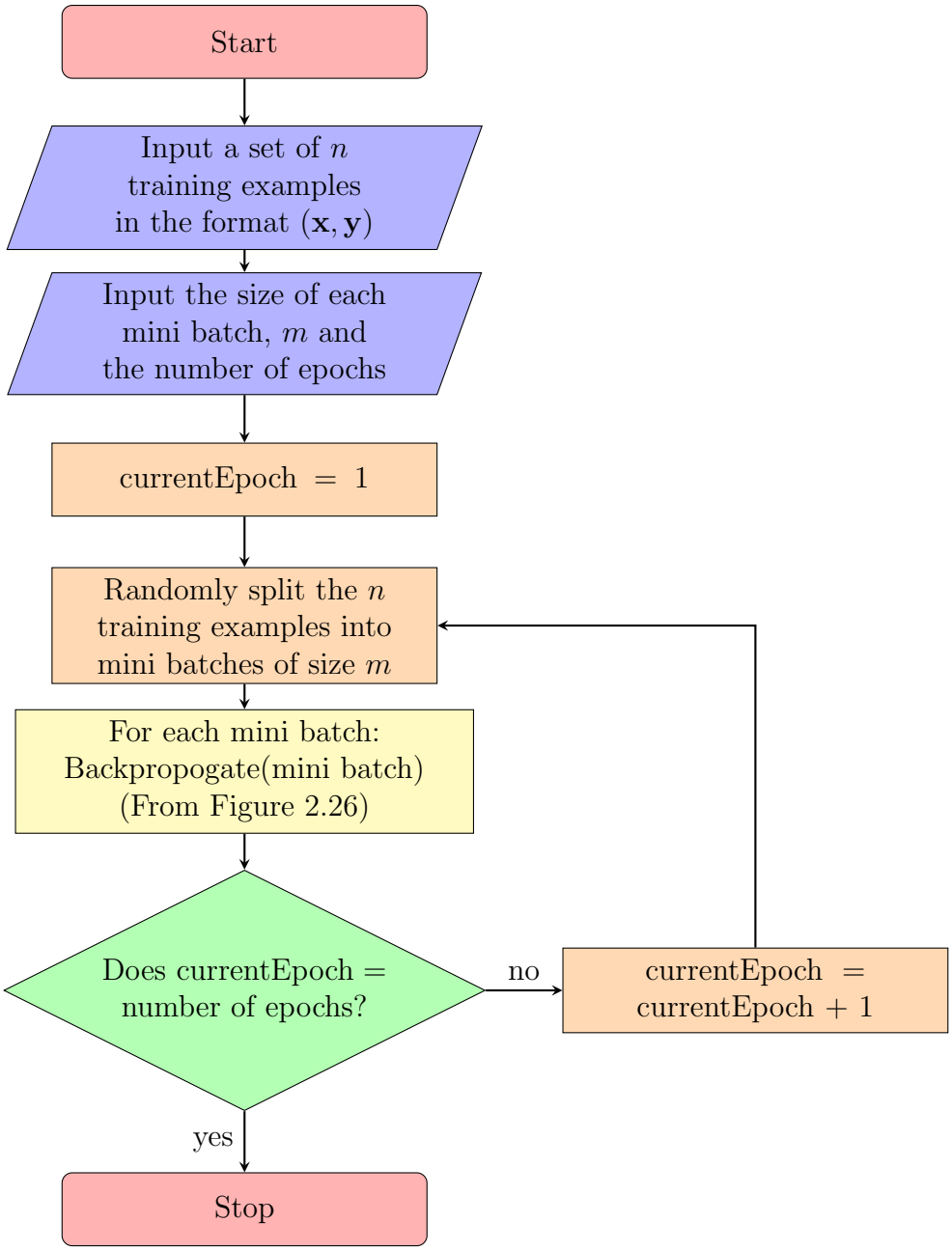


Figure 2.28: Full Training Algorithm Flowchart

2.3.5 Class Diagram

class MultilayerPerceptron(object)
layer_sizes: list[int] L: int activations: list[column_vector] weights: list[matrix] biases: list[column_vector] z: list[column_vector] errors: list[column_vector] training_activations: list[list[column_vector]] training_z: list[list[column_vector]] training_errors: list[list[column_vector]]
def __init__(self, layer_sizes) def predict(self, input_data) def train(self, examples, mini_batch_size, num_epochs, learning_rate) def feedforward(self, a0) def backpropogate(self, examples) def sigmoid(self, x) def sigmoid_prime(self, x) def save_model(self, filename) def load_model(self, file_path)

Figure 2.29: Multilayer Perceptron Class Diagram

The attribute `layer_sizes` is a list of integers, which represent the number of neurons in each layer. The attributes `L` and `z` represent the respective variables in the mathematical explanation above, `activations` represents the $\mathbf{a}$ vectors, `weights` represents the $\mathbf{W}$ matrices, `biases` represents the $\mathbf{b}$ vectors, and `errors` represents the $\boldsymbol{\delta}$ vectors. The data structures `column_vector` and `matrix` are not built into python, but come with the library `numpy`. The attributes `training_activations`, `training_z` and `training_errors` will hold the values for each of the training examples in a mini batch, as all of them need to be computed for each example before the gradient is calculated (which is why they are in an extra `list`).

The constructor method (`def __init__`) only takes in the layer sizes, as everything else will either be defined during training or prediction, or be initially set to random values. The other methods are as you would expect, with functions for prediction and training that are explained above. There are also the methods `def save_model` and `def load_model`, as the model will be trained beforehand and then the GUI will query that model. Therefore, we need a way of saving the weights and biases to an external file and then loading them from it at a later time.

All the methods will be public functions, as python does not allow for anything else. However, in theory, they would all be procedures except for `def predict`, `def sigmoid` and `def sigmoid_prime`, which will all return `float`s, while the rest will return `None`.

There is not a "List of key variables" for this section, as all the results of calculations will

be stored in the attributes in the `class MultilayerPerceptron`, and have therefore been described above.

2.4 User Interface

There are many features I would like to add eventually, which I have included in the Structure Diagram (Figure 2.1). I will not be including these in this section, and will be saving them for the next prototype. I will only be planning for the user entering their region on a map, the HDI being predicted, and the suggestions for improvements being made, as I believe these are the most essential feautures.

2.4.1 Interface Sketch

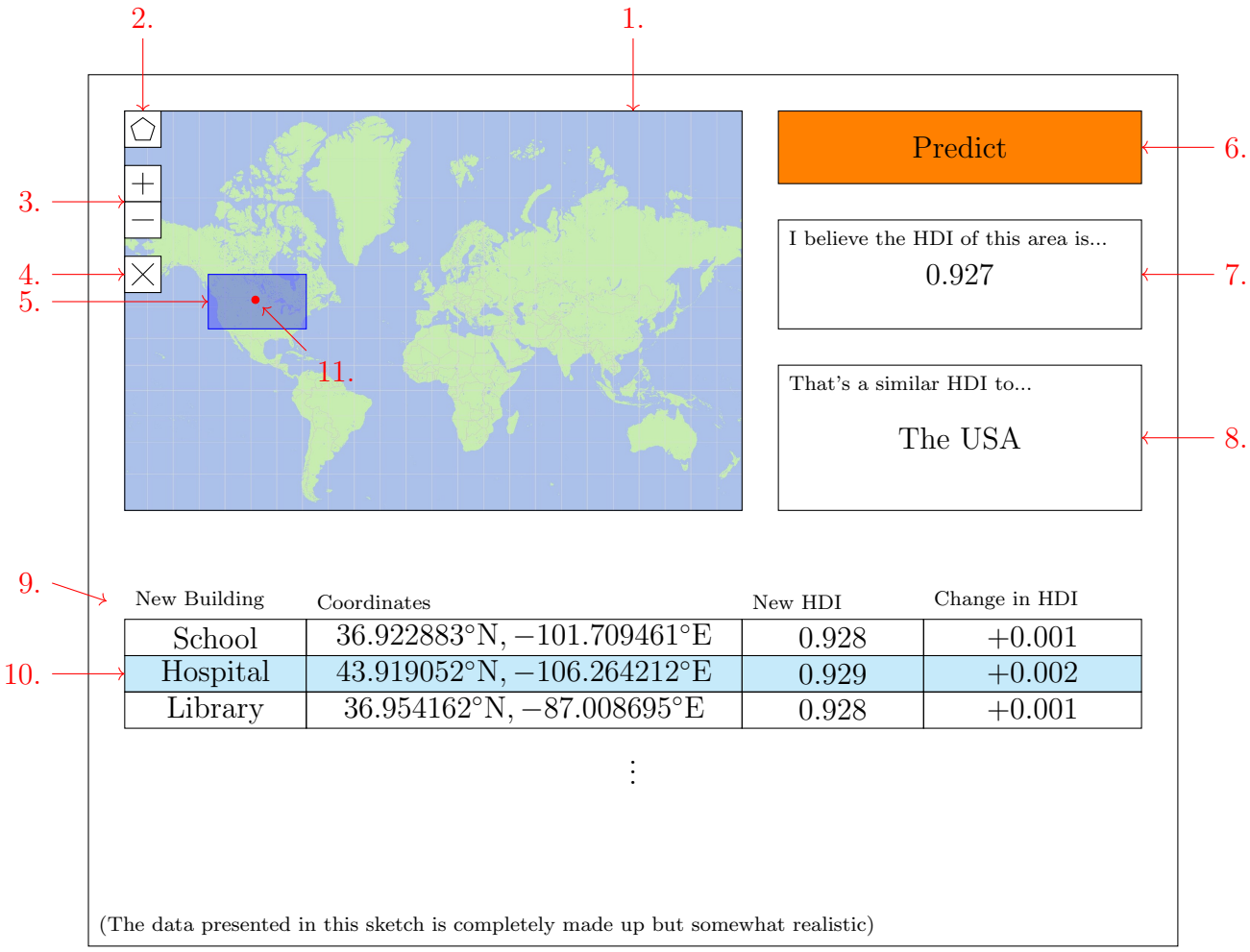
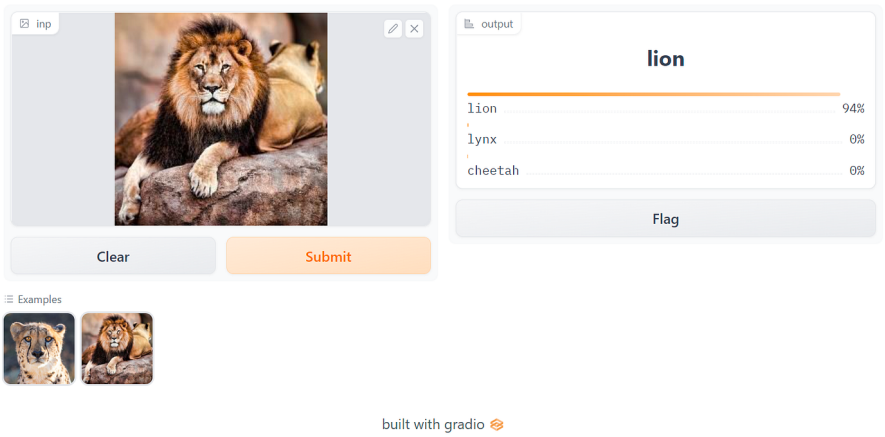


Figure 2.30: Technical Interface Sketch

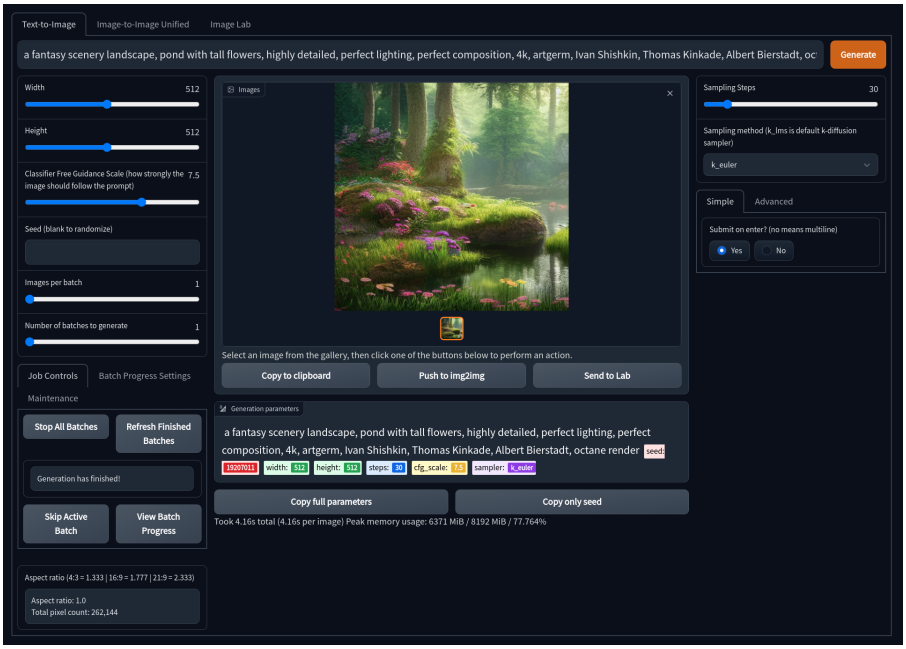
The labelled items shown in Figure 2.30 are:

1. Map for the user to draw on
2. Button to toggle between using the mouse for polygon drawing and moving
3. Zoom in and out buttons
4. Cancel Button
5. Example area the user may draw
6. Predict Button
7. Predicted HDI
8. A comparison to a country with a similar HDI
9. A list of suggestions to improve the HDI
10. A selected suggestion to view in more detail (that the user has selected)
11. The location on the map for the selected suggestion

Figure 2.30 shows the general structure of the user interface, while the style will be provided by the `gradio` python module. Exampled of the style can be seen in Figure 2.31. It will automatically adapt to light mode or dark mode based on the device's or browser's settings.



(a) Light Mode



(b) Dark Mode

Figure 2.31: Interface Style Examples

Additional features that will be added in later prototypes (such as an analysis of how much each factor affects the others or entering factors manually) will be accessed from different tabs at the top of the interface.

2.4.2 Input Validation

As the only input the user is providing is that of drawing on the map to select their region (for now), only that needs to be validated. I will ensure that the user can only draw polygons on the map, the polygons must be completed, and the polygons cannot self intersect. The mapping module I will be using, `folium` already accounts for this, except for the self

intersection. However, this can be detected by using part of the $D(x)$ algorithm, as shown in Figure 2.15.

The user should also only be able to interact with the map and any data visualisations. The GUI module I will be using, `gradio` also already accounts for this.

2.4.3 Usability Features

Map & Map Buttons

The map should be the biggest thing on the screen, as it is the main source of input the user will be providing. It should also be as big as possible to accomodate those with smaller screens, so that they can still accurately pinpoint locations.

The button to draw a polygon on the map will be displayed as a regular pentagon. This is because there are often distinct tools to draw rectangles or triangles, which would be represented by regular 4-sided or 3-sided polygons respectively. Therefore, if a regular 5-sided polygon is used (a pentagon) there is less likely to be confusion about what the button does, while still having a small number of sides (a large number of sides will look too similar to a circle). It also suggests that the user will not be able to draw freely (as this is often represented by a pencil), which they will not be able to do.

The buttons to zoom in and out will be represented by a plus and minus sign. This is also the standard design for this type of button. The zoom in button and zoom out buttons will be right next to each other, to minimise the consequence in accidentally zooming in/out one too many times.

The button to cancel will be represented by an X, as this is also a common symbol used for this type of button. It will be in the same approximate location as the other 3 buttons, as they all have similar functions.

The region the user draws will be shown in blue, and filled with a slightly transparent blue. This is so that they can see the region that they are drawing in real time, while also still being able to see the map beneath it.

Prediction of HDI

The button to make the prediction (and submit the region they have drawn) will be the largest button on the screen, and in the primary colour. By default, the primary colour of `gradio` interfaces is orange, as it compliments the (default, dark mode) dark blue background. This will make it very easy for the user to see what they need to do.

The 2 pieces of information below the predict button (the predicted HDI and the country comparison) will have the largest text size out of all the elements in the interface. This is because this is most likely the main information the user is trying to find, and so should be easy to find. Placing these elements next to the predict button is for the same reason.

Suggestions to Improve HDI

The suggestions to improve the HDI will be given in a table, as they will all follow a similar format: *"Build a new (building) at (these coordinates) to improve the HDI by (this much),*

which increases it to (this value).”. This information can be efficiently organised into a table, in order to retrieve the most important information quickly.

Each row of the table (each suggestion) may be selected to show the location of that new building on the map. This is because reading coordinates such as $36.922883°\text{N}, -101.709461°\text{E}$ does not give an intiutive sense of location, especially within a small area.

2.4.4 Finding an Optimal Place for a New Building

When suggestions are being made to the user to improve the HDI, they will be in the format: “Build a new *some building* at *some coordinates*”. Which building will be determined from a predefined list of possible suggestions, which will match the factors I am considering. The coordinates at which to place it are harder to determine.

In theory, increasing the Density factor involving this building, and decreasing the Average Distance factors involving this building will increase the HDI. No matter where we place the new building in the region, the Density factor will increase by the same amount, as the number of that building will always increase by 1, and the area of the region will never change.

Therefore, we only need to consider how the position of a new building will affect the Average Distance factors. Any place we choose will decrease it by some amount, but we need to find a place that will decrease it by a significant amount. Finding the *exact* optimal place is very difficult, so instead I will be using a heuristic, “good enough” approach.

One way that I would consider is “good enough”, is finding the midpoint of all the positions of the other objects, to find where to place this new building. For example, if we wish to decrease $A\,(\text{House}, \text{Hospital})$, we could place a hospital in the midpoint of all the positions of the houses. This will involve taking a mean of all the latitude and longitude coordinates of all the houses.

However, there may be some other factors involving hospitals, such as $A\,(\text{School}, \text{Hospital})$ for which we would need to follow the same procedure. This will result in a different optimal place to what was calculated from the houses.

To make a compromise between all the factors that involve the building we are trying to place, we can average the previously calculated optimal places for each different factor to get an overall optimal place to build the building.

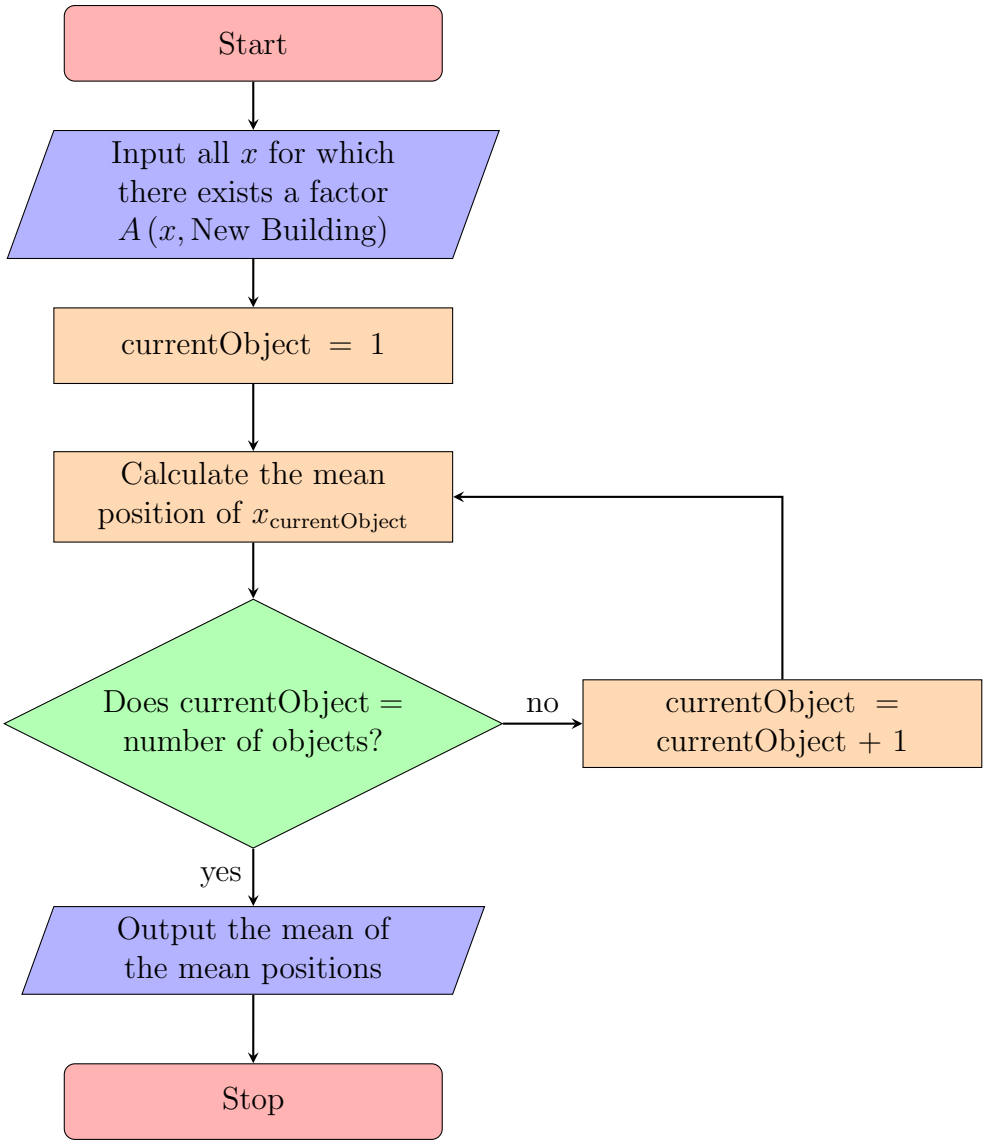


Figure 2.32: Finding an Optimal Place for New Building Flowchart

For validation, this function should only be called when there exists at least 1 factor in the form $A(x, \text{New Building})$.

I intend to implement a better algorithm to find an optimal place for a new building in future prototypes, however this will do for now.

2.4.5 Overall Algorithm for User Predicting HDI

The user will draw their region on the map, from that the relevant data will be retrieved from OSM using Overpass Turbo, and then the Density and Average Distance factors will be calculated. This will be fed into the pre-trained neural network, which will predict the HDI.

On top of this, a series of suggestions will be made to improve the HDI. The full list of suggestions I would like to make are (in no partucular order):

1. Building a new School
2. Building a new Hospital
3. Building a new Place of Worship
4. Building a new Police Station
5. Building a new Restaurant
6. Building a new Slot Machine
7. Building a new Library
8. Building a new Pharmacy

For each of the suggested buildings, the position at which to build it must be calculated, and then the factors must be calculated again before being fed into the pre-trained neural network.

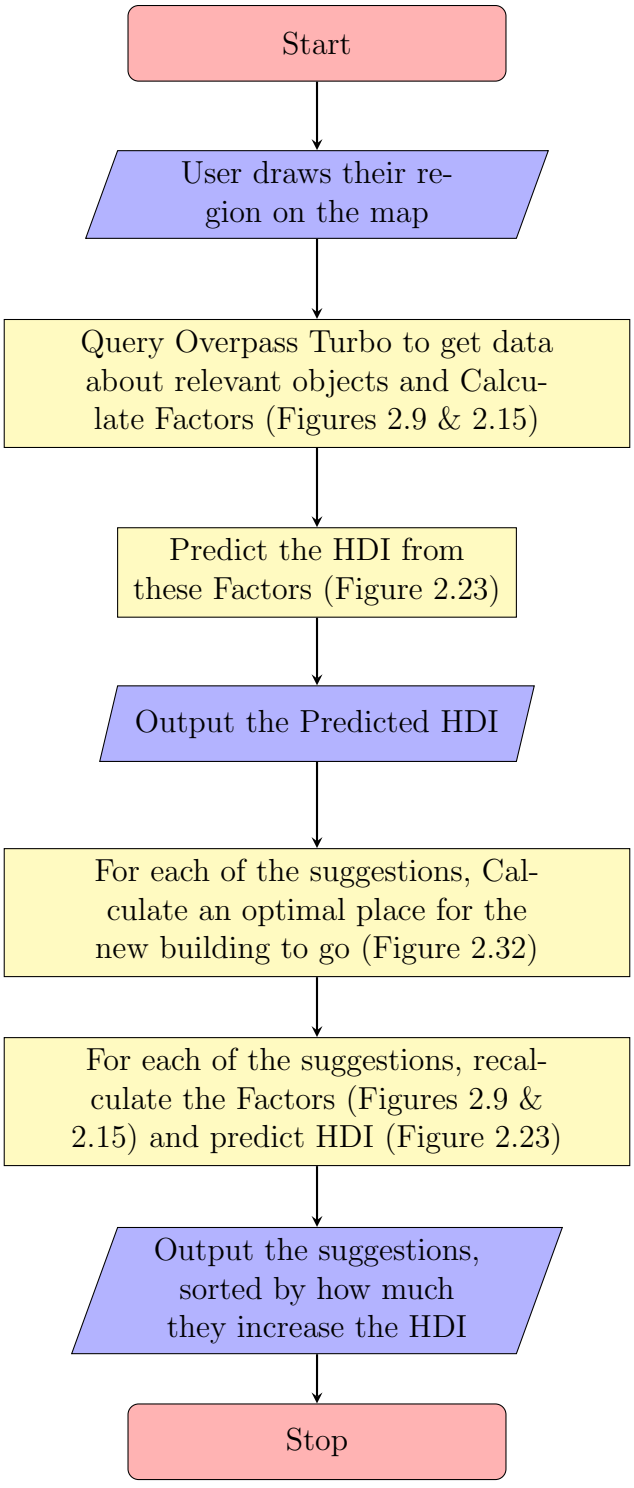


Figure 2.33: Overall User Predicting HDI Algorithm Flowchart

2.4.6 List of Key Variables

Variables in Finding Optimal Place for New Building

Identifier	Data Type	Explanation	Justification
`new_building_type`	`str`	Holds the type of new building to be places.	This is used to determine if there are any other factors which contain this type of building.
`other_objects`	`list[str]`	Holds each of the otehr types of object to consider.	We must iterate over this to find an optimal place for a new building.
`current_object`	`str`	Iterator variable for `other_objects`	This will be used to keep track of which object we are currently considering
`mean_positions`	`list[tuple[float]]`	Holds the mean positions of all the other objects we are considering	This is a list of latitude and longitude coordinates, so a tuple is used as before.

Table 2.4: List of Key Variables in Finding Optimal Place for New Building

Other Variables

Identifier	Data Type	Explanation	Justification
`app`	`object`	Contains the `gradio` app, which will be ran on the local web server.	The app (highest level "interface block") must be stored in a variable to pass to the web server.
`neural_network`	`object`	Stores the pre-trained model for predicting HDI.	This must be called upon when the user draws their region on the map.
`suggestions_types`	`list[str]`	List containing the different types of suggestion to be made.	Each `str` will be the name of the new building type to place.
`suggestions`	`dict[float]`	A dictionary, containing each predicted HDI, each with a key.	The key will be a string containing the type of new building.

Table 2.5: List of Other Key Variables in User Interface

2.5 Data

2.5.1 Data Structures

Coordinates

Any time I am dealing with coordinates on the surface of the Earth, I will be using a `tuple[float]`. The latitude and longitude are stored as `float`s, as they are real numbers that may not be integers. They are stored in a `tuple`, as the length of the data structure will never change (as that would correspond to adding a new perpendicular dimension), and the value of the coordinate never needs to be changed, and so `tuple`s being immutable and static is suitable.

Coordinates in 3D space can also be stored as `tuple[float]`, with 3 elements instead of 2 (corresponding to the xyz coordinate space).

Column Vectors & Matrices

A matrix is essentially a 2D array of numbers, which have some special operations. I will be using the `numpy` (as `np`) module's `np.matrix` for this, as it takes in a python 2D `list`, and turns it into a matrix, validating the fact that all of its entries are numbers (`int` or `float`, type check), and that all sublists are the same length (length check). This, along with the `np.matmul()` function (for multiplying matrices) will give all the required functionality of matrices.

A column vector is just a matrix with 1 column, and so `np.matrix` can still be used, except I will need to pass in a python `list` of `list`s of length 1, containing each element of the column vector.

Column vectors and matrices are used in the neural network and for calculating the area of a region.

Layers of Neurons & Weights

The neural network is mainly where column vectors & matrices will be used, as each layer of activations, biases, errors and $\mathbf{z}$-values will be stored in their own column vectors, with weights being stored in matrices (as each of the neurons in 1 layer is connected to all of the neurons in the next, we need some form of 2D data structure).

Lists of Things

Any time I need to store multiple of a similar thing, I will be using a `list`. This is because lists are both dynamic and mutable, so I can add, remove or change elements as I like.

For example, the column vectors corresponding to the layers of neurons will be stored in their own lists, so that they can be easily retrieved from the index of their layer. I will also be using a `list` to store a series of coordinates (`tuple[float]`) which define a particular region, or just contain the positons of objects we are interested in.

Neural Network & File Structure

Once the neural network has been fully trained to predict HDI, I would like to be able to store it in an external file, so that predictions can be made without having to go through the training process again.

Only the weights and biases need to be saved, as the particular activations and errors are dependant on what the input layer's activations are, while the weights and biases define how the network can make predictions.

Therefore, I will first store the network's `weights` and `biases` attributes in a python `dict`, such as `{"weights": ..., "biases": ...}`, in order to store them in 1 place. I will then use the `pickle` module in order to save this variable (containing a single `dict`) to a `.pkl` file, which is a binary file that can store variables. When the model needs to be loaded (e.g. in the GUI), this file can be read to get the `dict` back, and then the weights and biases can be retrieved using the keys `"weights"` and `"biases"`.

2.5.2 Testing Data for Development

I plan to have done all the tests under each milestone by the time I reach it. Once all the tests have gone successfuly, across all the milestones (for this prototype), I will ask for my stakeholders' opinions, to see what needs improving.

I will still be testing code in small increments as I am developing, to ensure that I do not make any silly mistakes.

Milestone 1 – Data Collection & Processing

No.	Test	Inputs	Expected Output/Justification
T_1	Overpass Turbo Query	Closed Polygon, Open Polygon, Valid object type, Invalid object type	The query should be able to handle both open and closed polygons as regions, and shouldn't break when an invalid object type is passed in.
T_2	Formula for distance between 2 points	2 Points in the same hemisphere, 2 Points in opposite hemispheres, 2 Points either side of the international date line, 2 Points whose shortest distance crosses the North Pole	The formula should be resiliant to all of the listed edge cases, and will be checked using an online calculator.

T_3	$A(x, y)$	Less than 500 x-objects, 500 x-objects, More than 500 x-objects, 0 x-objects	If more than 500 x-objects, random sampling should happen to find 500 random x-objects. If there are 0 x-objects, `None` should be returned, as there is no distance to find.
T_4	$D(x)$	Region drawn Clockwise, Region drawn Anticlockwise, Region drawn in mulitple loops of the Earth, Region that self-intersects	This function should be able to handle any region that does not self-intersect, no matter how it is drawn and how many times it wraps around the Earth. The correct area will be found on the internet (as I will be testing subnational regions)

Table 2.6: Testing Data for Milestone 1

Milestone 2 – Multilayer Perceptron

Testing to see if the neural network is actually working is incredibly difficult. Due to the many limitatons as a result of a (relatively) small data set, wrong predictions may be made, while the neural network is working perfectly. Therefore, in order to truly test if it is working, it must be trained on a different data set, which certainly has enough data. This data set will have nothing to do with HDI (as if it did, I would be using it for training anyway), but instead a data set on handwritten digits.

Testing each individual algorithm in the neural network is also very difficult, as there are so many small calculations that go into it, doing it by hand to check would take days. Therefore, I will only be testing the neural network once I have fully implemented it. However, I will still be doing the small, incremental tests to ensure no silly mistakes.

No.	**Test**	**Inputs**	**Expected Output/Justification**
T_5	Neural Network works	Using a *different* data set, e.g. handwritten digits, which certainly has enough training data	This is to ensure that the neural network works, and if it doesn't work for predicting HDI, it is as a result of a lack of training data.

Table 2.7: Testing Data for Milestone 2

Milestone 3 – Initial User Interface

No.	Test	Inputs	Expected Output/Justification
T_6	Overall Flow	User Draws Region and then Predicts, User tries to predict before selecting Region, User draws region and doesn't predict	The only valid sequence of steps the user should be able to take is drawing the region and then making a prediction.
T_7	Optimal Place for New Building	Building for which there exists 1 Factor, Building for which there exists more than 1 Factor, Building for which there exists 0 factors	This function should take an average over all of the factors the building is involved in, while throwing an error if it has 0 factors.
T_8	Suggestions fed back into Neural Network	All 8 sugestions for where to place new buildings	A new predicted HDI for each suggestion

Table 2.8: Testing Data for Milestone 3

3. Developing the Coded Solution of the 1st Prototype

3.1 Milestone 1 – Data Collection & Processing

3.1.1 Success Criteria

By the end of this milestone, I hope to have all the training data for the neural network to train on.

No.	Success Criterion	Justification
S_1	Find suitable HDI training data for the Neural Network	The neural network must train on pre-calculated HDI values for certain areas, along with their respective density and average distance factors.
S_2	Retrieve OSM data for a given area using Overpass Turbo	This will allow me to calculate useful geographical factors to predict the HDI from instead of more human ones (which are harder to determine).
S_3	Calculate average distance factors from a given area	The average distance from *some object* to *some other object* is also a good measure to consider. For example $A\,(\text{House}, \text{School})$ is a useful factor as school children shouldn't have to travel long distances for education.
S_4	Calculate density factors from a given area	The number of *some object* per unit area is a good way of measuring how many of that object are in a particular area while removing the bias of larger areas.

Table 3.1: Success Criteria for Milestone 1

To determine if I have met this success criteria, I will be using the following tests:

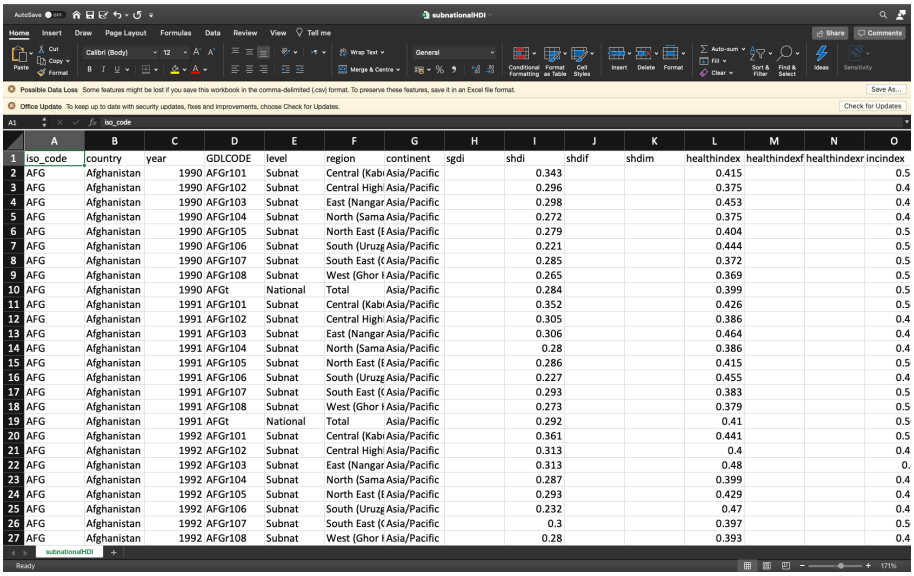
No.	Test	Inputs	Expected Output/Justification
T_1	Overpass Turbo Query	Closed Polygon, Open Polygon, Valid object type, Invalid object type	The query should be able to handle both open and closed polygons as regions, and shouldn't break when an invalid object type is passed in.
T_2	Formula for distance between 2 points	2 Points in the same hemisphere, 2 Points in opposite hemispheres, 2 Points either side of the international date line, 2 Points whose shortest distance crosses the North Pole	The formula should be resiliant to all of the listed edge cases, and will be checked using an online calculator.
T_3	$A(x, y)$	Less than 500 x-objects, 500 x-objects, More than 500 x-objects, 0 x-objects	If more than 500 x-objects, random sampling should happen to find 500 random x-objects. If there are 0 x-objects, `None` should be returned, as there is no distance to find.
T_4	$D(x)$	Region drawn Clockwise, Region drawn Anticlockwise, Region drawn in mulitple loops of the Earth, Region that self-intersects	This function should be able to handle any region that does not self-intersect, no matter how it is drawn and how many times it wraps around the Earth. The correct area will be found on the internet (as I will be testing subnational regions)

Table 3.2: Testing Data for Milestone 1

3.1.2 Part 1 – Collecting HDI Data

The first thing I need is a list of all subnational regions, with their corresponding true HDI. As stated before, I will download this from the Global Data Lab [4].

The full dataset contains lots of unnecessary data for my project, as this gives the HDI (and many other indicators) over time, whereas I just want the most recent HDI for each subnational region.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	iso_code	country	year	GDLCODE	level	region	continent	sgdi	shdi	shdif	shdim	healthindex	healthindexf	healthindexr	incindex
2	AFG	Afghanistan	1990	AFGr101	Subnat	Central (Kabı	Asia/Pacific		0.343			0.415			0.5
3	AFG	Afghanistan	1990	AFGr102	Subnat	Central High	Asia/Pacific		0.296			0.375			0.4
4	AFG	Afghanistan	1990	AFGr103	Subnat	East (Nangar	Asia/Pacific		0.298			0.453			0.4
5	AFG	Afghanistan	1990	AFGr104	Subnat	North (Sama	Asia/Pacific		0.272			0.375			0.4
6	AFG	Afghanistan	1990	AFGr105	Subnat	North East (E	Asia/Pacific		0.279			0.404			0.5
7	AFG	Afghanistan	1990	AFGr106	Subnat	South (Uruzg	Asia/Pacific		0.221			0.444			0.5
8	AFG	Afghanistan	1990	AFGr107	Subnat	South East (K	Asia/Pacific		0.285			0.372			0.5
9	AFG	Afghanistan	1990	AFGr108	Subnat	West (Ghor I	Asia/Pacific		0.265			0.369			0.5
10	AFG	Afghanistan	1990	AFGt	National	Total	Asia/Pacific		0.284			0.399			0.5
11	AFG	Afghanistan	1991	AFGr101	Subnat	Central (Kabı	Asia/Pacific		0.352			0.426			0.5
12	AFG	Afghanistan	1991	AFGr102	Subnat	Central High	Asia/Pacific		0.305			0.386			0.4
13	AFG	Afghanistan	1991	AFGr103	Subnat	East (Nangar	Asia/Pacific		0.306			0.464			0.4
14	AFG	Afghanistan	1991	AFGr104	Subnat	North (Sama	Asia/Pacific		0.28			0.386			0.4
15	AFG	Afghanistan	1991	AFGr105	Subnat	North East (E	Asia/Pacific		0.286			0.415			0.5
16	AFG	Afghanistan	1991	AFGr106	Subnat	South (Uruzg	Asia/Pacific		0.227			0.455			0.4
17	AFG	Afghanistan	1991	AFGr107	Subnat	South East (K	Asia/Pacific		0.293			0.383			0.5
18	AFG	Afghanistan	1991	AFGr108	Subnat	West (Ghor I	Asia/Pacific		0.273			0.379			0.5
19	AFG	Afghanistan	1991	AFGt	National	Total	Asia/Pacific		0.292			0.41			0.5
20	AFG	Afghanistan	1992	AFGr101	Subnat	Central (Kabı	Asia/Pacific		0.361			0.441			0.5
21	AFG	Afghanistan	1992	AFGr102	Subnat	Central High	Asia/Pacific		0.313			0.4			0.4
22	AFG	Afghanistan	1992	AFGr103	Subnat	East (Nangar	Asia/Pacific		0.313			0.48			0.
23	AFG	Afghanistan	1992	AFGr104	Subnat	North (Sama	Asia/Pacific		0.287			0.399			0.4
24	AFG	Afghanistan	1992	AFGr105	Subnat	North East (E	Asia/Pacific		0.293			0.429			0.4
25	AFG	Afghanistan	1992	AFGr106	Subnat	South (Uruzg	Asia/Pacific		0.232			0.47			0.4
26	AFG	Afghanistan	1992	AFGr107	Subnat	South East (K	Asia/Pacific		0.3			0.397			0.5
27	AFG	Afghanistan	1992	AFGr108	Subnat	West (Ghor I	Asia/Pacific		0.28			0.393			0.4

Figure 3.1: Full dataset containing unnecessary data

Therefore, I must write an algorithm to generate a suitable `.csv` file, containing only useful information.

`data_processing / process_hdi.py`

```
1  import csv
2
3  #retrieve data
4  with open('data/subnationalHDI.csv', 'r') as file:
5      reader = csv.DictReader(file)
```

Code Snippet 3.1: Initialising `csv` reader

The `csv` module contains many useful methods for handling `csv` files. I am opening `'data/subnationalHDI.csv'` in read mode, and saving it to the variable `file`. The `with` statement automatically closes the file once I am done with it. The `csv.DictReader` is a class that can read the csv file, and convert each record into a pyton `dict`.

data_processing / process_hdi.py

```
    newcsv = []
    for record in reader:
        if record['year'] == '2022': # the most recent year
            newRecord = {
                "country": record["country"],
                "region": record["region"],
                "code": record["GDLCODE"],
                "hdi": record["shdi"]
            } # only the most useful information
            newcsv.append(newRecord)
            print(newRecord)
```

Code Snippet 3.2: Refining the HDI Data

Here, I define a `list`, `newcsv` to hold all the new (nescessary) records. For each of the records in the full dataset, we must determine if it is from the most recent year (2022). If it is, we make a new record of only the most important information (the region's name, its code (which will be used to uniquely identify it), and its HDI), and append this to `newcsv`.

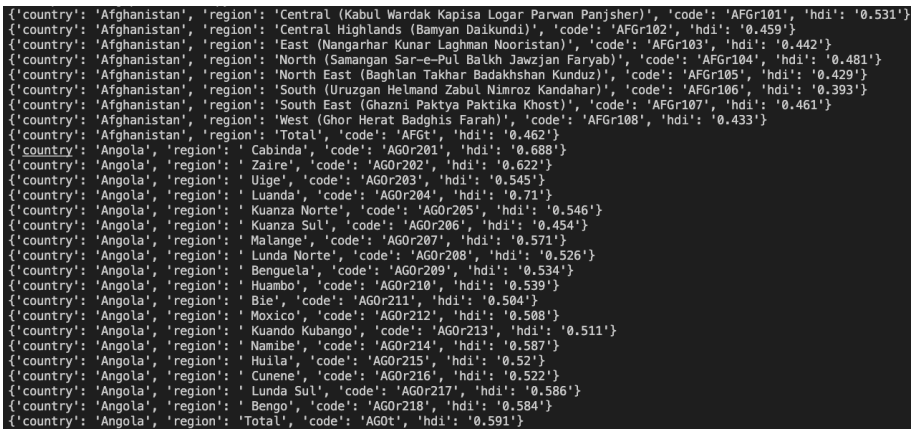


Figure 3.2: The new records

Figure 3.2 shows each `newRecord`, which are correct. Therefore, I can now add these to a new `csv` file.

No.	Test	Input	Type	Expectation	Output
t_1	Refined HDI Data	–	–	Only region and HDI of the most recent year	As Expected

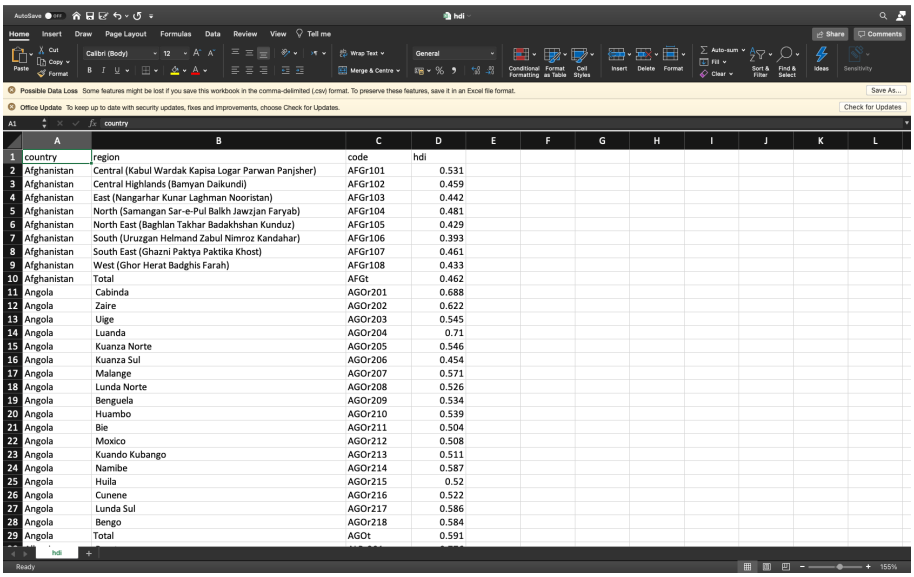
Table 3.3: Testing Table for Code Snippets 3.1 & 3.2

data_processing / process_hdi.py

```
16  #write to new csv
17  with open('data/hdi.csv', 'w') as file:
18      field_names = ['country', 'region', 'code', 'hdi']
19      writer = csv.DictWriter(file, fieldnames=field_names)
20      writer.writeheader()
21      for record in newcsv:
22          writer.writerow(record)
```

Code Snippet 3.3: Writing back to `csv`

Here, I must open `'data/hdi.csv'` in write mode, to ensure I can make changes to the empty file. The `csv.DictWriter` does the same thing as the `csv.DictReader` but in reverse, so we must specify the field names. After this, we iterate over the `newcsv` `list` and write all of the new records.



	A	B	C	D
1	country	region	code	hdi
2	Afghanistan	Central (Kabul Wardak Kapisa Logar Parwan Panjsher)	AFGr101	0.531
3	Afghanistan	Central Highlands (Bamyan Daikundi)	AFGr102	0.459
4	Afghanistan	East (Nangarhar Kunar Laghman Nooristan)	AFGr103	0.442
5	Afghanistan	North (Samangan Sar-e-Pul Balkh Jawzjan Faryab)	AFGr104	0.481
6	Afghanistan	North East (Baghlan Takhar Badakhshan Kunduz)	AFGr105	0.429
7	Afghanistan	South (Uruzgan Helmand Zabul Nimroz Kandahar)	AFGr106	0.393
8	Afghanistan	South East (Ghazni Paktya Paktika Khost)	AFGr107	0.461
9	Afghanistan	West (Ghor Herat Badghis Farah)	AFGr108	0.433
10	Afghanistan	Total	AFGt	0.462
11	Angola	Cabinda	AGOr201	0.688
12	Angola	Zaire	AGOr202	0.622
13	Angola	Uige	AGOr203	0.545
14	Angola	Luanda	AGOr204	0.71
15	Angola	Kuanza Norte	AGOr205	0.546
16	Angola	Kuanza Sul	AGOr206	0.454
17	Angola	Malange	AGOr207	0.571
18	Angola	Lunda Norte	AGOr208	0.526
19	Angola	Benguela	AGOr209	0.534
20	Angola	Huambo	AGOr210	0.539
21	Angola	Bie	AGOr211	0.504
22	Angola	Moxico	AGOr212	0.508
23	Angola	Kuando Kubango	AGOr213	0.511
24	Angola	Namibe	AGOr214	0.587
25	Angola	Huila	AGOr215	0.52
26	Angola	Cunene	AGOr216	0.522
27	Angola	Lunda Sul	AGOr217	0.586
28	Angola	Bengo	AGOr218	0.584
29	Angola	Total	AGOt	0.591

Figure 3.3: Dataset of only useful HDI information

Figure 3.3 shows the `csv` file with only the useful information required. Therefore, I can now delete the original `subnatonalHDI.csv` file, as I won't be needing it anymore.

No.	Test	Input	Type	Expectation	Output
t_2	Written HDI Data to `csv`	–	–	`csv` (with correct fields) written correctly	As Expected

Table 3.4: Testing Table for Code Snippet 3.3

3.1.3 Part 2 – Collecting OSM Data for a Given Region

Next, I need to be able to retrieve the locations of all the objects from OpenStreetMaps, using Overpass Turbo.

`data_processing / get_osm_data.py`

```
import requests

# makes a request to overpass turbo to get osm data
def get_osm_data(object_type, bounding_coords):
    overpass_url = 'http://overpass-api.de/api/interpreter'
    overpass_query = [
        '(overpass turbo query)'
    ] # placeholder query
    overpass_query = '\n'.join(overpass_query) # concatenates list items
    ↪  into a multi-line string

    response = requests.get(overpass_url, params={'data': overpass_query})
    print(response) # log message, shows the HTTP response code
    data = response.json()
```

Code Snippet 3.4: Overpass Turbo `GET` Request

Here, I have written a function, `def get_osm_data`, which takes in 2 parameters (the type of object we are looking for, and the region we are looking in). We then must make a query to Overpass Turbo, which often comes in multiple lines. Therefore, I have decided to store each line of the query as a `str`, in a `list`, `overpass_query`, and the join the list with newline characters (`'\n'`).

We then make a `GET` request to the URL of Overpass Turbo, passing in the query to ensure we get the correct data. This can be easily done with the `requests` module's `get` function. Once we get a response (hopefully a `200: OK` response, as otherwise something is wrong at Overpass Turbo), we must get the data from it, which will be in a `json` format.

data_processing / get_osm_data.py / def get_osm_data

```
6     key_type = ('building' if object_type == 'house' else 'natural' if
      ↪  object_type == 'tree' else 'gambling' if object_type ==
      ↪  'slot_machines' else 'amenity') # get the right key type
7     region_poly = ' '.join([' '.join([str(latlong) for latlong in coord])
      ↪  for coord in bounding_coords]) # converts list of tuples to
      ↪  correct format for overpass turbo
8     overpass_query = [
9         '[out:json];',
10        '(',
11        f'   nwr["{key_type}"="{object_type}"](poly:"{region_poly}");',
12        ');',
13        'out center;'
14    ]
```

Code Snippet 3.5: Overpass Turbo Query

OSM can store data in 3 distinct ways: nodes, ways and relations. A node is a single coordinate, a way is a collection of nodes (which, for example, might represent the outline of a building) and a relation is a collection of nodes, ways or other relations.

The first line of the query `'[out:json];'` specifies that I want the output to be in a `json` format. Using the `'nwr'` command specifies that I want all **n**odes, **w**ays and **r**elations, as what we are looking for could be represented as any of them.

We then must specify that we want only the ways, nodes, and relations of a particular type. Each node/way/relation on OSM has a series of keys, each of which has a value (for example, `"amenity"="restaurant"` would specify that a particular node represents the amenity of a restaurant). The majority of the objects I am looking for are amenities, with a few exceptions. Therefore, I must first determine the correct key before defining the query (as shown on line 6).

We then also need to specify which region we want the ways, nodes and relations to come from. I didn't know how to do this, so I carried out some research on the Overpass Turbo documentation, and found that you can specify a polygon by separating each coordinate with a space (`"La1 Lo1 La2 Lo2 ..."`). This is a different format to how it is passed into the `def get_osm_data` function, so I must first join each of the coordinate pairs with spaces, (`[' '.join(str(latlong) for latlong in coord)]`), and then join each of those with spaces, again before the query is defined (as shown on line 7).

Finally, including `'out center;'` specifies that for each of the ways and relations, their centers should be outputted, instead of their bounding coordinates.

Figure 3.4: Output of Example Overpass Turbo Query

Executing this function, passing in `'school'` for `object_type` and a list of coordinates (in the `list[tuple[float]]` form) of a ring around my local area for `bounding_coords` and printing the what is stored in the variable `data` gives a list of schools in my local area (one of which I go to (and a bit of the next one) is shown in Figure 3.4).

No.	Test	Input	Type	Expectation	Output
$T_{1.1}$	Overpass Turbo Query	Closed Polygon,Valid object type	Valid	Returns the appropriate data	As Expected
$T_{1.2}$	Overpass Turbo Query	Open Polygon,Valid object type	Edge Case	Returns the appropriate data, as if there was a line joining the first and last vertices	As Expected
$T_{1.3}$	Overpass Turbo Query	Invalid object type	Invalid	Returns nothing	As Expected

Table 3.5: T_1 Testing Table (for Code Snippets 3.4 & 3.5)

However, again, there is lots of unnecessary data here (for example names, phone numbers etc.) so we must refine it.

`data_processing / get_osm_data.py / def get_osm_data`

```
21      coords = []
22      for nwr in data['elements']:
23          if nwr['type'] == 'node':
24              coords.append((nwr['lat'], nwr['lon']))
25          else: # it is a way or relation
26              coords.append((nwr['center']['lat'], nwr['center']['lon']))
27
28      return coords
```

Code Snippet 3.6: Getting a List of Latitude/Longitude Pairs

The objects we are recieving come in a list, which has the key `'elements'`, so I am iterating over `data['elements']`. For each one, we must append only the latitude longitude coordinates (in a `tuple`) to a `list`. If a particular one is stored as a node, then the latitude and longitude coordinates are stored directly, but if it is a way or relation, then they are stored in an additional `dict` with the key `'center'`.

Once we have iterated over each node/way/relation, we can return the `list` of coordinates.

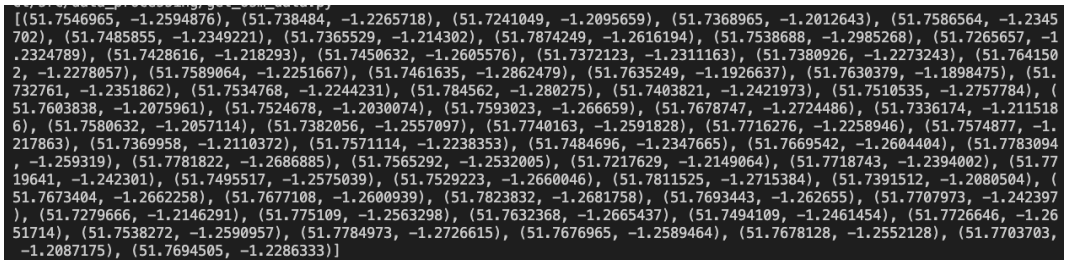


Figure 3.5: Output of a `list` of Latitude/Longitude pairs

Figure 3.5 shows the new output of the `def get_osm_data` function with the same input data from Figure 3.4, which is correct.

No.	Test	Input	Type	Expectation	Output
t_3	Refined OSM Data	–	–	Only Latiude/-Longitude pairs	As Expected

Table 3.6: Testing Table for Code Snippet 3.6

3.1.4 Part 3 – Implementing $A(x, y)$

Now, it's time to start calculating the factors from this data. Firstly, the average distance factors.

`data_processing / calc_factors.py`

```
import numpy as np

EARTH_RADIUS = 6371 # (in km)

# converts angle in degrees to angle in radians
def deg2rad(angle):
    return (np.pi/180)*angle

# finds the distance (in km) along the surface of the earth, between 2 coordinates
def distBetween2Points(p1, p2):
    La1 = deg2rad(p1[0])
    Lo1 = deg2rad(p1[1])
    La2 = deg2rad(p2[0])
    Lo2 = deg2rad(p2[1])
    return EARTH_RADIUS * np.arccos(np.sin(La1)*np.sin(La2) + np.cos(La1)*np.cos(La2)*np.cos(Lo1-Lo2))
```

Code Snippet 3.7: Finding the Distance between any 2 points

This code snippet implements equation 2.7, which finds the distance along the surface of the Earth, between 2 points (`p1` and `p2`), whose coordinares are given in degrees.

The first thing we need to do, is convert each of these angles into radians, by using the conversion factor (angle in radians) $= \frac{\pi}{180} \times$ (angle in degrees), and implemented with the function `def deg2rad`.

No.	Test	Input	Type	Expectation	Output
t_4	Conversion from degrees to radians	angle in degrees	–	$\frac{\pi}{180} \times$ (angle)	As Expected

Table 3.7: Testing Table for Code Snippet 3.7

We then must calculate the distance with equation 2.7, and return it. I am using the module `numpy` for the sin, cos and arccos functions, as well as a precise value for π.

For each of the following tests in T_2, the 2 points I selected to test were 30° away from each other, along a great circle. Therefore, the output for all of them should be $\frac{1}{12} \times 2\pi r$, which ≈ 3335.8 km.

No.	Test	Input	Type	Expectation	Output
$T_{2.1}$	Distance between 2 points	2 Points in the same hemisphere: $(30°, 0°)$, $(60°, 0°)$	Valid	3335.8 km	As Expected
$T_{2.2}$	Distance between 2 points	2 Points in opposite hemispheres: $(15°, 0°)$, $(-15°, 0°)$	Edge Case	3335.8 km	As Expected
$T_{2.3}$	Distance between 2 points	2 Points either side of the international dateline: $(0°, 165°)$, $(0°, -165°)$	Edge Case	3335.8 km	As Expected
$T_{2.4}$	Distance between 2 points	2 Points whose shortest distace crosses the North Pole: $(75°, 90°)$, $(75°, -90°)$	Edge Case	3335.8 km	As Expected

Table 3.8: T_2 Testing Table (for Code Snippet 3.7)

data_processing / calc_factors.py

```
2 import random
```

data_processing / calc_factors.py

```
18 # finds the average distance between 2 types of objects, A(x,y)
19 def averageDistance(x_objects, y_objects, max_x_objects=500):
20     if len(x_objects) == 0 or len(y_objects) == 0: # the average distance
       ↪  is undefined
21         return None
22     if len(x_objects) > max_x_objects: # random sample
23         x_objects = random.sample(x_objects, max_x_objects)
```

Code Snippet 3.8: $A(x, y)$ Validation

Here, I am defining the `def averageDistance` function, to calculate $A(x, y)$. It takes in `x_objects` (x), `y_objects` (y) and `max_x_objects` as parameters. By default (if nothing is passed in for `max_x_objects`), `max_x_objects` is set to 500, as it was in the flowchart in Figure 2.9.

If there are either no x-objects, or no y-objects, then the average distance between them is undefined, so we must return `None`. When this gets fed into the neural network, it will be replaced by an arbitrarily large number, as (in theory) larger values of $A(x, y)$ for any x, y are "bad", and should be decreased.

If there are instead more than (`max_x_objects`) x-objects, then we must take a random sample of (`max_x_objects`), as otherwise the function will take too long to execute. To do this, we must first `import random`, and then use its `random.sample()` method.

No.	Test	Input	Type	Expectation	Output
$T_{3.1}$	$A(x, y)$	Less than 500 x-objects	Valid	Uses all of the x-objects	As Expected
$T_{3.2}$	$A(x, y)$	500 x-objects	Boundary	Uses all 500 of the x-objects	As Expected
$T_{3.3}$	$A(x, y)$	More than 500 x-objects	Invalid	Uses a random 500 of the x-objects	As Expected
$T_{3.4}$	$A(x, y)$	0 x-objects	Invalid	Returns `None`	As Expected

Table 3.9: T_3 Testing Table (for Code Snippet 3.8)

`data_processing / calc_factors.py / def averageDistance`

```
24      # iterate over each x_object, and find the closest y_object
25      min_dists = []
26      for x_object in x_objects:
27          dist_to_ys = []
28          for y_object in y_objects:
29              dist_to_ys.append(distBetween2Points(x_object, y_object))
30          min_dists.append(min(dist_to_ys)) # the closest y_object
31      return np.mean(min_dists)
```

Code Snippet 3.9: Calculating $A(x, y)$

This code snippet implements the rest of the algorithm to find $A(x, y)$, as shown in Figure 2.9. (We must iterate over each x-object, find the closest y-object, and return the mean of those distances).

Running this function, passing in the list of schools shown in Figure 3.5, and a list of houses in the same area, returns 0.37311016450362056 km, which sounds about right. Running the function again, with the same inputs, but with no random sampling however returns 0.3586665816532994 km, which is the true value, but takes longer to calculate. Therefore, in the first run, the random sampling must have (randomly) selected particular houses that happened to be slightly further away from schools than usual (approximately 15 more metres).

No.	Test	Input	Type	Expectation	Output
t_5	$A(x, y)$	List of houses and schools in my local area	–	~ 0.36 km	As Expected

Table 3.10: Testing Table for Code Snippet 3.9

To see the full effect of the random sampling, we can run the function on the same data multiple times, to see the range of values it produces:

```
average distance from house to nearest school in oxford is: 0.3471860081253106 km
average distance from house to nearest school in oxford is: 0.34935882509741467 km
average distance from house to nearest school in oxford is: 0.3557128766643569 km
average distance from house to nearest school in oxford is: 0.35344981434591427 km
average distance from house to nearest school in oxford is: 0.35875781665549433 km
average distance from house to nearest school in oxford is: 0.3613735418982461 km
average distance from house to nearest school in oxford is: 0.3557637814311824 km
average distance from house to nearest school in oxford is: 0.35691715365803295 km
average distance from house to nearest school in oxford is: 0.34808368746306134 km
average distance from house to nearest school in oxford is: 0.3524024184457041 km
average distance from house to nearest school in oxford is: 0.364156322089742 km
average distance from house to nearest school in oxford is: 0.3528441770837829 km
average distance from house to nearest school in oxford is: 0.3636918565334129 km
average distance from house to nearest school in oxford is: 0.369403201575993 km
average distance from house to nearest school in oxford is: 0.362327943106565 km
average distance from house to nearest school in oxford is: 0.364705313921878 km
average distance from house to nearest school in oxford is: 0.3549901458177277 km
average distance from house to nearest school in oxford is: 0.3480712819732799 km
average distance from house to nearest school in oxford is: 0.3609290654924819 km
average distance from house to nearest school in oxford is: 0.3589591342932028 km
that took 11.004106998443604 seconds
```

Figure 3.6: $A(x, y)$ 20 times, with `max_x_objects` set to 500

Figure 3.6 shows executing $A(x, y)$ 20 times, with the same input data as before, and `max_x_objects` set to 500. The value it returns is often correct to 1 significant figure (0.4 km), and correct to 2 significant figures about half the time (0.36 km). In total, the whole thing took around 11 seconds and so each one took approximately 0.55 seconds.

This is not very accurate, and so I believe we should increase the `max_x_objects`. This will also increase the time for which it takes, so we shouldn't increase it by too much.

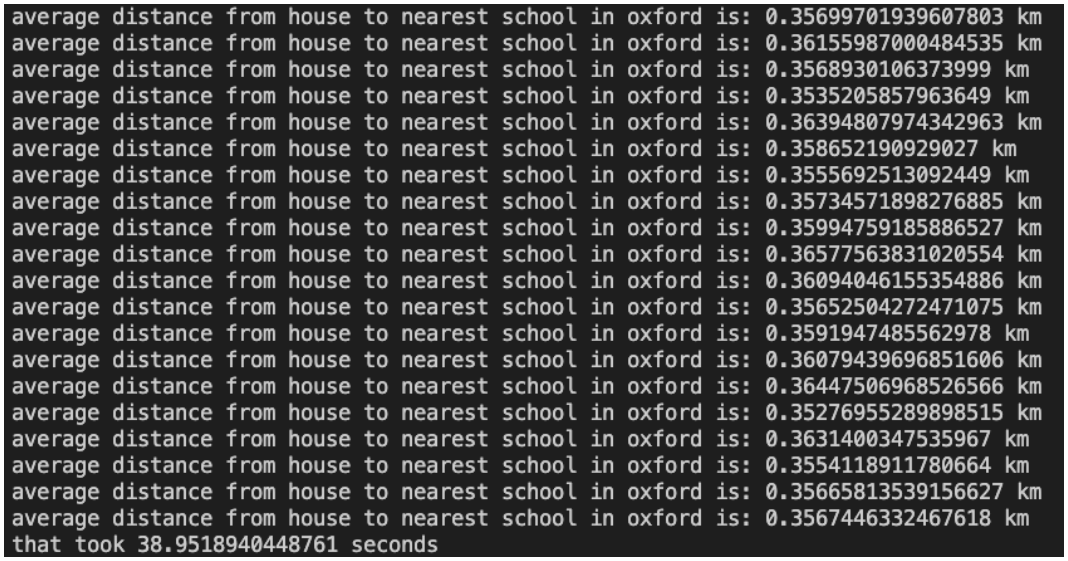


Figure 3.7: $A\,(x, y)$ 20 times, with `max_x_objects` set to 2000

Figure 3.7 shows executing $A\,(x, y)$ 20 times, with the same input data as before, and `max_x_objects` set to 2000. The value it now returns is always correct to 1 significant figure (0.4 km), and often correct to 2 significant figures (0.36 km). In total, the whole thing took around 39 seconds and so each one took approximately 1.95 seconds. This makes sense, as $A\,(x, y)$ should scale linearly with `max_x_objects`, and $0.55 \times \frac{2000}{500} \approx 1.95$.

This is a better result, in still a reasonable amount of time, and so I have decided to change the default value for `max_x_value` from 500 to 2000. The value can of course still be changed at any time, by passing in a different number. When the user comes to predict the HDI of their area in a later prototype, they may be able to change `max_x_objects` with a slider, to make a tradeoff between accuracy and time.

3.1.5 Part 4 – Implementing $D\,(x)$

Now its time for the harder factors to calculate, the density factors.

First, I decided to make the part of $D\,(x)$ that calculates the area of the region a seperate function, as the area will be the same for all density factors of a given area, and so it would be more efficient to calculate it once, beforehand.

data_processing / calc_factors.py

```
33 # calculates the area (in km^2) of a given region bound by a set of
   ↪ latitude/longitude coordinates
34 def calcArea(bounding_coords):
35     if bounding_coords[0] == bounding_coords[-1]: # if the first and last
       ↪ coordinates are the same, then delete the last one
36         bounding_coords.pop(-1)
37     # convert lat/lon pairs to 3d position vectors
38     bounding_vectors = []
39     for coord in bounding_coords:
40         La = coord[0]
41         Lo = coord[1]
42         bounding_vectors.append(np.matrix([
43             [np.cos(La)*np.cos(Lo)],
44             [np.cos(La)*np.sin(Lo)],
45             [np.sin(La)]
46         ]))
```

Code Snippet 3.10: Converting list of bounding coordinates into Column Vectors

Here, I am defining the `def calcArea` function, it takes in the `bounding_coords` as a parameter. The first thing we need to check is if the first and last elements of that list are the same. If they are, then remove the last one, as we don't want to be double-counting a particular vertex.

Next, we must find the absolute position of each coordinate in 3D space, by using equation 2.14 (remembering that r can be set to 1, as the radius of the Earth is irrelevant). As stated before, I am using the `np.matrix` data structure to represent column vectors and matrices.

No.	Test	Input	Type	Expectation	Output
t_6	Converting lat/lon to column vectors	A lat/lon coordinate, for example, $(45°, 45°)$	–	In this example, $\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{\sqrt{2}}\right)$	In this example, (0.276, 0.445, 0.851), to 3 significant figures

Table 3.11: Testing Table for Code Snippet 3.10

This is clearly the wrong output. I first thought that maybe equation 2.14 was wrong, but then I checked my working and couldn't find any mistakes. I then realised I hadn't converted from degrees to radians before calculating the vectors.

data_processing / calc_factors.py / def calcArea

```
40          La = deg2rad(coord[0])
41          Lo = deg2rad(coord[1])
```

Code Snippet 3.11: Coverting from Degrees to Radians

No.	Test	Input	Type	Expectation	Output
t_6	Converting lat/lon to column vectors	A lat/lon coordinate, for example, $(45°, 45°)$	–	In this example, $\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{\sqrt{2}}\right)$	In this example, (0.5, 0.5, 0.707), to 3 significant figures, As Expected

Table 3.12: Testing Table for Code Snippet 3.11

This now produces correct vectors. We now must iterate over each vertex, and find its anticlockwise angle.

data_processing / calc_factors.py / def calcArea

```
47      # iterate over each vertex and find its anticlockwise angle
48      for vertex_index, vertex in enumerate(bounding_vectors):
49          prev_vertex = bounding_vectors[vertex_index-1]
50          next_vertex = bounding_vectors[0 if vertex_index ==
            ↪  len(bounding_vectors)-1 else vertex_index+1]
51          La = deg2rad(bounding_coords[vertex_index][0])
52          Lo = deg2rad(bounding_coords[vertex_index][1])
53          # define the rotation and translation matrices
54          rotation_matrix_z = np.matrix([
55              [np.cos(-Lo), -np.sin(-Lo), 0],
56              [np.sin(-Lo), np.cos(-Lo), 0],
57              [0, 0, 1]
58          ])
59          rotation_matrix_y = np.matrix([
60              [np.cos(La-np.pi/2), 0, np.sin(La-np.pi/2)],
61              [0, 1, 0],
62              [-np.sin(La-np.pi/2), 0, np.cos(La-np.pi/2)]
63          ])
64          translation_vector = np.matrix([
65              [0],
66              [0],
67              [-1]
68          ])
```

Code Snippet 3.12: Defining Key Variables in calculating area

Firstly, we need to get the positions 2 vertices adjacent to the one we are considering. To get the previous one, we can just subtract 1 from the index of the current one, (as in the case that we are considering the first vertex, the index `-1` refers to the last element in a list, which is the previous one in the shape). To get the next vertex, the index will usually be 1 more than the current index, except for when we are considering the final vertex, as the next one will be the first vertex.

We then must find the latitude and longitude of the current vertex, so that we can rotate the plane tangent to it to be the xy-plane. Luckily, `bounding_coords` and `bounding_vectors` have the same indices, so we just need to pass the `bounding_coord` at the index `vertex_index` through the `def deg2rad` function to get what we need.

Finally, we must define the 3 matries used to transform the plane into the xy-plane, as shown in equations 2.15 to 2.17.

data_processing / calc_factors.py / def calcArea

```
69          # rotate and translate the plane tangent to the vertex such that
            ↪  it is the xy-plane
70          prev_vertex = np.matmul(rotation_matrix_y,
            ↪  np.matmul(rotation_matrix_z, prev_vertex)) +
            ↪  translation_vector
71          vertex = np.matmul(rotation_matrix_y, np.matmul(rotation_matrix_z,
            ↪  vertex)) + translation_vector
72          next_vertex = np.matmul(rotation_matrix_y,
            ↪  np.matmul(rotation_matrix_z, next_vertex)) +
            ↪  translation_vector
73          # project onto the xy-plane
74          prev_vertex[2][0] = 0
75          next_vertex[2][0] = 0
```

Code Snippet 3.13: Rotating the plane tangent to each vertex

Now we must transform each of these vertices, using the matrices we just defined. As stated before, the first is to perform a rotation about the z-axis, such that it is parrallel to the y-axis, the second is to perform a rotation about the y-axis such that it is parrallel to the x-axis (and hence a translation of the xy-plane), and the third is to translate it onto the xy-plane.

Then, to project the previous and next vertices onto the xy-plane, we can just set their z-coordinates to 0. This is at index `[2][0]`, as z is the third axis (second with 0-indexing) and the column vectors are stored as a matrix, so each item in the column vector is actually an array (hence the `[0]` index).

No.	Test	Input	Type	Expectation	Output
t_7	Getting the previous and next vertices	–	–	Works in the edge cases concerning the first and last vertices	As Expected
t_8	Transformation of vertices	–	–	Current vertex should be transformed to $(0, 0, 0)$	As Expected
t_9	Projection of other vertices	–	–	z-coordinate set to 0	As Expected

Table 3.13: Testing Table for Code Snippets 3.12 & 3.13

data_processing / calc_factors.py / def calcArea

```
48      anticlockwise_angles = []
```

data_processing / calc_factors.py / def calcArea

```
77          # calculate anticlockwise angle between the vectors
78          prev_to_current = vertex - prev_vertex
79          current_to_next = next_vertex - vertex
80          anticlockwise_angle =
            ↪  np.arctan2((prev_to_current.item(1,0)*current_to_next.item(2,0)
            ↪  - prev_to_current.item(2,0)*current_to_next.item(1,0)) -
            ↪  (prev_to_current.item(2,0)*current_to_next.item(0,0) -
            ↪  prev_to_current.item(0,0)*current_to_next.item(2,0)) +
            ↪  (prev_to_current.item(0,0)*current_to_next.item(1,0) -
            ↪  prev_to_current.item(1,0)*current_to_next.item(0,0)),
            ↪  prev_to_current.item(0,0)*current_to_next.item(0,0) +
            ↪  prev_to_current.item(1,0)*current_to_next.item(1,0) +
            ↪  prev_to_current.item(2,0)*current_to_next.item(2,0))
81          anticlockwise_angles.append(anticlockwise_angle)
```

Code Snippet 3.14: Finding the anticlockwise angles

Now we must find the anticlockwise angle between the vertices. First, we find the vector to get from the previous vertex to the current vertex, and the vector to get from the current vertex to the next vertex, and then use equation 2.18 to find the anticlockwise angle.

`np.arctan2` is the 2-argument form of the arctan function, giving a range of $-\pi$ to π, instead of the regular $-\frac{\pi}{2}$ to $\frac{\pi}{2}$.

As we will want to find the sum of these anticlockwise angles, we must first declare a `list` before the `for` loop (started in Code Snippet 3.12), and then append each anticlockwise angle to it.

No.	Test	Input	Type	Expectation	Output
t_{10}	Calculating the anticlockwise angle	–	–	Angle that you need to turn anticlockwise to be facing in the new direction	As Expected

Table 3.14: Testing Table for Code Snippet 3.14

data_processing / calc_factors.py / def calcArea

```
82      # find the interior angles from the anticlockwise angles
83      sum_anticlockwise_angles = sum(anticlockwise_angles)
84      if sum_anticlockwise_angles == 0:
85          return None
86      if sum_anticlockwise_angles < 0:
87          anticlockwise_angles = [-1*angle for angle in
            ↪  anticlockwise_angles]
88      interior_angles = [np.pi-angle for angle in anticlockwise_angles]
89      # return the area
90      num_edges = len(bounding_coords)
91      return (EARTH_RADIUS ** 2) * (sum(interior_angles) -
        ↪  np.pi*(num_edges-2))
```

Code Snippet 3.15: Calculating the Area

This code snippet implements the bottom half of the flowchart for $D(x)$, in Figure 2.15. We find the sum of the anticlockwise angles, if it is 0, then the shape must be invalid and so we `return None`.

If the sum is negative, then the shape must have been drawn the wrong way around, and so we must multiply each of the anticlockwise angles by -1, before continuing, which can be done with a list comprehension.

Then, to calculate the interior angles, we subtract each angle from π, again using a list comprehension. Finally, to calculate the area of the region we use equation 2.13. To find the number of edges of the shape, we just count the number of vertices, as they will always be the same thing.

No.	Test	Input	Type	Expectation	Output
$T_{4.1}$	$D(x)$	Region drawn Clockwise	Valid	Correct area calculated	As Expected
$T_{4.2}$	$D(x)$	Region drawn Anticlockwise	Valid	Correct area calculated, as if it were drawn clockwise	As Expected
$T_{4.3}$	$D(x)$	Region drawn in multiple loops of the Earth	Valid	Correct area calculated, as if all coordinates were in the regular range	As Expected
$T_{4.4}$	$D(x)$	Region that self-intersects	Invalid	`None`	Some large area

Table 3.15: T_4 Testing Table (for Code Snippet 3.15)

For the areas drawn clockwise and anticlockwise, I used the same area that I used to get the data about houses and schools in the $A(x, y)$ section, which was a polygon around Oxford. The areas they returned were about right, after researching the true area of Oxford. My values turned out greater as I was drawing *around* Oxford. The discrepancy between the clockwise and anticlockwise values is most likely due to floating point arithmetic and calculating things in different orders.

For the area drawn in multiple loops around the Earth, I again used the same area of Oxford, except some of the coordinates had multiples of 360 added to them. Again, the discrepancy between values can be explained by floating point binary.

```
area of oxford but it was drawn clockwise: 54.51510926217421 km^2
area of oxford but it was drawn anticlockwise: 54.51510919007252 km^2
area of oxford but in multiple loops around the earth: 54.515115535020925 km^2
some random polygon that self-intersects: 255032235.95489413
```

Figure 3.8: Area of Oxford, with an Invalid Self-Intersection

The area that self-intersects must also have not worked as a result of floating point binary. When inspecting the sum of the anticlockwise angles, I found that it was very close to 0, but not quite. Therefore, I must round the sum of the anticlockwise angles before seeing if it is 0.

`data_processing / calc_factors.py / def calcArea`

```
83      sum_anticlockwise_angles = round(sum(anticlockwise_angles), 5)
```

Code Snippet 3.16: Rounding the sum of anticlockwise angles

5 decimal places should be enough to determine if it is supposed to be 0, while not affecting anything that shouldn't be 0.

No.	Test	Input	Type	Expectation	Output
$T_{4.4}$	$D(x)$	Region that self-intersects	Invalid	`None`	As Expected

Table 3.16: Testing Table for Code Snippet 3.16

```
area of oxford but it was drawn clockwise: 54.51510926217421 km^2
area of oxford but it was drawn anticlockwise: 54.51510919007252 km^2
area of oxford but in multiple loops around the earth: 54.515115535020925 km^2
some random polygon that self-intersects: None
```

Figure 3.9: Area of Oxford, with a Valid Self-Intersection

Finally, all that is left to do is calculate $D(x)$.

`data_processing / calc_factors.py`

```
93  # finds the density of a type of object, D(x)
94  def density(x_objects, area):
95      return len(x_objects)/area
```

Code Snippet 3.17: Calculating $D(x)$

$D(x)$ is given by the number of x-objects per unit area, so we can just find the length of the `x_objects` list and divide by the area to get our answer.

No.	Test	Input	Type	Expectation	Output
t_{11}	$D(x)$	List of schools in Oxford, Area of Oxford	–	$\sim$ 1 School per km^2	As Expected

Table 3.17: Testing Table for Code Snippet 3.17

```
D(School) for oxford: 1.1923299958439 38 schools/km^2
```

Figure 3.10: D(School) for Oxford

3.1.6 Part 5 – Iterating over each Subnational Region

Now it is time to finally compile all of our training data. We first must be able to find all of the required factors for any given area.

data_processing / calc_factors.py

```
3  from get_osm_data import get_osm_data
```

data_processing / calc_factors.py

```
98  # finds all the factors for a given region
99  def getAllFactors(region):
100     # retrieve all relevant osm data
101     object_types = ['house', 'school', 'hospital', 'pharmacy',
        ↪  'restaurant', 'place_of_worship', 'bank', 'slot_machines',
        ↪  'fast_food', 'toilets', 'police', 'university', 'library',
        ↪  'post_box', 'vending_machine', 'bench', 'tree']
102     all_objects = {}
103     for object_type in object_types:
104         print(f'getting {object_type} data...') # log message
105         all_objects[object_type] = get_osm_data(object_type, region)
```

Code Snippet 3.18: Retrieving All OSM Data

To do this, I have defined a function, `def getAllFactors`, which takes in a region (in the `list[tuple[float]]` format), and returns a `dict` of all the factors. We must first retrieve all of the relevant data from OSM (using the `def get_osm_data` function implemented in Section 3.1.3).

Here, I define a list of all the object types needed for all of the factors, and an empty dictionary. For each of the object types, a key is created in the dictionary, with the value being the returned `list` of latitude and longitude coordinates

I am also printing a log message, as some of these fetches may take a long time, and so it is useful to keep track of where we are in the process

No.	Test	Input	Type	Expectation	Output
t_{12}	Collecting all objects	–	–	Each list saved in a `dict`	As Expected

Table 3.18: Testing Table for Code Snippet 3.18

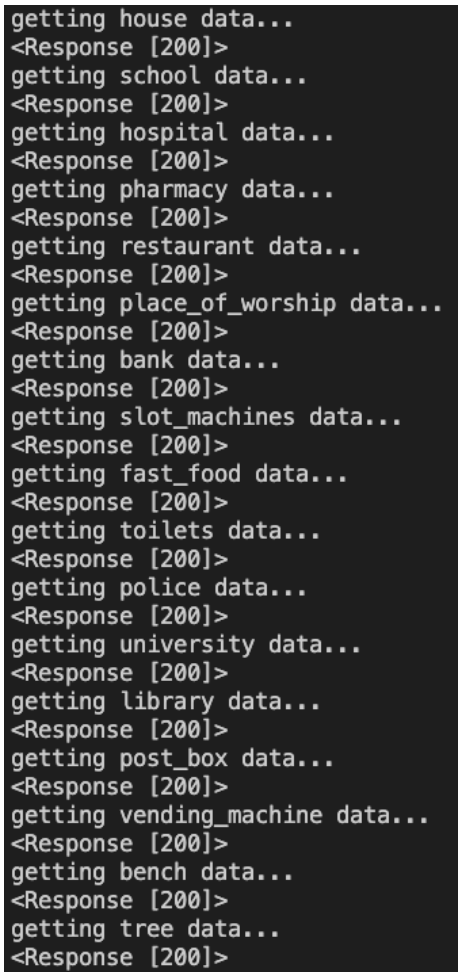


Figure 3.11: Log messages for Collecting data for all objects

Now we must calculate each factor with this information.

data_processing / calc_factors.py / def getAllFactors

```
106     # return a dictionary of all the factors
107     print('calculating area...') # log message
108     area = calcArea(region)
109     print('calculating factors...') # log message
110     return {
111         "A(House School)": averageDistance(all_objects['house'],
            ↪ all_objects['school']),
112         "A(House Hospital)": averageDistance(all_objects['house'],
            ↪ all_objects['hospital']),
113         "A(House Pharmacy)": averageDistance(all_objects['house'],
            ↪ all_objects['pharmacy']),
114         "A(House Restaurant)": averageDistance(all_objects['house'],
            ↪ all_objects['restaurant']),
115         "A(School Hospital)": averageDistance(all_objects['school'],
            ↪ all_objects['hospital']),
116         "A(Police Hospital)": averageDistance(all_objects['police'],
            ↪ all_objects['hospital']),
117         "A(House Place of Worship)": averageDistance(all_objects['house'],
            ↪ all_objects['place_of_worship']),
118         "A(Bank Slot Machine)": averageDistance(all_objects['bank'],
            ↪ all_objects['slot_machines']),
119         "A(Fast-Food Place Toilet)":
            ↪ averageDistance(all_objects['fast_food'],
            ↪ all_objects['toilets']),
120         "A(House Police)": averageDistance(all_objects['house'],
            ↪ all_objects['police']),
121         "A(University Library)":
            ↪ averageDistance(all_objects['university'],
            ↪ all_objects['library']),
122         "A(House Library)": averageDistance(all_objects['house'],
            ↪ all_objects['library']),
```

Code Snippet 3.19: Calculating All Average Distance Factors

Here, I am returning a `dict` witha key for each factor. For the average distance factors, we can just pass the relevant data into the `def averageDistance` function (implemented in Section 3.1.4).

The factor names don't have a comma between the 2 parameters of the A function, as that would mess up the `csv` headers.

data_processing / calc_factors.py / def getAllFactors

```
123         "D(School)": density(all_objects['school'], area),
124         "D(Hospital)": density(all_objects['hospital'], area),
125         "D(Pharmacy)": density(all_objects['pharmacy'], area),
126         "D(Police)": density(all_objects['police'], area),
127         "D(Library)": density(all_objects['library'], area),
128         "D(Toilet)": density(all_objects['toilets'], area),
129         "D(Restaurant)": density(all_objects['restaurant'], area),
130         "D(Place of Worship)": density(all_objects['place_of_worship'],
            ↪ area),
131         "D(Post Box)": density(all_objects['post_box'], area),
132         "D(Vending Machine)": density(all_objects['vending_machine'],
            ↪ area),
133         "D(Bench)": density(all_objects['bench'], area),
134         "D(Tree)": density(all_objects['tree'], area),
135     }
```

Code Snippet 3.20: Calculating All Density Factors

For the density factors, we can just pass the relevant data into the `def density` function (implemented in Section 3.1.5), along with the area of the region. The area of the region must therefore be calculated before, by using the `def calcArea` function (also implemented in Section 3.1.5).

No.	Test	Input	Type	Expectation	Output
t_{13}	Each factor is calculated	–	–	We have a `dict` of all the factors of a given region	As Expected

Table 3.19: Testing Table for Code Snippets 3.19 & 3.20

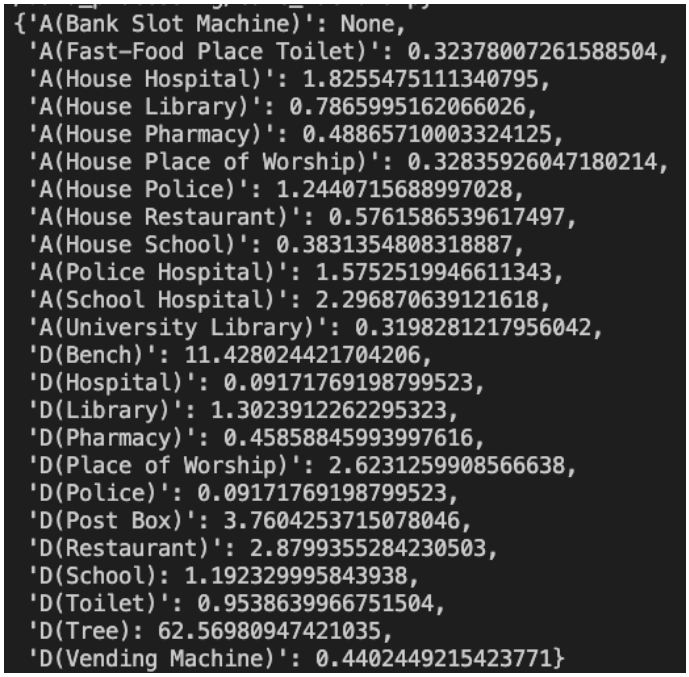


Figure 3.12: Example all factors `dict` for Oxford

Figure 3.12 shows the output of the `def getAllFactors` function for the same region I have been testing in previously (a rough ring around Oxford).

I now need a list of all of the subnational regions (that is, a `list` of bounding coordinates for each one). As stated before, I will also download this from the Global Data Lab [5].

The full dataset is a shapefile `.shp`, which I didn't know how to read. After researching the problem, I found a StackOverflow solution [8] on how to convert a shapefile into a `.json` file, which I do know how to read.

(This code has been adapted from this StackOverflow solution: [8])
`data_processing / process_shapefile.py`

```
import shapefile
import json

# read records of shapefile
reader = shapefile.Reader("data/shapefiles/GDL Shapefiles V6.3 large.shp")
fields = reader.fields[1:]
field_names = [field[0] for field in fields]
data = []
for shape_record in reader.shapeRecords():
    print(shape_record.record) # log message
    data.append({
        "type": "Feature",
        "geometry": shape_record.shape.__geo_interface__,
        "properties": dict(zip(field_names, shape_record.record))
    })

# write data to file
print('writing to file...') # log message
with open("data/region_coords.json", "w") as file:
    json.dump(data, file)
print('done!') # log message
```

Code Snippet 3.21: Processing the Shapefile

Code Snippet 3.21 shows the code from the StackOverflow solution [8], with some slight modifications.

First, we must `import shapefile` and `import json`, to be able to handle the 2 different types of files. It then looks like we define a `reader`, similar to what we did with the `.csv` file.

It appears that shapefiles are similarly structured to databases, as we have a set of fields, and records containing values for each field. To turn this into a `.json` file, we can iterate over each record, create a dictionary out of it, and append it to a `list`. Once we have a full list, we can write it to a `.json` file.

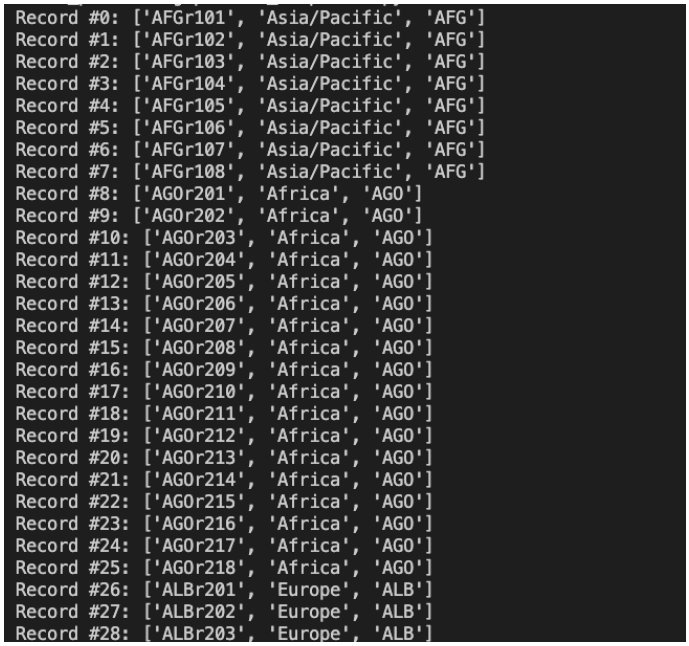


Figure 3.13: Each record of the shapefile

Figure 3.13 shows each record being read from the shapefile. The list of coordinates does not appear to be here, so that must be what is stored in the `shape_record.shape.__geo_interface__` attribute.

No.	Test	Input	Type	Expectation	Output
t_{14}	Process Shapefile	–	–	Readable `.json` file	As Expected

Table 3.20: Testing Table for Code Snippet 3.21

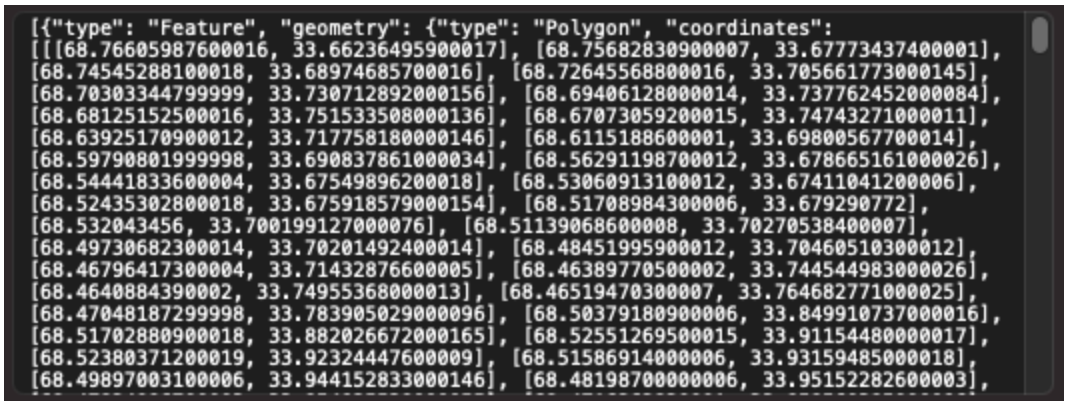


Figure 3.14: Sample of the resulting `.json` file

Now it is time to execute the `def getAllFactors` function on each of these regions, to make our training data.

`data_processing / compile_data.py`

```
import json
import csv

print('reading files...') # log message
# read json file
with open('data/region_coords.json', 'r') as file:
    regionData = json.load(file)

# read csv file
with open('data/hdi.csv', 'r') as file:
    reader = csv.DictReader(file)
    hdiData = []
    for record in reader:
        hdiData.append(record)
```

Code Snippet 3.22: Reading the Region & HDI files

Here, I am reading the 2 files I need, to get the region data and the HDI data. We must first `import json` and `import csv`, as those are the 2 types of file we are dealing with. Reading these files is done in the same way as before, by defining a `csv.DictReader` of the file.

No.	Test	Input	Type	Expectation	Output
t_{15}	Read `csv` and `json` file	–	–	Data stored to variables	As Expected

Table 3.21: Testing Table for Code Snippet 3.22

`data_processing / compile_data.py`

```
16 SKIP_THE_FIRST = 0
17
18 # iterate over each region
19 for region_number, region in enumerate(regionData):
20     if region_number < SKIP_THE_FIRST:
21         continue
22     print(f'Processing {region["properties"]["gdlcode"]}...') # log
       ↪  message
```

Code Snippet 3.23: Iterating over each region

This code will iterate over each `region`, contained within the `region_coords.json` file, by making use of a `for` loop. Carrying this out is likely to take an extremely long time, and so I might want to pause the process, or if it crashes for some reason, I won't want to start from the beginning. Therefore, I have defined a variable, `SKIP_THE_FIRST` n, which will tell the program to not calculate the first n regions. We can implement this by using the `enumerate` function, which will allow for us to keep count, and the `continue` key word, which will skip the remaining code of the `for` loop, and continue to the next iteration.

No.	Test	Input	Type	Expectation	Output
t_{16}	Skipping the first n regions	n (3, 5, 0 etc)	–	Starts on region $n+1$	As Expected

Table 3.22: Testing Table for Code Snippet 3.23

I then noticed that from Figure 3.14, the regions seem to be in a different format to what I expected, as they were in `list[list[coordinate]]`, where `coordinate` is a `list[float]`, where I had thought it would have been just a `list[coordinate]`. This must be due to the fact that a region may contain different polygons, which, for example, may represent different islands or exclaves. To store each of these, there must be an extra `list`.

If the region contains multiple polygons (exclaves), it is often the fact that there is 1 main one, and a number of tiny ones that can be considered insignificant. Therefore, I must find which of these `list[coordinate]`s is the main one, in order to use that as my region.

data_processing / compile_data.py

```
23      # find the main bit of the region (if it has multiple)
24      shapes = region['geometry']['coordinates']
25      lengths = [len(polygon) for polygon in shapes]
26      mainShape = shapes[lengths.index(max(lengths))]
27      mainShape = [coordinate[::-1] for coordinate in mainShape]
```

Code Snippet 3.24: Finding the Main Area

Here, I am getting the lengths of each region, finding the maximum of those lengths, and using the shape at that index (of the maximum length).

After further inspection on Figure 3.14, I also noticed that in each coordinate, the longitude was at index `0`, and the latitude was at index `1`, which is the opposite to how everything else in this program works. Therefore, we must reverse each coordinate, to swap the indices of the latitude and longitude, such that the latitude is first.

No.	Test	Input	Type	Expectation	Output
t_{17}	Finding the main shape	First 5 regions	–	Shape with most vertices is selected	As Expected
t_{18}	Reversing Coordinates	–	–	Latitude is first	As Expected

Table 3.23: Testing Table for Code Snippet 3.24

data_processing / compile_data.py

```
1 from calc_factors import getAllFactors
```

data_processing / compile_data.py

```
17 field_names = ['country', 'region', 'code', 'hdi', 'A(House School)',
    ↪ 'A(House Hospital)', 'A(House Pharmacy)', 'A(House Restaurant)',
    ↪ 'A(School Hospital)', 'A(Police Hospital)', 'A(House Place of
    ↪ Worship)', 'A(Bank Slot Machine)', 'A(Fast-Food Place Toilet)',
    ↪ 'A(House Police)', 'A(University Library)', 'A(House Library)',
    ↪ 'D(School)', 'D(Hospital)', 'D(Pharmacy)', 'D(Police)', 'D(Library)',
    ↪ 'D(Toilet)', 'D(Restaurant)', 'D(Place of Worship)', 'D(Post Box)',
    ↪ 'D(Vending Machine)', 'D(Bench)', 'D(Tree)']
```

data_processing / compile_data.py

```
30     # find all the factors
31     allFactors = getAllFactors(mainShape)
32     # match it with the hdi
33     matchingHDI = [record for record in hdiData if record['code'] ==
        ↪ region['properties']['gdlcode']]
34     if len(matchingHDI) != 0:
35         newRecord = matchingHDI[0]
36         newRecord.update(allFactors)
37         print('writing to file...') # log message
38         with open('data/training_data.csv', 'a') as file:
39             writer = csv.DictWriter(file, fieldnames=field_names)
40             writer.writerow(newRecord)
```

Code Snippet 3.25: Writing training data to `csv`

Now we can calculate each of the factors for this region, by passing it into the `def getAllFactors` function. Before we do this however, we must `import` it `from calc_factors`, as it is in a different file (`calc_factors.py`).

We then need to find the "true" value for the HDI for this region, which will be in the `.csv` file made in Section 3.1.2. For each of the records in that `.csv` file, we must check if the `code` is the same as the `gdlcode` stored in this region. The number of records in this file and the number of regions are not the same, and so there may be some double-counting, or missing codes. Therefore, we want to get the first instance of the matching HDI (at index `0`), if there is at least 1 record for which the codes match.

`newRecord` is then a dictionary, representing the record containing the matching HDI. We then call the `update` method, passing in `allFactors`, which will merge the `newRecord` and `allFactors` dictionaries, which will represent a new record in the training data `.csv`

file.

Finally, we write to the `.csv` file, in the same way as before, by defining a `csv.DictWriter`, specifying the field names with a `list` of field names, and writing a row. We must `open` the file in `'a'` mode (append), rather than `'w'` mode, as we don't want to erase all previous records each time we write a new one.

No.	Test	Input	Type	Expectation	Output
t_{19}	Writing Full Record	The AFGr101 region	–	HDI and all factors written to `.csv`	HTTP `414: URI Too Long` error

Table 3.24: Testing Table for Code Snippet 3.25

```
Processing AFGr101...
getting house data...
<Response [414]>
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/requests/models.py", line 974, in json
    return complexjson.loads(self.text, **kwargs)
  File "/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/json/__init__.py", line 346, in loads
    return _default_decoder.decode(s)
  File "/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/json/decoder.py", line 337, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
  File "/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/json/decoder.py", line 355, in raw_decode
    raise JSONDecodeError("Expecting value", s, err.value) from None
json.decoder.JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

Figure 3.15: HTTP `414` Error

Unfortunately, this did not work and threw a HTTP `414: URI Too Long` error, when trying to get the house data. As a result of this, the `GET` request returned an empty string, and so the rest of the error message is a `JSONDecodeError`, trying to decode an empty string into `json`.

After researching the problem, I found that `GET` requests have a maximum length, and going over it is considered "abusing `GET`". I then went to the Overpass Turbo frontend, to test if my query works there, which it does. This lead me to believe that they aren't using a `GET` request.

To find out what type of request they were using, I used the Safari Developer Tools, to see all of the incomming and outgoing traffic in the browser, including all HTTP requests.

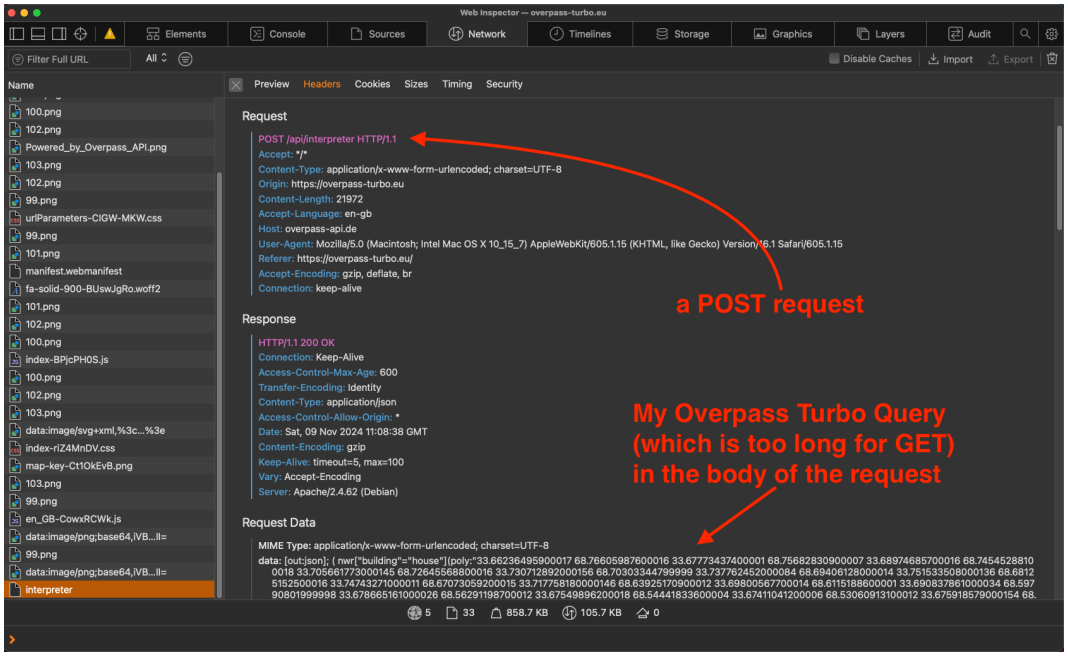


Figure 3.16: Overpass Turbo's `POST` request

It turns out that they are using a `POST` request, and sending the Overpass Turbo query in the body of the request, instead of using a `GET` request, and sending the query in the header.

`data_processing / get_osm_data.py / def get_osm_data`

```
17      response = requests.post(overpass_url, data=overpass_query)
```

Code Snippet 3.26: Using a `POST` request instead of a `GET` request

This code snippet replaces the original request, and implements the same thing that Overpass Turbo are doing on their frontend (using a `POST` request and sending the query in the body of the request).

No.	Test	Input	Type	Expectation	Output
t_{19}	Writing Full Record	The AFGr101 region	–	HDI and all factors written to `.csv`	As Expected

Table 3.25: Testing Table for Code Snippet 3.26

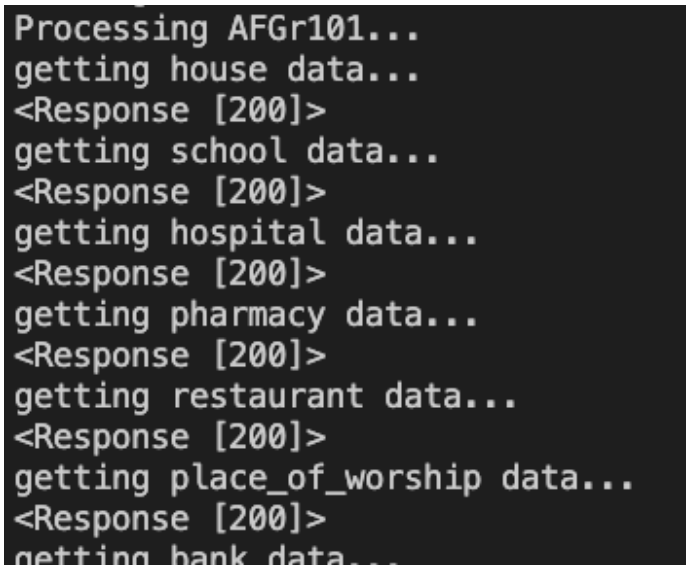


Figure 3.17: All HTTP Responses now `200: OK`

Unfortunately, by the time it reached the 8th region (AFGr108), it came up with another error. This time, a HTTP `400: Bad Request` error, which suggest that the format of my query is incorrect.

After looking into the problem, I found that the line to convert the `list[coordinate]` (where `coordinate` is a `list[float]` or `tuple[float]`) into a `str` (which Overpass Turbo can read as a polygon) had not worked correctly, as each coordinate was still in a `list`, rather than 2 numbers separated by a space. I then found that this region was stored as a `list[list[list[coordinate]]]`, rather than a `list[list[coordinate]]`.

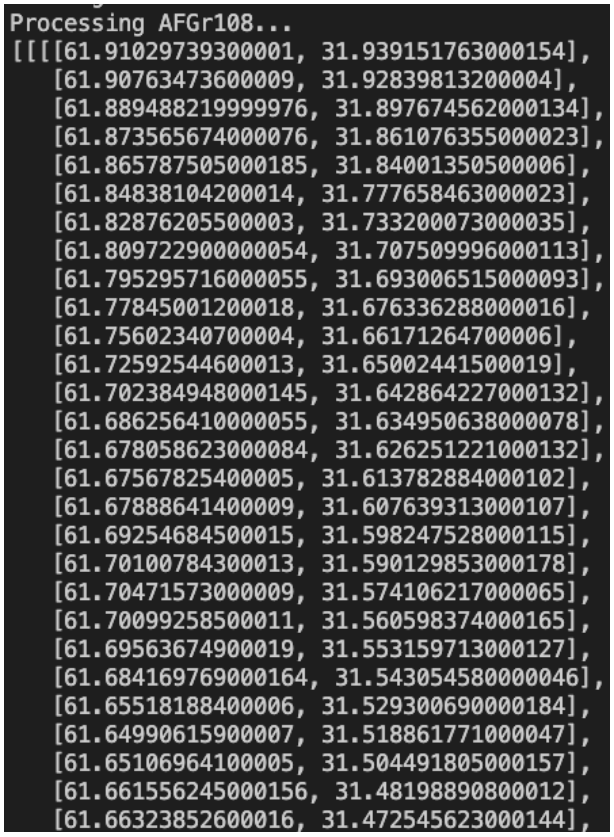


Figure 3.18: `list[list[list[coordinate]]]` debug print statement

Therefore, my initial assumption on how the regions must be stored is incorrect. After looking into the `.geojson` documentation, I found that coordinates can be stored in 1 of 4 ways:

1. `'Coordinate'` (as `coordinate`): A point on the Earth's Surface.
2. `'LineSegment'` (as `list[coordinate]`): A set of `'Coordinate'`s, joined together.
3. `'Polygon'` (as `list[list[coordinate]]`): A set of `'LineSegment'`s, to create a polygon.
4. `'MultiPolygon'` (as `list[list[list[coordinate]]]`): A collection of `'Polygon'`s.

where `coordinate` is a `list[float]`. This means that regions AFGr101 to AFGr107 were of type `'Polygon'`, but region AFGr108 is of type `'MultiPolygon'`.

In my initial assumption, I had not accounted for the existance of the `'LineSegment'`, which is why I was off by 1 dimension of `list`s.

data_processing / compile_data.py

```
25      # find the main bit of the region (if it has multiple)
26      shapes = region['geometry']['coordinates']
27      if region['geometry']['type'] == 'Polygon':
28          mainShape = shapes[0]
29      else:
30          lengths = [len(polygon[0]) for polygon in shapes]
31          mainShape = shapes[lengths.index(max(lengths))][0]
32      mainShape = [coordinate[::-1] for coordinate in mainShape]
```

Code Snippet 3.27: Correctly finding the Main Area

Each region must either be a `'Polygon'` or `'MultiPolygon'`, with each `'Polygon'` being 1 `'LineSegment'`, as nothing else makes any sense.

If the region is a `'Polygon'`, then it will just be a single `'LineSegment'`, and so we can find it at the index `0`.

Otherwise, it must be a `'MultiPolygon'`, and so this is where we iterate over the lengths and find the main shape. We must also use the `0` index here, to retrive the first (and only) `'LineSegment'` from the `'Polygon'`.

Afterwards, we must of course still reverse each coordinate, as they are still the wrong way around.

No.	Test	Input	Type	Expectation	Output
t_{19}	Writing Full Record	The AFGr108 region	–	HDI and all factors written to `.csv`	As Expected

Table 3.26: Testing Table for Code Snippet 3.27

After doing this, I found that it was easier to specify the most recent region, instead of the number of regions done, when not starting from the beginning:

data_processing / compile_data.py

```
17 MOST_RECENT_REGION = 'AFGr108'
```

data_processing / compile_data.py

```
23     if not started:
24         if region['properties']['gdlcode'] == MOST_RECENT_REGION:
25             started = True
26         continue
```

Code Snippet 3.28: Specifying the most recent region

No.	Test	Input	Type	Expectation	Output
t_{16}	Skipping the first n regions	Most recent region	–	Starts on the next region	As Expected

Table 3.27: Testing Table for Code Snippet 3.28

Everything was downloading successfully until I got to Fiji, where it threw another HTTP 400 error, which meant there was something wrong with my prompt again.

When pasting it into the Overpass Turbo frontend, it came back with an error, saying that I could not have coordinates whose longitude is greater that 180°, or less than −180°. This makes sense as to why this error only came about when processing Fiji, as it almost lies on the international dateline.

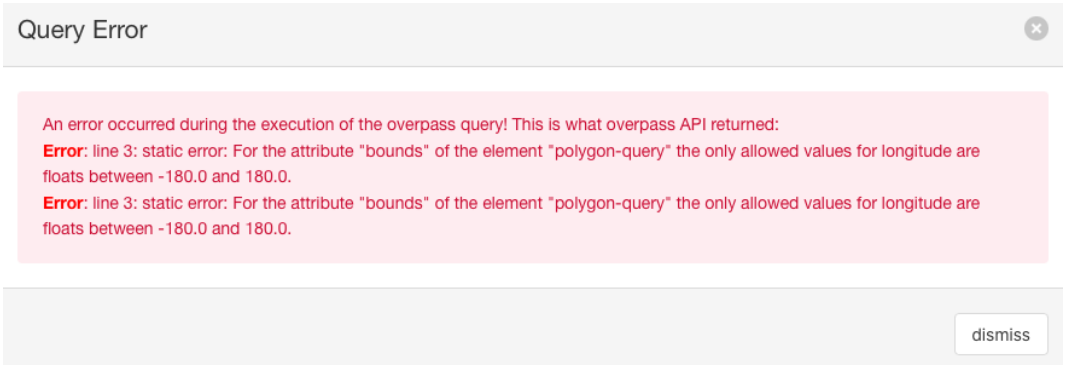


Figure 3.19: Longitude Error Message

To fix this, we can add 180° to it, and then mod 360°, and then subtract 180° again, such that $-180° \leq Lo \leq 180°$.

data_processing / get_osm_data.py / def get_osm_data

```
7       bounding_coords = [(latitude, (longitude + 180) % 360 - 180) for
        ↪  latitude, longitude in bounding_coords]
```

Code Snippet 3.29: Finding the right Longitude

No.	Test	Input	Type	Expectation	Output
t_{20}	Finding the right longitude	Longitude not inbetween $-180°$ and $180°$	–	Equivalent logitude inbetween $-180°$ and $180°$	As Expected

Table 3.28: Testing Table for Code Snippet 3.29

Again, everything was downloading successfully, until this time Scotland caused a problem, throwing yet another HTTP 400 error.

When I went to copy and paste the prompt into Overpass Turbo, in hopes of getting a similar error message, I quickly discovered that this prompt was much larger than the others, and had trouble copy and pasting it, as it was approximately 50 megabytes of text.

This was because Scotland was stored as a polygon with more than 500,000 vertices, most likely due to its irregular shape and jagged coastline, and so Overpass Turbo must have not been able to handle such a massive prompt.

Therefore, I decided to truncate regions with more than 100,000 vertices, by removing every other vertex until there were less than 100,000.

data_processing / get_osm_data.py / def get_osm_data

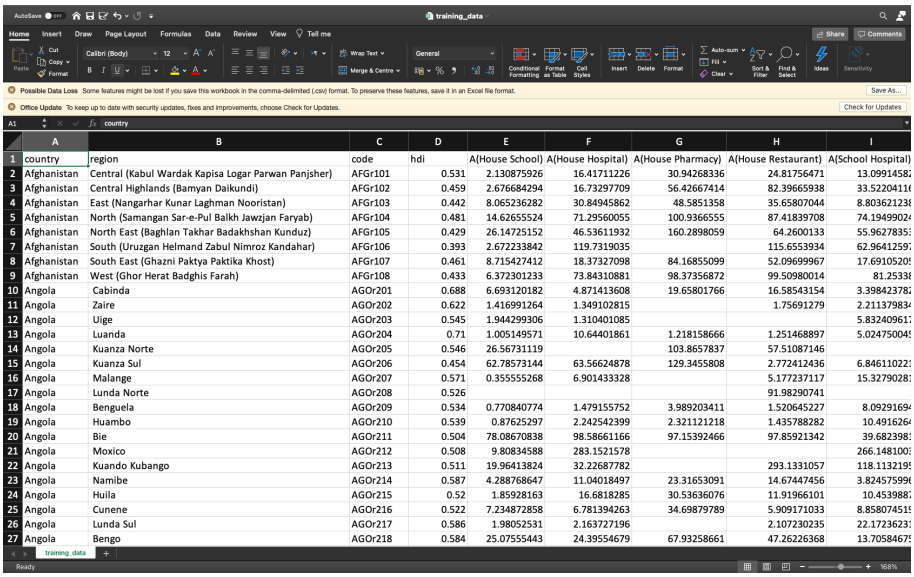
```
7       while len(bounding_coords) >= 100000:
8           print(f'truncating region with {len(bounding_coords)}
            ↪  vertices...') # log message
9           bounding_coords = [bounding_coord for n, bounding_coord in
            ↪  enumerate(bounding_coords) if n % 2 == 0]
```

Code Snippet 3.30: Truncating Detailed Regions

No.	Test	Input	Type	Expectation	Output
t_{21}	Truncating Detailed Regions	Region with more than 100,000 vertices	–	Less detailed region of the same area	As Expected

Table 3.29: Testing Table for Code Snippet 3.30

The rest of the process went smoothly, with no other errors, except for the occasional time out.



	A	B	C	D	E	F	G	H	I
1	country	region	code	hdi	A(House School)	A(House Hospital)	A(House Pharmacy)	A(House Restaurant)	A(School Hospital)
2	Afghanistan	Central (Kabul Wardak Kapisa Logar Parwan Panjsher)	AFGr101	0.531	2.130875926	16.41711226	30.94268336	24.81756471	13.0991458
3	Afghanistan	Central Highlands (Bamyan Daikundi)	AFGr102	0.459	2.676684294	16.73297709	56.42667414	82.39665938	33.5220411
4	Afghanistan	East (Nangarhar Kunar Laghman Nooristan)	AFGr103	0.442	8.065236282	30.84945862	48.5851358	35.65807044	8.803621238
5	Afghanistan	North (Samangan Sar-e-Pul Balkh Jawzjan Faryab)	AFGr104	0.481	14.62655524	71.29560055	100.9366555	87.41839708	74.1949902
6	Afghanistan	North East (Baghlan Takhar Badakhshan Kunduz)	AFGr105	0.429	26.14725152	46.53611932	160.2898059	64.2600133	55.9627835
7	Afghanistan	South (Uruzgan Helmand Zabul Nimroz Kandahar)	AFGr106	0.393	2.672233842	119.7319035		115.6553934	62.9641259
8	Afghanistan	South East (Ghazni Paktya Paktika Khost)	AFGr107	0.461	8.715427412	18.37327098	84.16855099	52.09699967	17.6910520
9	Afghanistan	West (Ghor Herat Badghis Farah)	AFGr108	0.433	6.372301233	73.84310881	98.37356872	99.50980014	81.2533
10	Angola	Cabinda	AGOr201	0.688	6.693120182	4.871413608	19.65801766	16.58543154	3.398423787
11	Angola	Zaire	AGOr202	0.622	1.416991264	1.349102815		1.75691279	2.211379834
12	Angola	Uige	AGOr203	0.545	1.944299306	1.310401085			5.832409617
13	Angola	Luanda	AGOr204	0.71	1.005149571	10.64401861	1.218158666	1.251468897	5.024750045
14	Angola	Kuanza Norte	AGOr205	0.546	26.56731119		103.8657837	57.51087146	
15	Angola	Kuanza Sul	AGOr206	0.454	62.78573144	63.56624878	129.3455808	2.772412436	6.846110223
16	Angola	Malange	AGOr207	0.571	0.355555268	6.901433328		5.177237117	15.3279028
17	Angola	Lunda Norte	AGOr208	0.526				91.98290741	
18	Angola	Benguela	AGOr209	0.534	0.770840774	1.479155752	3.989203411	1.520645227	8.092916
19	Angola	Huambo	AGOr210	0.539	0.87625297	2.242542399	2.321121218	1.435788282	10.491626
20	Angola	Bie	AGOr211	0.504	78.08670838	98.58661166	97.15392466	97.85921342	39.682398
21	Angola	Moxico	AGOr212	0.508	9.80834588	283.1521578			266.148100
22	Angola	Kuando Kubango	AGOr213	0.511	19.96413824	32.22687782		293.1331057	118.113219
23	Angola	Namibe	AGOr214	0.587	4.288768647	11.04018497	23.31653091	14.67447456	3.824575996
24	Angola	Huila	AGOr215	0.52	1.85928163	16.6818285	30.53636076	11.91966101	10.453988
25	Angola	Cunene	AGOr216	0.522	7.234872858	6.781394263	34.69879789	5.909171033	8.858074515
26	Angola	Lunda Sul	AGOr217	0.586	1.98052531	2.163727196		2.107230235	22.17236233
27	Angola	Bengo	AGOr218	0.584	25.07555443	24.39554679	67.93258661	47.26226368	13.70584675

Figure 3.20: Final Training Data

Figure 3.20 shows the final `.csv` file, containing the training data for the neural network, which is what I had hoped to have achieved by the end of this milestone.

I can now therefore delete `region_coords.json` and `hdi.csv`, as I only needed them to create `training_data.csv`.

3.1.7 Review

No.	Test	Inputs	Pass/Fail
T_1	Overpass Turbo Query	Closed Polygon, Open Polygon, Valid object type, Invalid object type	Pass
T_2	Formula for distance between 2 points	2 Points in the same hemisphere, 2 Points in opposite hemispheres, 2 Points either side of the international date line, 2 Points whose shortest distance crosses the North Pole	Pass
T_3	$A(x, y)$	Less than 500 x-objects, 500 x-objects, More than 500 x-objects, 0 x-objects	Pass

T_4	$D(x)$	Region drawn Clockwise, Region drawn Anticlockwise, Region drawn in mulitple loops of the Earth, Region that self-intersects	Pass

Table 3.30: Passed Testing Data for Milestone 1

After some successful debugging, all 4 of the main tests in Milestone 1 passed. As a result of that, the following success criteria has been met:

No.	**Success Criterion**	**Justification**	**Success**
S_1	Find suitable HDI training data for the Neural Network	The neural network must train on pre-calculated HDI values for certain areas, along with their respective density and average distance factors.	✓
S_2	Retrieve OSM data for a given area using Overpass Turbo	This will allow me to calculate useful geographical factors to predict the HDI from instead of more human ones (which are harder to determine).	✓
S_3	Calculate average distance factors from a given area	The average distance from *some object* to *some other object* is also a good measure to consider. For example $A(\text{House}, \text{School})$ is a useful factor as school children shouldn't have to travel long distances for education.	✓
S_4	Calculate density factors from a given area	The number of *some object* per unit area is a good way of measuring how many of that object are in a particular area while removing the bias of larger areas.	✓

Table 3.31: Achieved Success Criteria for Milestone 1

I also have a full set of training data, so overall, I believe this milestone has been a success.

3.2 Milestone 2 – Multilayer Perceptron

3.2.1 Success Criteria

By the end of this milestone, I hope to have a fully trained neural network, which is able of predicting HDI from a set of factors.

No.	Success Criterion	Justification
S_5	Successfully Implement the Feedforward Algorithm	This algorithm will allow us to make predictions on the HDI given a set of factors.
S_6	Successfully Implement the Backpropogation Algorithm	This algorithm will allow us to train the neural network base on a set of training examples.
S_7	Successfully train the Neural Network	In order to accurately predict the HDI of a given area, the neural network must train on existing examples to see what makes a high HDI.

Table 3.32: Success Criteria for Milestone 2

To determine if I have met this success criteria, I will be using the following test:

No.	Test	Inputs	Expected Output/Justification
T_5	Neural Network works	Using a *different* data set, e.g. handwritten digits, which certainly has enough training data	This is to ensure that the neural network works, and if it doesn't work for predicting HDI, it is as a result of a lack of training data.

Table 3.33: Testing Data for Milestone 2

Again, determining if the neural network is actually working is very difficult, so I indtend to train a neural network to do something else, like recognise handwritten digits, which certainly has enough training data, to determine if it is working in principle.

3.2.2 Part 6 – Implementing the Neural Network Structure

We must first define the constructor method for a Multilayer Pereptron.

neural_network.py

```
1  import numpy as np
2
3  class MultilayerPerceptron(object):
4      # constructor method
5      def __init__(self, layer_sizes):
6          # initialise layer sizes constant
7          self.layer_sizes = layer_sizes
8          self.L = len(layer_sizes)-1
```

Code Snippet 3.31: Defining the Constructor Method

Here, I am importing `numpy`, as it will be very useful in all the calculations we will be doing in this class.

Next, we define the `class MultilayerPerceptron`, which inherits from the generic python `object`. As stated before, we must only pass the `layer_sizes` into the constructor method (as well as `self`), as everything else can be defined implicitly from that.

The `layer_sizes` attribute will obviously be set to the `layer_sizes` parameter, and the `L` attribute is the number of layers, not including the input layer, so we subtract 1 from the length of the `layer_sizes list`.

neural_network.py / class MultilayerPerceptron / def __init__

```
9          # initialise matrices for the different quantities
10         self.activations = [np.matrix(np.zeros(shape=(layer_size,1))) for
           ↪  layer_size in layer_sizes]
11         self.weights =
           ↪  [np.matrix(np.random.randn(layer_sizes[index+1],layer_size))
           ↪  for index, layer_size in enumerate(layer_sizes[:-1])]
12         self.biases = [np.matrix(np.random.randn(layer_size,1)) for
           ↪  layer_size in layer_sizes]
13         self.z = [np.matrix(np.zeros(shape=(layer_size,1))) for layer_size
           ↪  in layer_sizes]
14         self.errors = [np.matrix(np.zeros(shape=(layer_size,1))) for
           ↪  layer_size in layer_sizes]
```

Code Snippet 3.32: Initialising the Matrices

Next, I added the attributes of all the `list`s of matrices, and column vectors, representing the activations, weights, biases, z-values and errors within the network.

The activations of the neurons are stored as a `list` of column vectors, where the size of each column vector is that layer's size. The activations will initially be set to 0, so we can use the `np.zeros` function to get a multidimensional array of 0s. The size of multidimensional

array we want is `layer_size` rows and 1 column, for some `layer_size`. To turn this into a column vector, we pass this into the `np.matrix` function, as done before. To get the full `list`, we must iterate this process for each of the `layer_sizes` in `layer_sizes`. The `z` and `errors` attributes are set to the same lists, as they are also quantities stored in neurons and should also be 0s to begin with.

Bias is also a quantity stored in the neurons, but should not be set to 0 to begin with, as it is a parameter of the system. Instead it should be set to a random number, which has been picked from a random distribution. To get a sensible range of values, I used the standard normal distribution (using `np.random.randn` in place of `np.zeros`):

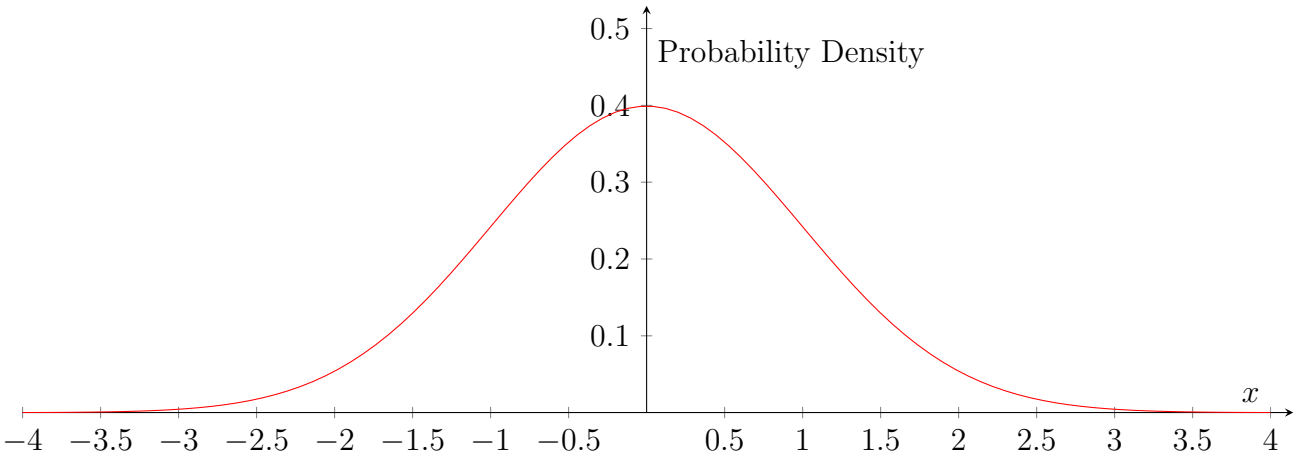


Figure 3.21: Graph of the Standard Normal Distribution

Figure 3.21 visually shows the normal distribution, and its bell-shaped curve. Most of the random values should therefore be in between -2 and 2, with very few being outside that range.

Weights are also paramaters of the system, and so should also get random values from the normal distribution. However, weights should be stored as a matrix between 2 layers, and so the shape must be different. Here, the number of rows must be the size of the next layer (`layer_sizes[index+1]`), and the number of columns must be the size of the current layer (`layer_size`). This time, instead of iterating over all of the layer sizes, we must only go up to 1 before the end, again due to the fact that weights go in between the layers.

```
activations
[matrix([[0.],
         [0.]]),
 matrix([[0.],
         [0.],
         [0.]]),
 matrix([[0.],
         [0.],
         [0.],
         [0.]])]
weights
[matrix([[ 0.00724576, -0.79401028],
         [-0.30978017,  0.20706976],
         [-0.0887502 ,  0.68012537]]),
 matrix([[-1.8620002 ,  0.23609408,  0.94124899],
         [-0.56005494, -0.11782026, -0.1842248 ],
         [-0.99255937, -0.75466923, -0.76560838],
         [ 1.13878057,  2.803111  ,  1.37908258]])]
biases
[matrix([[-1.63383171],
         [ 0.84018748]]),
 matrix([[ 0.94904602],
         [-0.7174148 ],
         [ 0.37237926]]),
 matrix([[-0.60068549],
         [-1.26667354],
         [-0.19706229],
         [ 1.89735122]])]
```

Figure 3.22: Printing activations, weights and biases

Figure 3.22 shows the output if we print `self.activations`, `self.weights` and `self.biases`, for a neural network with `layer_sizes = [2,3,4]` (`self.z` and `self.errors` will be the same as `self.activations`).

The final attributes we must define are the copies of the activations, z-values and errors used in training.

`neural_network.py`

```
2  import copy
```

`neural_network.py / class MultilayerPerceptron / def __init__`

```
16         # initialise lists for training states
17         self.training_activations = []
18         self.training_z = []
19         self.training_errors = []
```

Code Snippet 3.33: Initialising the Training Lists

Right now, they are set as empty lists, as we have not done any training yet.

In anticipation of making copies of the attributes, I have also imported the `copy` library.

No.	Test	Input	Type	Expectation	Output
t_{22}	Setting the activations `list`	(for example) Layer sizes = 2, 3, 4	–	A column vector of size 2, a column vector of size 3 and a column vector of size 4, filled with 0s	As Expected
t_{23}	Setting the weights `list`	(for example) Layer sizes = 2, 3, 4	–	A matrix of size 3×2 and a matrix of size 4×3, filled with random numbers from the normal distribution	As Expected
t_{24}	Setting the biases `list`	(for example) Layer sizes = 2, 3, 4	–	A column vector of size 2, a column vector of size 3 and a column vector of size 4, filled with random numbers from the normal distribution	As Expected
t_{25}	Setting the training `lists`	–	–	Empty `lists`	As Expected

Table 3.34: Testing Table for Code Snippets 3.31, 3.32 & 3.33

We must also define the sigmoid function, to use as our activation function.

`neural_network.py`

```
21 # activation function
22 def sigmoid(x):
23     return 1/(1+np.exp(-x))
```

Code Snippet 3.34: Defining the Sigmoid Function

This matches with the definition stated in equation 2.22. In the class diagram (Figure 2.29), I had made this function a method in the `class MultilayerPerceptron`, but now I believe that it makes more sense for it to be a stand-alone function, as it does not use any of the attributes from the class.

We must also define the derivative of the sigmoid function, as it will be used in the back-propogation algorithm. We can differentiate it using the quotient rule, and then rearrange it so that it is in terms of $\sigma(x)$:

$$\begin{aligned}
\sigma'(x) &= \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{1}{1+e^{-x}}\right) \\
&= \frac{(0)(1+e^{-x}) - (1)(-e^{-x})}{(1+e^{-x})^2} \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \left(\frac{e^{-x}}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) \\
&= \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) \\
&= \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) \\
&= (1-\sigma(x))(\sigma(x))
\end{aligned} \tag{3.1}$$

`neural_network.py`

```
25 # derivative of the activation function
26 def sigmoid_prime(x):
27     return sigmoid(x)*(1-sigmoid(x))
```

Code Snippet 3.35: Defining the derivative of the Sigmoid Function

Again, this was previously planned to be a method in the `class MultilayerPerceptron`, but I have since decided to move it out.

No.	Test	Input	Type	Expectation	Output
t_{26}	Sigmoid Function	Real numbers (for example, 0, 5, -2.73)	–	For those examples: 0.5, 0.9931, 0.0612	As Expected
t_{27}	Sigmoid Prime Function	Real numbers (for example, 0, -1, 1.78	–	For those examples: 0.25, 0.1966, 0.1235	As Expected

Table 3.35: Testing Table for Code Snippets 3.34 & 3.35

We must also create methods for saving and loading weights and biases.

neural_network.py

```
3  import pickle
```

neural_network.py / class MultilayerPerceptron

```
22      # saves weights and biases to an external file in the models folder
23      def save_model(self, filename):
24          parameters = {
25              'weights': self.weights,
26              'biases': self.biases
27          }
28          with open(f'models/{filename}.pkl', 'wb') as file:
29              pickle.dump(parameters, file)
```

Code Snippet 3.36: Saving weights and biases

As stated before, we can save the weights and biases attributes to a python `dict`, and then write that dictionary to a file. I believe the most suitable file type is the `.pkl` binary file, which can be use to store variables. As this is a binary file, we must open it in `'wb'` mode (w for write, b for binary), and then dump the `parameters` dictionary using the `pickle` module (which we must import first).

neural_network.py / class MultilayerPerceptron

```
31      # loads a model from a file path
32      def load_model(self, file_path):
33          with open(file_path, 'rb') as file:
34              parameters = pickle.load(file)
35          self.weights = parameters['weights']
36          self.biases = parameters['biases']
```

Code Snippet 3.37: Loading Models

To load the model, we can do the same thing but in reverse. First, open the file in `'rb'` mode (r for read, b for binary) and load the dictionary into a `parameters` variable. We can then set the `weights` and `biases` attributes to the respective elements in the dictionary.

No.	Test	Input	Type	Expectation	Output

t_{28}	Saving a Model	–	–	Weights and biases saved to an external binary file	As Expected
t_{29}	Loading a Model	–	Valid	Attributes set to the same weights and biases	As Expected

Table 3.36: Testing Table for Code Snippets 3.36 & 3.37

We should only be able to load weights and biases if they originally came from a network with the same layer sizes.

`neural_network.py / class MultilayerPerceptron / def save_model`

```
24          parameters = {
25              'weights': self.weights,
26              'biases': self.biases,
27              'layer_sizes': self.layer_sizes
28          }
```

`neural_network.py / class MultilayerPerceptron / def load_model`

```
36          if self.layer_sizes != parameters['layer_sizes']:
37              raise Exception('layer sizes do not match!')
```

Code Snippet 3.38: Ensuring Layer Sizes are the Same

To do this, we can also store the `layer_sizes` attribute in the dictionary which is written to file, and then when loading, we can check if the `layer_sizes` in the dictionary matches this instance's `layer_sizes`, before we set the weights and biases. If it does not match, then we can `raise` an `Exception`, to say that the layer sizes do not match.

No.	Test	Input	Type	Expectation	Output
t_{30}	Loading a Model with different layer sizes	–	Invalid	Error is thrown	As Expected

Table 3.37: Testing Table for Code Snippet 3.38

3.2.3 Part 7 – Implementing the Feedforward Algorithm & Prediction

Now it's time to implement the Feedforward algorithm, as shown in the flowchart in Figure 2.23.

`neural_network.py / class MultilayerPerceptron`

```
22      # carry out the feedforward algorithm
23      def feedforward(self):
24          for layer in range(0, self.L):
25              self.z[layer+1] = np.matmul(self.weights[layer],
                ↪  self.activations[layer]) + self.biases[layer+1]
26              self.activations[layer+1] = sigmoid(self.z[layer+1])
```

Code Snippet 3.39: Implementing the Feedforward Algorithm

In the flowchart and the class diagam, I had said the input layer would be an input/parameter to this function, but I have since decided it would make more sense to just directly access it from the `self.activations` attribute (it will be at index `0`).

Next, according to the flowchart, we must iterate for l from 0 to $L-1$, which can be simply done with a python `for` loop (iterator variable renamed to `layer` for greater readability).

Then for each of the values for l (`layer`), we must compute $\mathbf{a}^{(l+1)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)}\right)$ (equation 2.29). As we will also need to get the various values for z during training, I have broken this expression up into $\mathbf{z}^{(l+1)} = \mathbf{W}^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)}$ and $\mathbf{a}^{(l+1)} = \sigma\left(\mathbf{z}^{(l+1)}\right)$, which you can see implemented on lines 25 and 26.

`neural_network.py / class MultilayerPerceptron`

```
28      # run the feedforward algorithm on a set of input data and return the
        ↪  result
29      def predict(self, input_data):
30          self.activations[0] = input_data
31          self.feedforward()
32          return self.activations[self.L]
```

Code Snippet 3.40: Making a Prediction

If we then want to more formally make a prediction, we must first get the input data (the parameter of this `def predict` function), then feed it forward through the network, and return the activations in the output layer (the layer at index L).

No.	Test	Input	Type	Expectation	Output

t_{31}	Feedforward Algorithm	–	–	All activations and z-values are updated	As Expected
t_{32}	Making a Prediction	–	–	Returns the output layer of neurons	As Expected

Table 3.38: Testing Table for Code Snippets 3.39 & 3.40

Again, due to the nature of the caulculations involved in the network, it is difficult to test these algorithms. Therefore, I was only looking for the correct structure of the answers (e.g. the correct dimensions of matrix) to determine if the output is as expected.

3.2.4 Part 8 – Implementing the Backpropogation Algorithm & Training

If we actually try to make a prediction, the network will return absolute nonsense, and so we must be able to train it, using the backpropogation algorithm.

`neural_network.py / class MultilayerPerceptron`

```
28      # carry out the backpropogation algorithm
29      def backpropogate(self, examples):
30          # reset the training lists
31          # iterate over each example
32          for example in examples:
33              # reset the activations, z-values and errors
34              # make a feedforward pass
35              # calculate the error in the output layer
36              # backpropogate the error to previous layers
37              # save the activations, z-values and errors
38          # use gradient descent to update the weights and biases
```

Code Snippet 3.41: Backpropogation Algorithm Structure

Here, in Code Snippet 3.41, I have outlined the general backpropogation algorithm, as shown in the flowchart in Figure 2.26, with some additional steps we need to consider when practically carrying it out (for example, resetting the lists).

neural_network.py / class MultilayerPerceptron / def backpropogate

```
30          # reset the training lists
31          self.training_activations = []
32          self.training_z = []
33          self.training_errors = []
34          # iterate over each example
35          for example in examples:
36              # reset the activations, z-values and errors
37              self.activations = [np.matrix(np.zeros(shape=(layer_size,1)))
                ↪  for layer_size in self.layer_sizes]
38              self.z = [np.matrix(np.zeros(shape=(layer_size,1))) for
                ↪  layer_size in self.layer_sizes]
39              self.errors = [np.matrix(np.zeros(shape=(layer_size,1))) for
                ↪  layer_size in self.layer_sizes]
```

Code Snippet 3.42: Resetting the Lists

Resetting these lists is easy enough, as all we need to do is ensure the training lists are empty, and reset the activations, z-values and errors to 0 when processing each example (by doing the same things as in the constructor method).

neural_network.py / class MultilayerPerceptron / def backpropogate

```
40              # make a feedforward pass
41              self.activations[0] = example['input']
42              self.feedforward()
43              # calculate the error in the output layer
44              self.errors[self.L] = np.multiply(
45                  self.activations[self.L] - example['output'],
46                  sigmoid_prime(self.z[self.L])
47              )
```

Code Snippet 3.43: Calculating the error in the output layer

Next, we must feedforward the example's input, to get the network's current prediction, and then compare that with the "correct" output to get the error.

We must use the formula $\boldsymbol{\delta}^{(L)} = \left(\mathbf{a}^{(L)} - \mathbf{y}\right) \odot \left(\sigma'\left(\mathbf{z}^{(L)}\right)\right)$ (equation 2.41) to calculate the error in the output layer, which has been implemented in lines 44 to 47 (the `np.multiply` function carries out the $\odot$ operation).

neural_network.py / class MultilayerPerceptron / def backpropogate

```
48              # backpropogate the error to previous layers
49              for layer in range(self.L, 1, -1):
50                  self.errors[layer-1] = np.multiply(
51                      np.matmul(
52                          self.weights[layer-1].T,
53                          self.errors[layer]
54                      ),
55                      sigmoid_prime(self.z[layer-1])
56                  )
57              # save the activations, z-values and errors
58              self.training_activations.append(
                ↪  copy.deepcopy(self.activations))
59              self.training_z.append(copy.deepcopy(self.z))
60              self.training_errors.append(copy.deepcopy(self.errors))
```

Code Snippet 3.44: Backpropogating the Error

After this, we can backpropogate the error, using the formula $\boldsymbol{\delta}^{(l-1)} = \left(\left(\mathbf{W}^{(l-1)}\right)^T \boldsymbol{\delta}^{(l)}\right) \odot \left(\sigma'\left(\mathbf{z}^{(l-1)}\right)\right)$ (equation 2.51), for each l from L to 2. The iteration is easily implemented with a `for` loop on line 49, and the equation is implemented in lines 50 to 56. The attribute `.T` of a `np.matrix` is the transpose of that matrix, as required by the formula.

Now that we have a full set of activations, z-values, and errors, we can save them to their respective lists to be able to compute the gradient of the cost function.

neural_network.py / class MultilayerPerceptron

```
29      def backpropogate(self, examples, learning_rate):
```

neural_network.py / class MultilayerPerceptron / def backpropogate

```
61          # use gradient descent to update the weights and biases
62          for layer in range(self.L, 0, -1):
63              self.weights[layer-1] -= (learning_rate/len(examples)) * sum(
64                  [np.matmul(
65                      self.training_errors[example][layer],
66                      self.training_activations[example][layer-1].T
67                  ) for example in range(len(examples))],
68                  np.matrix(np.zeros(self.weights[layer-1].shape)) # (extra
                    ↪ argument in sum function to add matrices instead of
                    ↪ numbers)
69              )
70              self.biases[layer] -= (learning_rate/len(examples)) * sum(
71                  [self.training_errors[example][layer] for example in
                    ↪ range(len(examples))],
72                  np.matrix(np.zeros(self.biases[layer].shape)) # (extra
                    ↪ argument in sum function to add matrices instead of
                    ↪ numbers)
73              )
```

Code Snippet 3.45: Updating the weights and biases

Finally, we must update the weights using the formula $\mathbf{W}^{(l-1)} = \mathbf{W}^{(l-1)} - \frac{\eta}{m} \sum_{x=1}^{m} \boldsymbol{\delta}_x^{(l)} \left(\mathbf{a}_x^{(l-1)}\right)^T$ (equation 2.56), implemented on lines 63 to 69.

And we must update the biases using the formula $\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \frac{\eta}{m} \sum_{x=1}^{m} \boldsymbol{\delta}_x^{(l)}$ (equation 2.57), implemented on lines 70 to 73. (for each of the layers from the final layer, to the input layer).

One of the terms in these equations is η, the learning rate, which is just a constant to determine how fast the network should learn. I had forgotten to add this as a parameter of the `def backpropogate` function in the class diagram, but I have since added it here.

Both of these formulas also involve summing over multiple matrices, which I have implemented with the default python `sum` function. As we are summing matrices instead of numbers, we must also pass in a matrix of 0s, of the same dimension as the matrix we are updating, for the sum to work.

This should be the full backpropogation algorithm, and so we are ready to test it. All I am looking for at the moment is that it does not throw an error, as it is impossible to determine if the answer is correct or not without carrying out the algorithm again.

No.	Test	Input	Type	Expectation	Output
t_{33}	Backpropogation algorithm doesn't throw an error	–	–	No errors when carrying out the backpropogation algorithm	An Error

Table 3.39: Testing Table for Code Snippets 3.41, 3.42, 3.43, 3.44 & 3.45

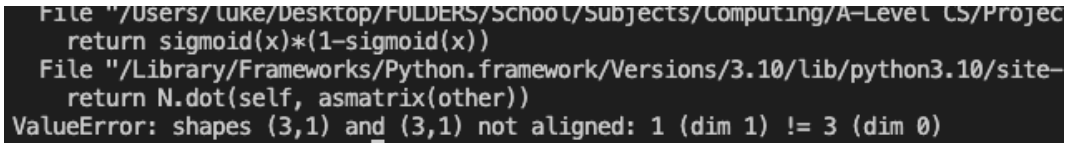


Figure 3.23: Backpropogation Error

The error seems to be that we are attempty to carry out matrix multiplication between $\sigma(x)$ and $1 - \sigma(x)$.

`neural_network.py`

```
100 # derivative of the activation function
101 def sigmoid_prime(x):
102     return np.multiply(sigmoid(x), 1-sigmoid(x))
```

Code Snippet 3.46: Fixing the σ' function

We can fix this by specifying we want to do an element-wise multiplication `np.multiply` instead of a matrix multiplication.

No.	Test	Input	Type	Expectation	Output
t_{33}	Backpropogation algorithm doesn't throw an error	–	–	No errors when carrying out the backpropogation algorithm	As Expected

Table 3.40: Testing Table for Code Snippet 3.46

The backpropogation algorithm now does not crash. Whether or not it actually works will be tested shortly, in the next section.

neural_network.py

```
4 import random
```

neural_network.py / class MultilayerPerceptron

```
82     # carry out stochastic gradient descent over a number of epochs to
       ↪ train a model
83     def train(self, examples, mini_batch_size, num_epochs, learning_rate):
84         for epoch in range(1, num_epochs+1):
85             print(f'training epoch {epoch}/{num_epochs}...') # log message
86             # randomly split into mini batches
87             random.shuffle(examples)
88             mini_batches = [examples[(batch_number *
               ↪ mini_batch_size):((batch_number+1) * mini_batch_size)] for
               ↪ batch_number in range(int(np.ceil(len(examples) /
               ↪ mini_batch_size)))]
89             # backpropogate for each mini batch
90             for mini_batch in mini_batches:
91                 self.backpropogate(mini_batch, learning_rate)
```

Code Snippet 3.47: Implementing Stochastic Gradient Descent

In order to fully train the network, we must split the training examples into mini batches of a particular size, and then carry out the backpropogation algorithm on each of those. We then may repeat the process over a number of "epochs", to further train the model.

No.	Test	Input	Type	Expectation	Output
t_{34}	Stochastic Gradient Descent doesn't throw an error	–	–	No errors when carrying out the full training algorithm	As Expected

Table 3.41: Testing Table for Code Snippet 3.47

3.2.5 Part 9 – Testing the Neural Network

The `class MultilayerPerceptron` is now complete, and can be used to make predictions.

Before training it on my HDI training data, I am going to first test that it actually works, by training it on something else with a lot of training data. Both Grant Sanderson and Micheal Nielsen (who wrote the resources I used to learn about neural networks) used the MNIST handwritten digits data set to test their networks, so I have decided to do the

same thing. This is because it has been proven that this method works to make accurate predictions on these handwritten digits.

The MNIST data set contains 70000 images of handwritten digits, each attatched with a number 0 to 9, to say which digit it is supposed to be. Each image is grayscale and 28 by 28 pixels, and so is therefore made of $28^2 = 784$ numbers between 0 and 1, which represent the brightness of each pixel.

	image	label
0		4
1		1
2		0
3		7
4		8
5		1
6		2
7		7
8		1
9		6

Figure 3.24: Examples from the MNIST Data Set

Figure 3.24 shows 10 of the handwritten digits from the MNIST data set, and their respective labels.

I first needed to download the dataset and read it. I decided to retrieve it and some code to read it from Micheal Nielsen's code repository [9], to ensure that I get the correct data.

(This code has been adapted from Micheal Nielsen's code repository [9])
`network_training / load_digits_data.py`

```
import pickle
import gzip

# loads the MNIST data set
def load_data():
    with gzip.open('data/mnist.pkl.gz', 'rb') as file:
        unpickled = pickle._Unpickler(file)
        unpickled.encoding = 'latin1'
        training_data, validation_data, test_data = unpickled.load()
    return (training_data, validation_data, test_data)
```

Code Snippet 3.48: Loading in the MNIST data set

The MNIST data set seems to be in the `.pkl.gz` format, which means we must open it with the `gzip` library, and then unpickle it with the `pickle` module.

We can then load the training data, validation data, and testing data from the unpickled and uncompressed file. The 70000 examples are split up into 50000 training examples, 10000 validation examples and 10000 testing examples. I will only be using the 50000 training examples and the 10000 testing examples for the purposes of testing my network.

We now must convert this data into the correct format, to feed into our neural network.

`network_training / load_digits_data.py`

```
3  import numpy as np
```

(This code has been adapted from Micheal Nielsen's code repository [9])
`network_training / load_digits_data.py`

```
13 # makes a vector for the output layer from the correct output label
14 def convert_to_vector(correct_output):
15     output_layer = np.zeros((10, 1))
16     output_layer[correct_output] = 1
17     return output_layer
18 
19 training_data, _, test_data = load_data()
20 # convert training data into the right format
21 training_inputs = [np.reshape(input_data, (784, 1)) for input_data in
    ↪  training_data[0]]
22 training_outputs = [convert_to_vector(output_data) for output_data in
    ↪  training_data[1]]
23 training_data = [{
24     'input': training_inputs[example],
25     'output': training_outputs[example]
26 } for example in range(len(training_inputs))]
27 # convert testing data into the right format
28 test_inputs = [np.reshape(input_data, (784, 1)) for input_data in
    ↪  test_data[0]]
29 test_outputs = test_data[1]
30 test_data = [{
31     'input': test_inputs[example],
32     'output': test_outputs[example]
33 } for example in range(len(test_inputs))]
```

Code Snippet 3.49: Converting the MNIST data to the right format

First, we must load the data using the function defined in the previous code snippet. It returns all 3 data sets, but I only need the training one and the testing one, so in place of

the validation data I have used an underscore, to show that that return variable will not be used.

As each image is made of 784 grayscale pixels, that means our input layer must have 784 neurons, and so should therefore be a matrix with 1 column and 784 rows (hence why we `np.reshape` it).

The neural network will have 10 neurons in the output layer, representing the confidence in each digit. For example, an output layer of `[0, 0, 0.9, 0, 0, 0.1, 0, 0, 0, 0]` would suggest that the network believe that particular digit is a 2, as `0.9` > all of the other activations.

Therefore, we must write a function, `def convert_to_vector`, which takes in the correct output number, and creates the output layer vector from it. This will be a matrix with 10 rows and 1 column, filled with 0s, except for at the index of the correct answer, where it will be 1. This function must then be applied to all items in the training outputs.

We can then save these 2 variables, `training_inputs` and `training_outputs` to a dictionary ready for the neural network to read.

This process must be repeated for the test data, except we don't need to convert the output to a vector, as we will instead be converting the vector generated by the network into a formal prediciton (1 digit).

`network_training / load_digits_data.py`

```
29 # save data to a file
30 with open(f'data/digits.pkl', 'wb') as file:
31     pickle.dump({'training': training_data, 'testing': test_data}, file)
```

Code Snippet 3.50: Saving the MNIST data to a file

Once we have all of the training data and testing data, we can save it to a file, again using `pickle`.

No.	Test	Input	Type	Expectation	Output
t_{35}	Convert MNIST training and testing data to right format	Strange format	–	Correct dimensions of matrix, in a dictionary with input and output keys	As Expected

Table 3.42: Testing Table for Code Snippets 3.48, 3.49 & 3.50

network_training / train_digits.py

```
import pickle

# load the training data from the file
with open('data/digits.pkl', 'rb') as file:
    data = pickle.load(file)
training_data = data['training']

# import the class MultilayerPerceptron from the neural_network.py file
from neural_network import MultilayerPerceptron
```

Code Snippet 3.51: Loading the MNIST Training Data

In a different file, for training the network, we can then read this data and select only the training data from the dictionary.

We then need to import the `class MultilayerPerceptron` from the `neural_network.py` file. Unfortunately, this returned a `ModuleNotFoundError`.

```
train_digits.py", line 14, in <module>
    from neural_network import MultilayerPerceptron
ModuleNotFoundError: No module named 'neural_network'
```

Figure 3.25: `ModuleNotFoundError`

This is because the current file (`train_digits.py`) is in another directory (`network_training`) from `neural_network.py`. We must therefore import from a parent directory.

I did not know how to resolve this issue, but after researching the proble, I found a GeeksforGeeks solution [10] on how to import from a parent directory.

(This code has been adapted from this GeeksforGeeks solution: [10])
network_training / train_digits.py

```
# import the class MultilayerPerceptron from the neural_network.py file
import os
import sys
current = os.path.dirname(os.path.realpath(__file__))
parent = os.path.dirname(current)
sys.path.append(parent)
from neural_network import MultilayerPerceptron
```

Code Snippet 3.52: Importing from a Parent Directory

This solution seems to be getting around the problem by finding the "real path" of the

current file (`__file__`), finding its parent directory, and adding that to the `sys.path`, which means we can now accecss files from the parent directory (namely, `neural_network.py`).

No.	Test	Input	Type	Expectation	Output
t_{36}	Loading the MNIST training data	–	–	Loads the MNIST training data and imports the Multilayer Perceptron class	As Expected

Table 3.43: Testing Table for Code Snippets 3.51 & 3.52

`network_training / train_digits.py`

```
16 # train the network
17 network = MultilayerPerceptron([784, 16, 16, 10])
18 network.train(training_data, 1000, 10, 1)
19 network.save_model('digits')
```

Code Snippet 3.53: Training a Neural Network on the MNIST Data Set

Now that we have successfully imported the `class MultilayerPerceptron`, we can attempt to train it. We must first instantiate a `MultilayerPerceptron`, with 784 neurons in the input layer and 10 neurons in the output layer. The number of neurons in the hidden layer(s), and the number of hidden layers have no requirements, so I have arbitrarily decided to make 2 hidden layers, each with 16 neurons.

We can then call the `train` method, passing in the training data. Again, I have arbitrarily decided that the mini batch size is 1000, the number of epochs is 10, and the learning rate is 1.

Once it is fully trained, we can save the weights and biases to a file, by calling the `save_model` method.

`neural_network.py / class MultilayerPerceptron / def train`

```
90             for num, mini_batch in enumerate(mini_batches):
91                 print(f' mini batch {num}/{len(mini_batches)}...') # log
                    ↪  message
```

Code Snippet 3.54: Adding another log message

Each epoch was taking a very long time, so I decided to add another log message, for each mini batch.

No.	Test	Input	Type	Expectation	Output
t_{37}	Training the neural network on MNIST	–	–	No errors when carrying out the full training algorithm	As Expected

Table 3.44: Testing Table for Code Snippets 3.53 & 3.54

`network_training / test_digits.py`

```
import pickle

# load the test data from the file
with open('data/digits.pkl', 'rb') as file:
    data = pickle.load(file)
testing_data = data['testing']

# import the class MultilayerPerceptron from the neural_network.py file
import os
import sys
current = os.path.dirname(os.path.realpath(__file__))
parent = os.path.dirname(current)
sys.path.append(parent)
from neural_network import MultilayerPerceptron
```

Code Snippet 3.55: Loading the MNIST Testing Data

In order to test our trained neural network, we must first do the same thing as in the training file, which is loading the testing data from the file, and then importing the `class MultilayerPerceptron` from the parent directory.

No.	Test	Input	Type	Expectation	Output
t_{38}	Loading the MNIST testing data	–	–	Loads the MNIST testing data and imports the Multilayer Perceptron class	As Expected

Table 3.45: Testing Table for Code Snippet 3.55

network_training / test_digits.py

```
16 # iterate over each of the testing examples and see how many it gets
   ↪ right
17 network = MultilayerPerceptron([784, 16, 16, 10])
18 network.load_model('models/digits.pkl')
19 success = 0
20 for total_so_far, example in enumerate(testing_data):
21     prediction = network.predict(example['input'])
22     list_of_probabilities = [prediction.item(x,0) for x in range(10)]
23     if list_of_probabilities.index(max(list_of_probabilities)) ==
       ↪ example['output']:
24         success += 1
25     print(f'after {total_so_far+1} examples, the success rate is
       ↪ {(success/(total_so_far+1))*100}%')
```

Code Snippet 3.56: Testing the Neural Network on the MNIST Data Set

To test it, again we must first instantiate a `MultilayerPerceptron`, with the same layer sizes, and then use the `load` method to set the weights and biases to that of the trained network.

Then, we must iterate over each of the pieces of testing data, and make a prediction. Currently, this prediction is a column vector with 10 entries, one of which should hopefully be much larger than the rest, which is the network's prediction. To get this prediction, we must find the index of the maximum value of the column vector. If it is the same as the example's output (what the number is supposed to be), then the network has successfully predicted the digit.

After each prediction, we can print the current success rate, to see how it evolves over time.

No.	Test	Input	Type	Expectation	Output
t_{39}	Testing the neural network on MNIST	–	–	Cumulative % Success printed after each test	As Expected

Table 3.46: Testing Table for Code Snippet 3.56

```
after 1 examples, the success rate is 100.0%
after 2 examples, the success rate is 100.0%
after 3 examples, the success rate is 100.0%
after 4 examples, the success rate is 100.0%
after 5 examples, the success rate is 80.0%
after 6 examples, the success rate is 83.33333333333334%
after 7 examples, the success rate is 71.42857142857143%
```

Figure 3.26: Output for the first 7 training examples

Figure 3.26 shows the output for the first 7 examples. This shows that it got the first 4 correct, then 1 wrong, then another one correct, and then another one wrong.

```
after 9994 examples, the success rate is 53.53211927156294%
after 9995 examples, the success rate is 53.536768384192094%
after 9996 examples, the success rate is 53.54141656662665%
after 9997 examples, the success rate is 53.546063819145736%
after 9998 examples, the success rate is 53.5507101420284%
after 9999 examples, the success rate is 53.54535453545355%
after 10000 examples, the success rate is 53.55%
```

Figure 3.27: Output for the last 7 training examples

Figure 3.27 shows the output for the last 7 examples. This shows that it has converged on a success rate of around 53.55%. This is a good result, but not great as it still got about half of them wrong.

`network_training / train_digits.py`

```
18 network.train(training_data, 10, 30, 3)
```

Code Snippet 3.57: Improving the Training Paremeters

If we adjust the training parameters, to have smaller mini batches, more epochs and a greater leanring rate, the success rate will increase. (To get a new model, we must of course re-run `train_digits.py` and `test_digits.py`)

```
after 9994 examples, the success rate is 93.52611566940165%
after 9995 examples, the success rate is 93.52676338169084%
after 9996 examples, the success rate is 93.52741096438577%
after 9997 examples, the success rate is 93.52805841752526%
after 9998 examples, the success rate is 93.52870574114823%
after 9999 examples, the success rate is 93.52935293529353%
after 10000 examples, the success rate is 93.53%
```

Figure 3.28: Improved Output for the last 7 training examples

Figure 3.28 shows the output for the last 7 examples, with new training parameters. We now converge on around 93.53%, which is much better.

If the training didn't work, then the success rate should be around 10%, as it would just be picking digits at random, and so it would have a $\frac{1}{10}$ chance of getting it right. 93.53% is much greater than 10%, and so I believe there is sufficient evidence to suggest that the neural network, and the backpropogation algorithm is in fact working.

No.	Test	Input	Type	Expectation	Output
T_5	Neural Network works	MNIST Handwritten Digits	–	A Success rate much larger than 10%	As Expected

Table 3.47: T_5 Testing Table (for Code Snippet 3.57)

3.2.6 Part 10 – Training the Neural Network

Now it is time to trian the neural network to predict the HDI of a given area.

`network_training / load_hdi_data.py`

```
import csv
import numpy as np

# read csv file
with open('data/training_data.csv', 'r') as file:
    reader = csv.DictReader(file)
    hdiData = []
    for record in reader:
        # convert it to the right format
        hdiData.append({
            "input": np.matrix([[1000 if factor == '' else float(factor)]
            ↪  for factor in list(record.values())[4:]]),
            "output": np.matrix([[float(record['hdi'])]])
        })
```

Code Snippet 3.58: Loading the HDI Data

First, we must convert the HDI data into the right format. We can read the `.csv` file using the same methods as before, except when we go to append each record, we must slightly change the format.

Currently, each record is a dictionary of all the factors, along with some extra information (such as the name of the region). The input data should be a list of factors (as a column

vector), which currently start at index 4. Each of these factors is currently of type `str`, so we must convert it to a `float`. Some of the factors are also currently an empty string, which correspond to average distance factors in regions that have 0 of that building. I have set these to 1000, as that is larger than every other average distance factor.

The output should just be the HDI, which is located at key `'hdi'`.

`network_training / load_hdi_data.py`

```
3  import random
4  import pickle
```

`network_training / load_hdi_data.py`

```
17 # split the data set into training and testing
18 def test_train_split(data_set, proportion_train):
19     random.shuffle(data_set)
20     return {
21         "training": data_set[:int((len(data_set)*proportion_train)//1)],
22         "testing": data_set[int((len(data_set)*proportion_train)//1):]
23     }
24
25 hdiData = test_train_split(hdiData, 0.9)
26 # save data to a file
27 with open(f'data/hdi.pkl', 'wb') as file:
28     pickle.dump(hdiData, file)
```

Code Snippet 3.59: Splitting into Testing and Training Data

We then must randomly split the data set into training and testing data. For this, I have written a function, `def test_train_split`, which will take in a data set and a proportion which should be training data.

If we multiply the length of the data set by the proportion, we get the index of the final piece of data which should be for training. Therefore, everything before that should be training and everything after it should be testing.

After we have split the data set, we can again save it to file as we have done before.

No.	Test	Input	Type	Expectation	Output
t_{40}	Converting HDI Data to the correct format	`.csv`	–	Numpy Matrices	As Expected

t_{41}	Test Train Split	Full Data Set	–	First `proportion_train`% are training, rest is testing	As Expected

Table 3.48: Testing Table for Code Snippets 3.58 & 3.59

`network_training / train_hdi.py`

```
import pickle

# load the training data from the file
with open('data/hdi.pkl', 'rb') as file:
    data = pickle.load(file)
training_data = data['training']

# import the class MultilayerPerceptron from the neural_network.py file
import os
import sys
current = os.path.dirname(os.path.realpath(__file__))
parent = os.path.dirname(current)
sys.path.append(parent)
from neural_network import MultilayerPerceptron

# train the network
network = MultilayerPerceptron([24, 10, 10, 1])
network.train(training_data, 10, 30, 3)
network.save_model('hdi-temp')
```

Code Snippet 3.60: Training the Neural Network on HDI

We can then train the neural network on the HDI training data in exactly the same way as we did for the MNIST handwritten digits.

Except, this time we must have 24 neurons in the input layer (for each of the 24 factors), and 1 neuron in the output layer. I initially decided to have 2 hidden layers, each with 10 neurons, and to keep the remaining training parameters the same. (mini batch size = 10, number of epochs = 30, learning rate = 3)

No.	**Test**	**Input**	**Type**	**Expectation**	**Output**

t_{42}	Training the neural network on HDI	–	–	No errors when carrying out the full training algorithm	As Expected

Table 3.49: Testing Table for Code Snippet 3.60

network_training / test_hdi.py

```
import pickle

# load the test data from the file
with open('data/hdi.pkl', 'rb') as file:
    data = pickle.load(file)
testing_data = data['testing']

# import the class MultilayerPerceptron from the neural_network.py file
import os
import sys
current = os.path.dirname(os.path.realpath(__file__))
parent = os.path.dirname(current)
sys.path.append(parent)
from neural_network import MultilayerPerceptron

# iterate over each of the testing examples and see how many it gets
↪ right
network = MultilayerPerceptron([24, 10, 10, 1])
network.load_model('models/hdi-temp.pkl')
```

Code Snippet 3.61: Preparing to Test on the HDI

When we test the network, we should again load it in exactly the same way as with the MNIST Data Set, with the same changes to the hidden layers.

network_training / test_hdi.py

```
19 differences = []
20 for num, example in enumerate(testing_data):
21     correctAnswer = example["output"].item(0,0)
22     prediction = round(network.predict(example['input']).item(0,0), 3)
23     difference = round(abs(correctAnswer-prediction), 3)
24     print(f'({num}) correct output: {correctAnswer}, prediction:
       ↪  {prediction}, difference: {difference}')
25     differences.append(difference)
26
27 import numpy as np
28 print(f'average difference: {np.mean(differences)}')
```

Code Snippet 3.62: Testing the Neural Network on HDI

As the output for this network is slightly different, we should test it in a different way. The correct HDI is given by the `'output'` of the testing example, and we can compare that with the prediction.

At each stage, we round our values to 3 decimal places, as this is what HDI is most commonly quoted to. At the end, we can take the mean to find an average differene.

No.	Test	Input	Type	Expectation	Output
t_{43}	Testing the neural network on HDI	–	–	Difference for each prediction calculated, as well as average difference	As Expected

Table 3.50: Testing Table for Code Snippets 3.61 & 3.62

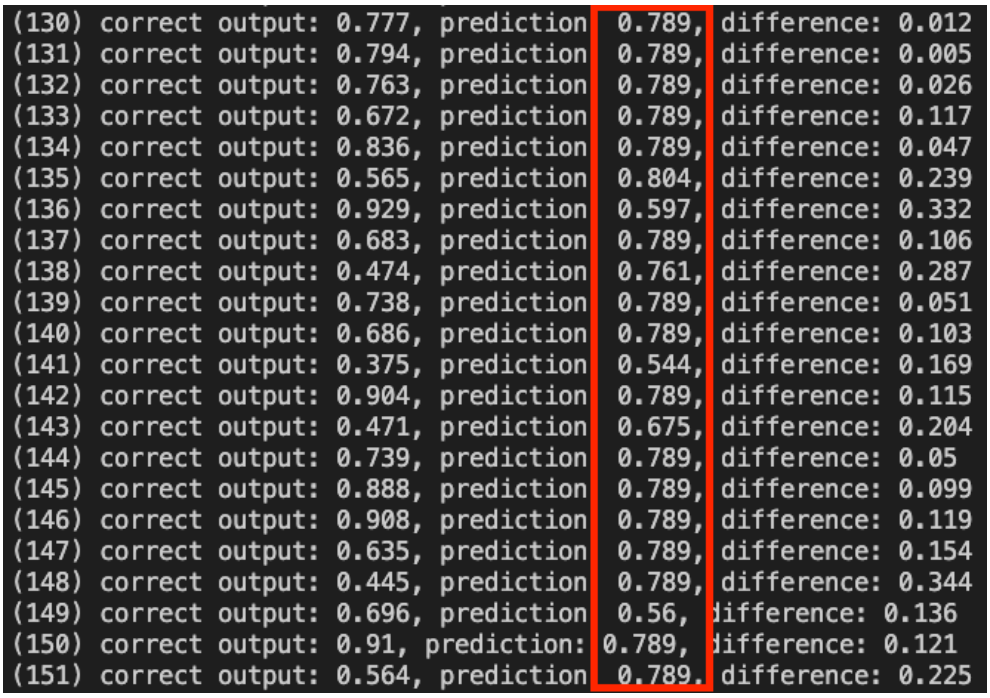


Figure 3.29: Some HDI Predictions

Figure 3.29 shows some of the network's predictons. As you can see, most of them are the same, with a prediction of 0.789, which is obviously not ideal.

The first thing I thought of to ammend this is to change the scale of the factors.

`network_training / load_hdi_data.py`

```
11          # convert it to the right format
12          hdiData.append({
13              "input": np.matrix([[100 if factor == '' else float(factor)/10
                ↪  if index <= 11 else float(factor)] for index, factor in
                ↪  enumerate(list(record.values())[4:])]),
14              "output": np.matrix([[float(record['hdi'])]])
15          })
```

Code Snippet 3.63: Dividing some factors by 10

Each of the average distance factors often take values in between 1 and 10, but this particular type of neural network often works best with numbers between 0 and 1. Therefore, I have divided each of the average distance factors (at indexes `0` to `11`) by 10, to get most of them in this range.

No.	Test	Input	Type	Expectation	Output

t_{44}	Scaling Average Distance Factors	Average Distance Factor	–	Divided by 10	As Expected

Table 3.51: Testing Table for Code Snippet 3.63

The network was still not predicting very well, so I had to adjust the training parameters.

`network_training / train_hdi.py`

```
16 # train the network
17 network = MultilayerPerceptron([24, 18, 12, 6, 3, 1])
18 network.train(training_data, 5, 50, 0.2)
19 network.save_model('hdi-temp')
```

`network_training / test_hdi.py`

```
16 # iterate over each of the testing examples and see how many it gets
   ↪ right
17 network = MultilayerPerceptron([24, 18, 12, 6, 3, 1])
18 network.load_model('models/hdi-temp.pkl')
```

Code Snippet 3.64: Adjusting the Training Parameters

After many attempts, I ended up with a neural network with 4 hidden layers, each with 18, 12, 6 and 3 neurons respectively, which was trained with a mini batch size of 5, over 50 epochs and a learning rate of 0.2.

No.	**Test**	**Input**	**Type**	**Expectation**	**Output**
t_{42}	Training the neural network on HDI	–	–	No errors when carrying out the full training algorithm	As Expected

Table 3.52: Testing Table for Code Snippet 3.67

```
(159) correct output: 0.707, prediction: 0.694, difference: 0.013
(160) correct output: 0.534, prediction: 0.539, difference: 0.005
(161) correct output: 0.496, prediction: 0.601, difference: 0.105
(162) correct output: 0.543, prediction: 0.675, difference: 0.132
(163) correct output: 0.856, prediction: 0.812, difference: 0.044
(164) correct output: 0.858, prediction: 0.697, difference: 0.161
(165) correct output: 0.605, prediction: 0.595, difference: 0.01
(166) correct output: 0.577, prediction: 0.669, difference: 0.092
(167) correct output: 0.494, prediction: 0.669, difference: 0.175
(168) correct output: 0.762, prediction: 0.695, difference: 0.067
(169) correct output: 0.733, prediction: 0.695, difference: 0.038
(170) correct output: 0.491, prediction: 0.565, difference: 0.074
(171) correct output: 0.769, prediction: 0.69, difference: 0.079
(172) correct output: 0.8, prediction: 0.829, difference: 0.029
(173) correct output: 0.811, prediction: 0.726, difference: 0.085
(174) correct output: 0.593, prediction: 0.669, difference: 0.076
(175) correct output: 0.864, prediction: 0.81, difference: 0.054
(176) correct output: 0.933, prediction: 0.821, difference: 0.112
(177) correct output: 0.793, prediction: 0.767, difference: 0.026
average difference: 0.08447191011235955
```

Figure 3.30: Improved HDI Predictions

The variance in these predictions was very high, and the average difference is very low, which is what we need.

3.2.7 Review

No.	Test	Inputs	Pass/Fail
T_5	Neural Network works	Using a *different* data set, e.g. handwritten digits, which certainly has enough training data	Pass

Table 3.53: Passed Testing Data for Milestone 2

The fact that we successfully predicted 93.53% of MNIST handwritten digits suggests that T_5 has been passed, which implies the Success Criteria S_5 and S_6 have been met

No.	Success Criterion	Justification	Success
S_5	Successfully Implement the Feedforward Algorithm	This algorithm will allow us to make predictions on the HDI given a set of factors.	✓
S_6	Successfully Implement the Backpropogation Algorithm	This algorithm will allow us to train the neural network base on a set of training examples.	✓

S_7	Successfully train the Neural Network	In order to accurately predict the HDI of a given area, the neural network must train on existing examples to see what makes a high HDI.	✓

Table 3.54: Achieved Success Criteria for Milestone 2

I believe S_7 has also been met, as we have been able to train a neural network which can (vaguely) predict the HDI of a given area.

3.3 Milestone 3 – Initial User Interface

3.3.1 Success Criteria

By the end of this milestone, I hope to have a fully functioning user interface, in which the user can select their region on a map, predict the HDI of it, and have suggestions made to improve it.

No.	Success Criterion	Justification
S_8	User can select an area on the map for which they want to analyse	This will allow the user to choose which area they want to predict the HDI of by drawing on the map.
S_9	Neural Network accurately predicts HDI from those factors	Once the user has selected their area, the HDI must be predicted based on factors calculated from that area.
S_{10}	Changes are suggested to increase the HDI	This will allow the user to make an informed decision on what to do in order to improve their area.
S_{11}	Accurately calculate where a new building would need to be in order to increase the HDI the most	The specific placement of new buildings (as suggestions to increase HDI) is important to consider, so a suitable place must be chosen in order to decrease the average distance to it as much as possible.

Table 3.55: Success Criteria for Milestone 3

To determine if I have met this success criteria, I will be using the following tests:

No.	Test	Inputs	Expected Output/Justification
T_6	Overall Flow	User Draws Region and then Predicts, User tries to predict before selecting Region, User draws region and doesn't predict	The only valid sequence of steps the user should be able to take is drawing the region and then making a prediction.
T_7	Optimal Place for New Building	Building for which there exists 1 Factor, Building for which there exists more than 1 Factor, Building for which there exists 0 factors	This function should take an average over all of the factors the building is involved in, while throwing an error if it has 0 factors.
T_8	Suggestions fed back into Neural Network	All 8 sugestions for where to place new buildings	A new predicted HDI for each suggestion

Table 3.56: Testing Data for Milestone 3

3.3.2 Part 11 – Drawing on the Map

I believe that having a map to draw on is the most important thing in the GUI, so I will be implemented that first.

`app.py`

```
import gradio as gr

# structure of the UI
with gr.Blocks() as app:
    pass

# launch the UI
app.launch()
```

Code Snippet 3.65: Initial Gradio Framework

Here, I am using the `gradio` library to begin to define the interface. Gradio uses context managers (`with` statement) to make the structure of the GUI very readable and easy to understand (this will become more apparent when we add more and more elements).

Gradio offers many types of interface, but the one I will be using is the `Blocks` interface, which allows for the most customisation. For now, we have no structure to add, so we can just `pass`.

Once we have defined the GUI, we can call the `launch` method on the app.

```
Running on local URL:  http://127.0.0.1:7860

To create a public link, set `share=True` in `launch()`.
```

Figure 3.31: Launching the Gradio App

Running this code produces the output shown in Figure 3.31. When we called the `launch` method, it initialised the gradio interface on port `7860`, of the localhost IP address (`127.0.0.1`). It also tells us that we could create a public link, which I will be doing later to share with my stakeholders.

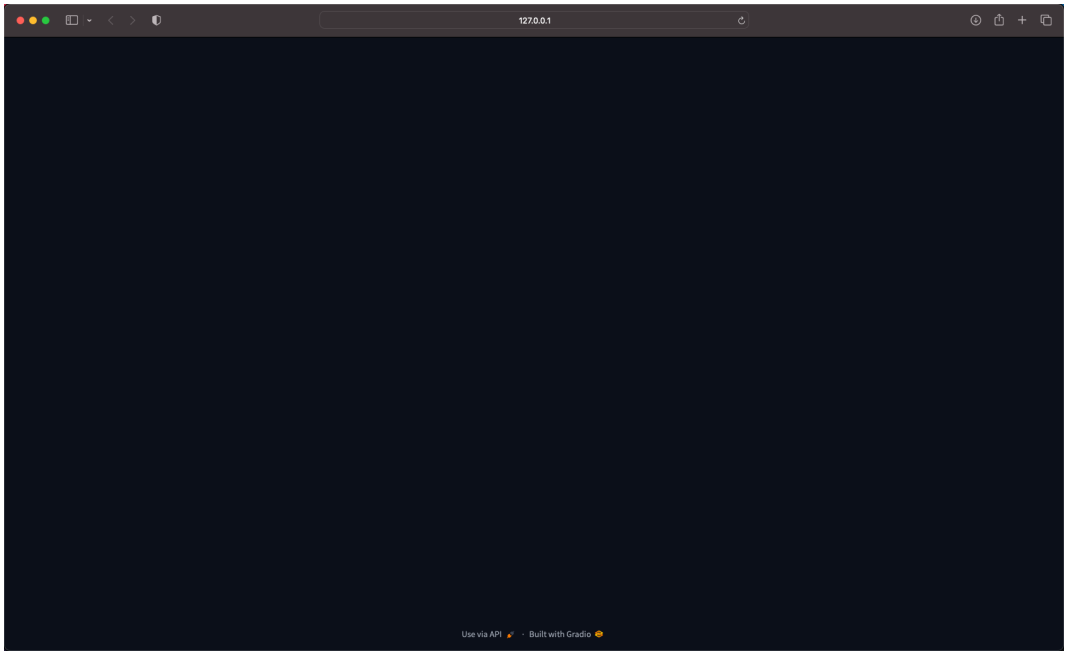


Figure 3.32: Empty Gradio UI

Following the link `http://127.0.0.1:7860` takes us to this empty gradio interface. Once we add components, they will show up here.

No.	Test	Input	Type	Expectation	Output
t_{45}	Initialise Gradio Interface	–	–	Empty Gradio Interface	As Expected

Table 3.57: Testing Table for Code Snippet 3.65

app.py

```
import folium
from folium.plugins import Draw

# initialise the map for drawing on
foliumMap = folium.Map()
draw = Draw(
    export=False,
    position='topleft',
    draw_options={
        'polyline': False,
        'circlemarker': False,
        'polygon': {'allowIntersection': False},
        'marker': False,
        'circle': False,
        'rectangle': False,
    },
    edit_options={'poly': {'allowIntersection': False}}
)
draw.add_to(foliumMap)
```

Code Snippet 3.66: Initialising the Map

Next, we must also import the `folium` library, which lets us create interactive maps.

First, we can make an instance of the `class Map`, and save that to a variable. To be able to draw on it, we must also make an instance of the `class Draw`, and then add that to the map at the end.

The `class Draw` has many attributes which must be set. Namely, specifying `position` to be `'topleft'` will make the drawing tools appear on the top left. Setting most of the `draw_options` to `False` except for `'polygon'` (which in turn has `'allowIntersection'` set to `False`) ensures that the user can only draw polygons, and the polygons should not be able to self intersect.

app.py

```
from gradio_folium import Folium
```

app.py

```
# structure of the UI
with gr.Blocks() as app:
    map = Folium(value=foliumMap)
```

Code Snippet 3.67: Adding the Map to the GUI

To add this to the gradio interface, we must first import the `Folium` element from the `gradio_folium` library, and then add this to our `Blocks` interface. If we save this to a variable, `map`, we will be able to access and update it. We also have zoom in and out buttons in the top left as well.

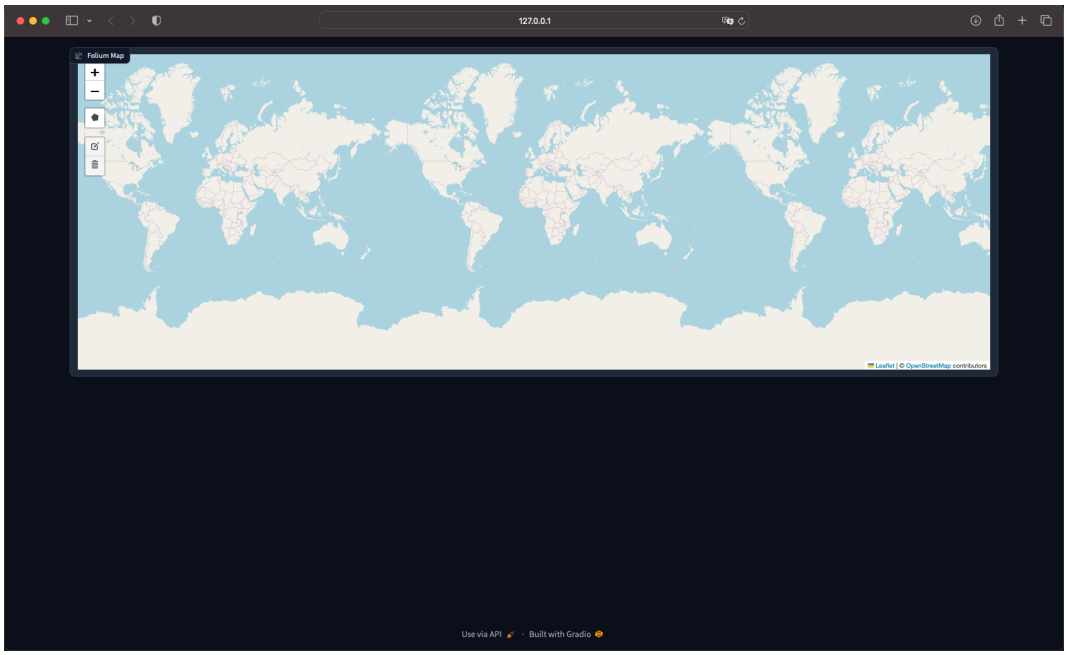


Figure 3.33: Map Added to GUI

Figure 3.33 shows the current output for the program, with the drawing tools shown in the top left of the map.

Users can also use the scroll wheel to zoom in and out of the map if they wish. By default, the mouse will drag around the map, but if the user clicks the pentagon icon, using the mouse will then begin to draw a polygon.

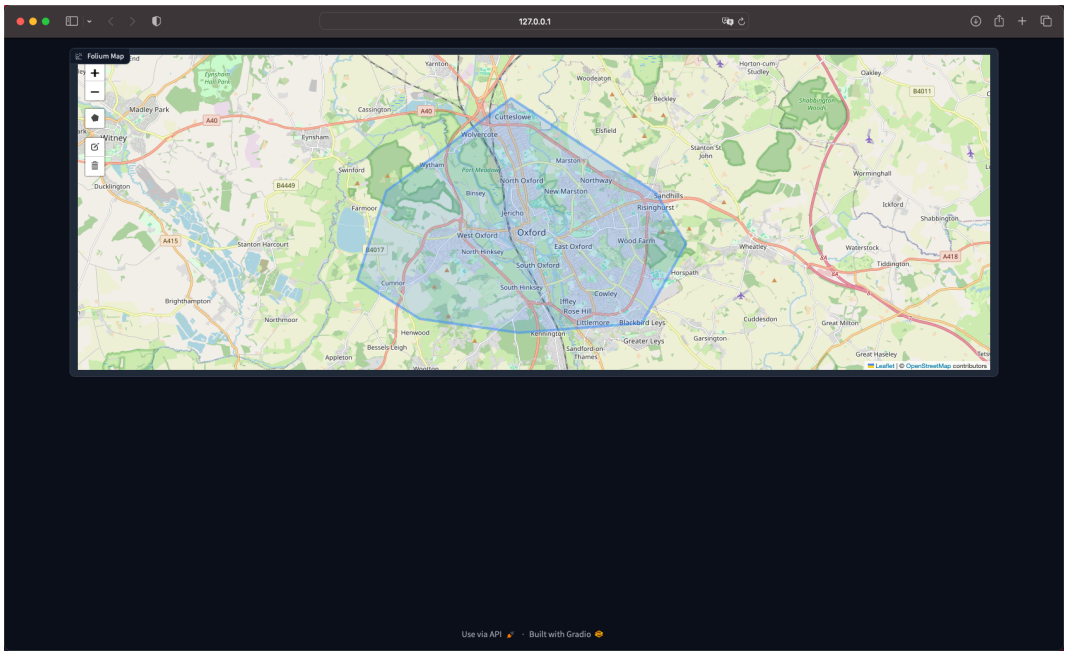


Figure 3.34: Drawing a Region

No.	Test	Input	Type	Expectation	Output
t_{46}	Map added to Interface	–	–	Gradio Interface with Map + Controlls	As Expected
t_{47}	Drawing on Map	–	–	Can only draw polygons	As Expected

Table 3.58: Testing Table for Code Snippets 3.66 & 3.67

Instead of the map starting very zoomed out, I would like for the map to start at a random city.

app.py

```
5  import random
6
7  # define some starting locations
8  locations = [[51.5360, -0.1196], [40.8093, -73.9678], [34.0069,
   ↪  -118.2304], [25.7617, -80.1918], [48.8488, 2.3470], [41.8840,
   ↪  12.4790], [52.5078, 13.4003], [37.9964, 23.7293], [59.3265, 18.0680],
   ↪  [40.4172, -3.6867], [41.3940, 2.1606], [38.7253, -9.1403], [55.7476,
   ↪  37.6209], [31.2230, 121.4860], [39.8958, 116.4000], [37.5467,
   ↪  126.9901], [35.6777, 139.7685], [1.3294, 103.8081], [-33.8782,
   ↪  151.1754], [30.0465, 31.2246], [6.5047, 3.3732], [-1.2873, 36.8198],
   ↪  [-33.9425, 18.5065],  [19.3916, -99.1279], [-23.5727, -46.6207],
   ↪  [36.1458, -115.1832], [38.8939, -77.0372], [49.2501, -123.1164],
   ↪  [43.6888, -79.4190]]
9
10 # initialise the map for drawing on
11 foliumMap = folium.Map(location=random.choice(locations))
```

Code Snippet 3.68: Setting the Initial Location to a Random City

Here, the `list` `locations` specifies the latitude/longitude coordinates of many famous cities across the world. To pick a random one, we must first import the `random` library and get a random choice from the `locations`. If we set the attribute `location` of the `Map` class to this, then the map will initialise at that location.

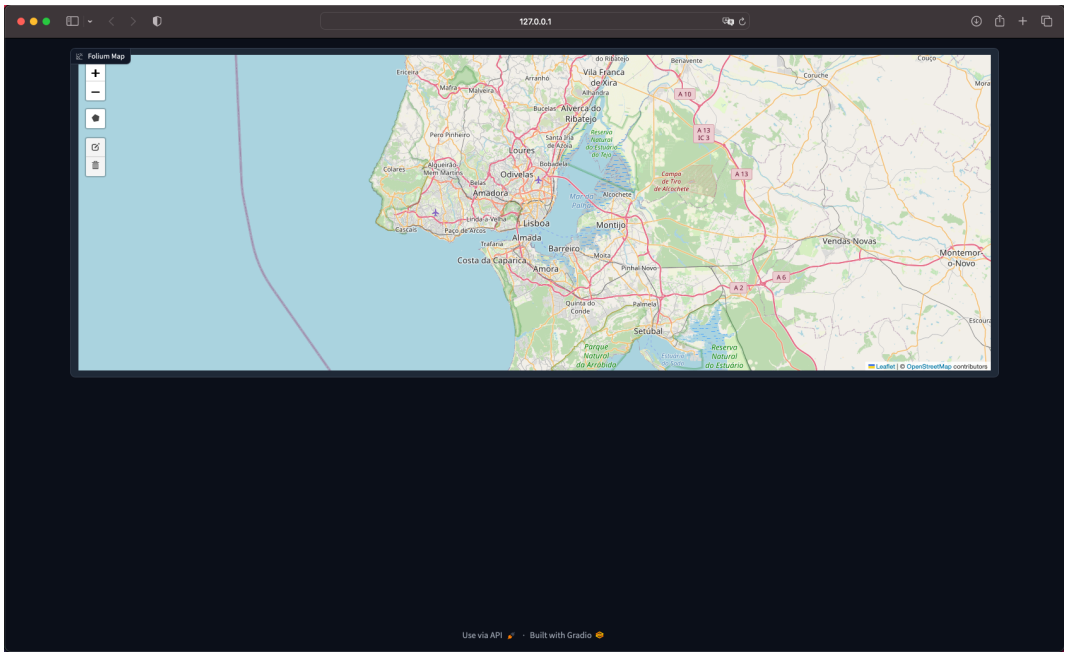


Figure 3.35: Map Starting at a Random City

No.	Test	Input	Type	Expectation	Output
t_{48}	Starting at a Random Location	–	–	Map Initialises at a Random City	As Expected

Table 3.59: Testing Table for Code Snippet 3.68

Now I need to retrieve the region the user draws on the map, in order to process it and make predictions. I did not know how to do this, so I began to research the problem.

After looking through many forums and StackOverflow pages and even asking ChatGPT, it seems that accessing the region drawn in the folium map dynamically is impossible.

`app.py`

```
13      export=True,
14      filename='region.geojson',
```

`app.py`

```
28  # structure of the UI
29  with gr.Blocks() as app:
30      map = Folium(value=foliumMap)
31      uploadButton = gr.UploadButton(label='Upload the GeoJSON file',
        ↪  file_count='single')
```

Code Snippet 3.69: Reading the Map for the Region

Instead, I have decided to set the `export` attribute (in `Draw`) to `True`, meaning that there will now be an export button on the top right of the map. This will download a `.geojson` file to the user's computer containing the region.

The user can then upload their `.geojson` file to be read and processed.

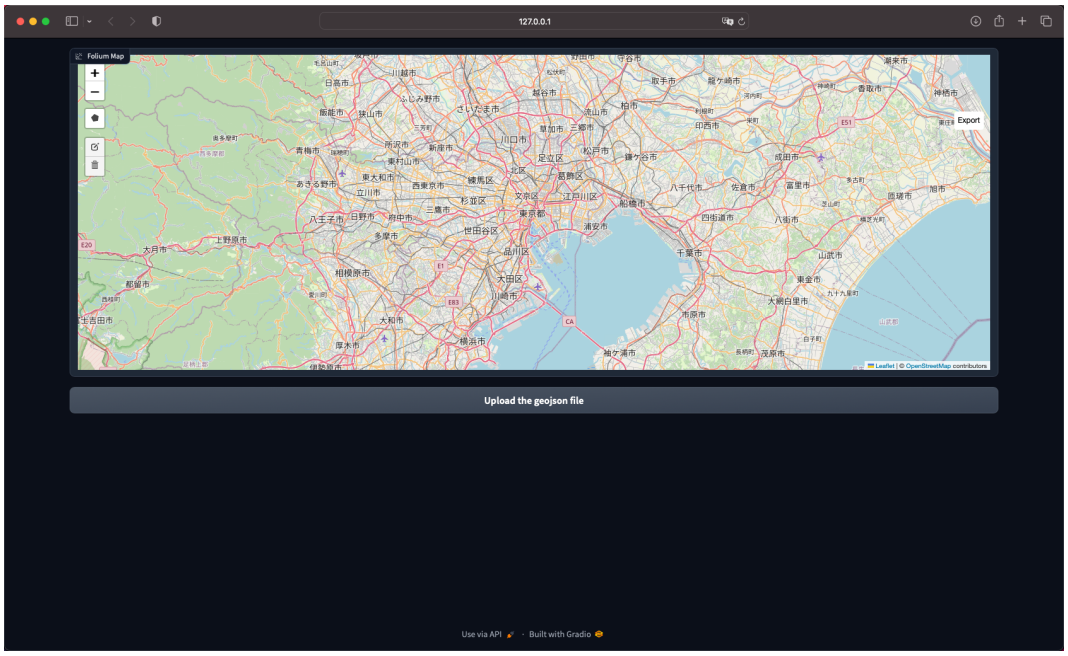


Figure 3.36: Upload Button Added

Figure 3.36 shows the output with these new changes. This is not an ideal solution, as the way the user uses the main function of the app has become a bit cumbersome. However, I cannot see a way around this, for now at least.

No.	Test	Input	Type	Expectation	Output
t_{49}	Export GeoJSON	–	–	User can export GeoJSON from map	As Expected

Table 3.60: Testing Table for Code Snippet 3.69

The button to export the `.geojson` file does not work and will be implemented in the next part.

3.3.3 Part 12 – Predicting the HDI

`app.py`

```
32    log = gr.Textbox(label='Log Messages', value='', interactive=False)
```

Code Snippet 3.70: Adding a Log Textbox

I first wanted to add a Log Textbox to the GUI, which will show messages when executing operations successfully. For this, we can use the gradio `Textbox`, with the `interactice` argument set to `False`. This will make it so that the user cannot write in the text box.

This was added below the other elements, and should appear below them on the screen.

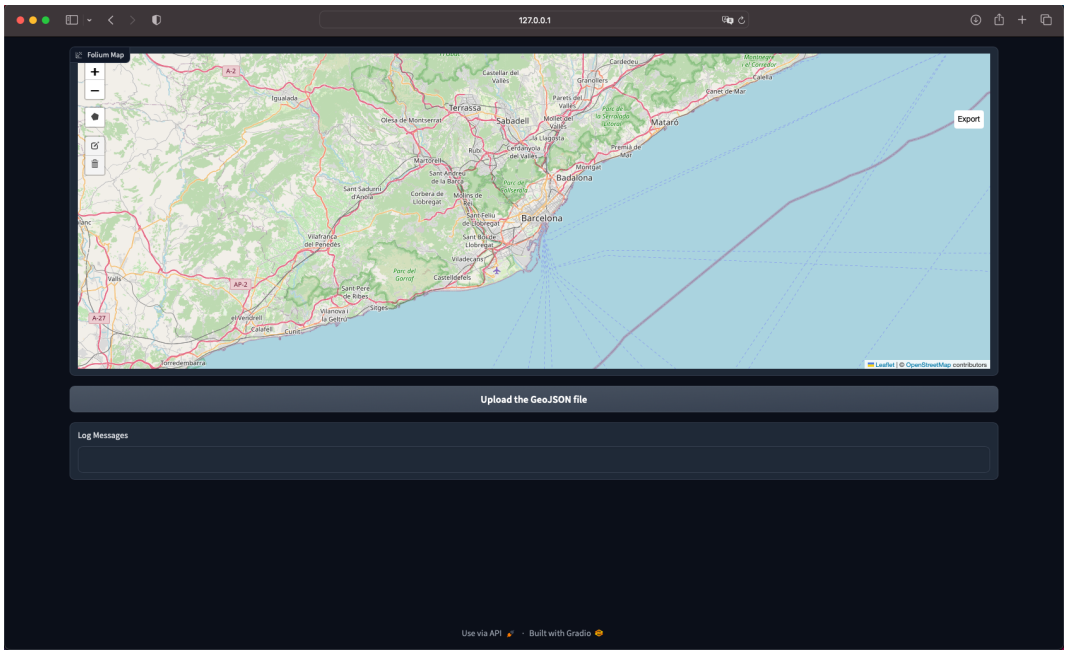


Figure 3.37: Log Textbox Added

No.	Test	Input	Type	Expectation	Output
t_{50}	Can't Write in Log Textbox	–	–	Log Textbox is read only	As Expected

Table 3.61: Testing Table for Code Snippet 3.70

Now we must add functionality to the upload button.

app.py

```
6  import json
7
8  # define global vars
9  currentRegion = []
10
11 # code to process the user uploading their geojson file
12 def uploadGeoJSON(filename):
13     global currentRegion
14     with open(filename, 'r') as file:
15         data = json.load(file)
16         currentRegion = [coordinate[::-1] for coordinate in
           ↪  data['features'][0]['geometry']['coordinates'][0]]
17     return f'Successfully Uploaded {filename[filename.rindex("/")+1:]}'
18
```

app.py

```
46     # functionality
47     uploadButton.upload(uploadGeoJSON, inputs=[uploadButton],
       ↪  outputs=[log])
```

Code Snippet 3.71: Adding Functionality to the Upload Button

All of the functionalities in the gradio interface will be of the format: `component.action(function, inputs=[...], outputs=[...])`, where the inputs and outputs are lists of components, and still contained within the `with gr.Blocks()` statement. Therefore, the code to make the upload button interactive is that shown on line 47, where `uploadGeoJSON` is the identifier of some function.

To store the current region, we can use a `global` variable, `currentRegion`. Recalling how `.geojson` files are structured, to get the coordinates into the right format, we must reverse each coordinate before storing it.

Once we have read the contents of the uploaded file, we can return a string to be displayed in the log textbox. The filename given by the parameter will be its full path, so we only want to take a substring of that starting with the character after the last `'/'`.

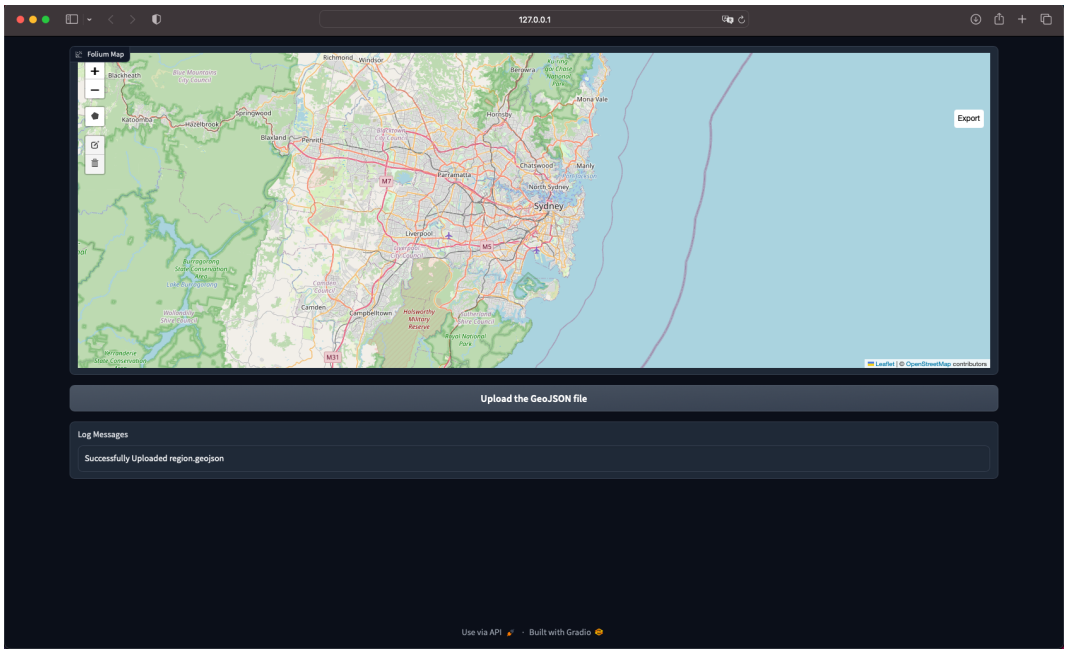


Figure 3.38: Uploading GeoJSON Files

No.	Test	Input	Type	Expectation	Output
t_{51}	Upload GeoJSON Files	`region.geojson`	–	Coordinates saved to `currentRegion`, and appropriate message shown	As Expected

Table 3.62: Testing Table for Code Snippet 3.71

app.py

```
40 # structure of the UI
41 with gr.Blocks() as app:
42     with gr.Row():
43         with gr.Column(scale=3):
44             map = Folium(value=foliumMap)
45             uploadButton = gr.UploadButton(label='Upload the GeoJSON
                ↪  file', file_count='single')
46         with gr.Column(scale=1):
47             predictHDIbutton = gr.Button(value='Predict HDI',
                ↪  variant='primary')
48             HDIprediction = gr.Textbox(label='I believe that the HDI of
                ↪  this region is...', value='', interactive=False)
49     log = gr.Textbox(label='Log Messages', value='', interactive=False)
```

Code Snippet 3.72: Placing the Predict Button

In order to place the prediction button as shown by the Interface Sketch (Figure 2.30), we must make use of gradio `Rows` and `Columns`.

As we want out map and upload button to be arranged vertically, we can place them in a `Column`. The same goes fro the button to predict HDI and the output textbox for the HDI prediction. As we want these two columns to be arranged side by side, we can put them in a `Row`. As the log textbox should be below everything, it should be placed outside of the `Row`. In my opinion, this use of context managers (`with gr.Row()` and `with gr.Column()`) makes the code extremely readable, and easy to understand where elements will end up.

If we want the width of the first column (containing the map) to be larger than the width of the second, we can use the `scale` attribute, to specify the ratio of the widths. Here, I have specified that the width of the map column should be 3 times the width of the prediciton column.

For the actual button, setting the `variant` attribute to `'primary'` makes the button appear in the primary colour of the app, which is orange.

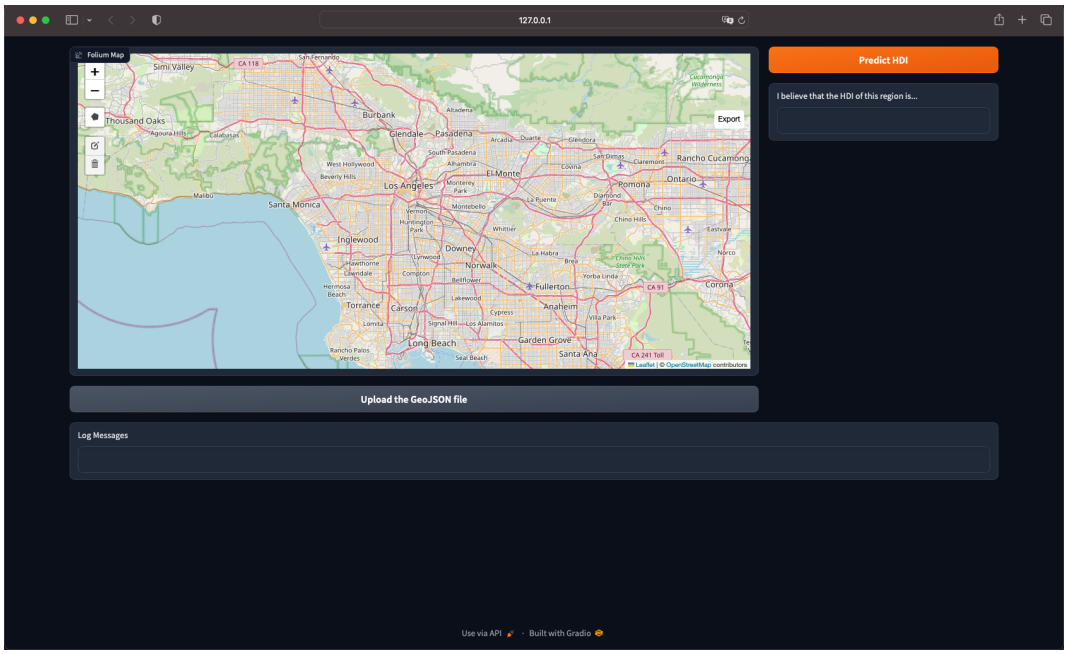


Figure 3.39: New Structure of the GUI

No.	Test	Input	Type	Expectation	Output
t_{52}	Column Strucure of GUI	–	–	Width of map is 3 times the width of the prediction button	As Expected

Table 3.63: Testing Table for Code Snippet 3.72

Now it is time to implement the functionality of the predict HDI button.

`app.py`

```
19 # predict the HDI of the given region
20 def processPrediction(progress=gr.Progress()):
21     global currentRegion
22     if currentRegion == []:
23         return 'You have not uploaded a region!'
24     progress(0, desc='Starting...')
```

Code Snippet 3.73: Processing the Prediction

Again, we must define a funciton, to pass into the gradio interface, which will carry out the required processes.

We must first determine if the user had in fact uploaded a region. If they haven't, it will still be an empty `list`, and so we should return an error message before any calculations are carried out.

I imagine this function will take a lot of time to run, and so I would like for it to show a progress bar to the user. We can do this by using a `gr.Progress()` bar. This allows us to set the progress bar to different percentages at different times within the function (as shown on line 24).

The rest of this function will be very similar to stuff we have written before, mainly `def getAllFactors`, with some small changes.

app.py

```
7  from data_processing.get_osm_data import get_osm_data
```

app.py

```
11 all_objects = {}
```

app.py / def processPrediction

```
23     global all_objects
```

app.py / def processPrediction

```
27     # retrieve all relevant osm data
28     object_types = ['house', 'school', 'hospital', 'pharmacy',
       ↪  'restaurant', 'place_of_worship', 'bank', 'slot_machines',
       ↪  'fast_food', 'toilets', 'police', 'university', 'library',
       ↪  'post_box', 'vending_machine', 'bench', 'tree']
29     all_objects = {}
30     for num, object_type in enumerate(object_types):
31         progress(0.3*(num/len(object_types)), desc=f'Downloading
           ↪  {object_type} data...')
32         print(f'getting {object_type} data...') # log message
33         all_objects[object_type] = get_osm_data(object_type,
           ↪  currentRegion)
```

Code Snippet 3.74: Retrieving All Relevant OSM Data

The first thing we must do is download all of the different objects from OpenStreetMaps, by calling the `def get_osm_data` function written in Section 3.1.3. We must also initialise a global dictionary to contain all of these objects, as we will need to use them later, to improve the HDI.

Again, we can use the `progress` function to set the length of the progress bar according to how far through the list of object types to download we are.

No.	Test	Input	Type	Expectation	Output
t_{12}	Collecting all objects	–	–	Each list saved in a `dict`	As Expected

Table 3.64: Testing Table for Code Snippets 3.73 & 3.74

app.py

```
8  from calc_factors import averageDistance
```

app.py / def processPrediction

```
35      # calculate average distances
36      allFactors = []
37      averageDistanceFactors = [['house', 'school'], ['house', 'hospital'],
        ↪  ['house', 'pharmacy'], ['house', 'restaurant'], ['school',
        ↪  'hospital'], ['police', 'hospital'], ['house',
        ↪  'place_of_worship'], ['bank', 'slot_machines'], ['fast_food',
        ↪  'toilets'], ['house', 'police'], ['university', 'library'],
        ↪  ['house', 'library']]
38      for num, averageDistanceFactor in enumerate(averageDistanceFactors):
39          progress(0.3+0.6*(num/len(averageDistanceFactors)),
            ↪  desc=f'Calculating Average Distance between
            ↪  {averageDistanceFactor[0]} and {averageDistanceFactor[1]}...')
40          print(f'finding A({averageDistanceFactor[0]},
            ↪  {averageDistanceFactor[1]})...') # log message
41          allFactors.append(averageDistance(
            ↪  all_objects[averageDistanceFactor[0]],
            ↪  all_objects[averageDistanceFactor[1]]))
```

Code Snippet 3.75: Calculating $A(x, y)$ Factors

Next, we must calculate each of the average distance factors. I have defined a `list` of the pairs of each object type, and so they can just be fed into the function to calculate the average distance. We can again use the `progress()` function to update the progress bar.

At this point, I decided to make a copy of the `calc_factors.py` file, and placed it in the main directory, instead of the `data_processing` directory. This is because I plan to make small edits to how some of these functions are going to work.

app.py

```
8  from calc_factors import averageDistance, calcArea, density
```

app.py / def processPrediction

```
42      # calculate area
43      progress(0.9, desc='Calculating Area...')
44      print('calculating area...') # log message
45      area = calcArea(currentRegion)
46      # calculate density factors
47      densityFactors = ['school', 'hospital', 'pharmacy', 'police',
        ↪  'library', 'toilets', 'restaurant', 'place_of_worship',
        ↪  'post_box', 'vending_machine', 'bench', 'tree']
48      for num, densityFactor in enumerate(densityFactors):
49          progress(0.91+0.05*(num/len(densityFactors)), desc=f'Calculating
            ↪  {densityFactor} Density...')
50          print(f'finding D({densityFactor})')
51          allFactors.append(density(all_objects[densityFactor], area))
```

Code Snippet 3.76: Calculating $D(x)$ Factors

We can also calculate the density factors in a similar way, by defining a `list` of all the factors, and iterating over them. We must first however calculate the area of the region, to pass into the `def density` function.

No.	Test	Input	Type	Expectation	Output
t_{13}	Each factor is calculated	–	–	We have a `dict` of all the factors of a given region	As Expected

Table 3.65: Testing Table for Code Snippets 3.75 & 3.76

Now that we have a full list of factors, we can predict the HDI

app.py

```
import numpy as np
from neural_network import MultilayerPerceptron

# initialise neural network
network = MultilayerPerceptron([24, 18, 12, 6, 3, 1])
network.load_model('models/hdi.pkl')
```

app.py / def processPrediction

```
    # predict HDI
    progress(0.96, desc='Predicting HDI...')
    inputLayer = np.matrix([[100 if factor == None else float(factor)/10
    ↪  if index <= 11 else float(factor)] for index, factor in
    ↪  enumerate(allFactors)])
    prediction = round(network.predict(inputLayer).item(0,0), 3)
    return prediction
```

Code Snippet 3.77: Predicting the HDI

For this, we must first import the `MultilayerPerceptron` from `neural_network.py`, and set the layer sizes to the right values. We can then use the `load_model` method to load the model we trained in Section 3.2.6.

Then, at the end of the `def processPrediction` function, we can get a matrix for the input layer (dividing the average distance factors by 10 in the same way as done before), and then we can make a prediction, to be returned.

app.py / def processPrediction

```
    predictHDIbutton.click(processPrediction, inputs=[],
    ↪  outputs=[HDIprediction])
```

Code Snippet 3.78: Adding Functionality to the Prediction Button

To make the button actually work, we must add a gradio functionality. This line of code makes it so that when the `predictHDIbutton` is `click`ed, it calles the `processPrediction` function, passing in no inputs (as the region is already a global variable), and outputting to the `HDIprediction` textbox.

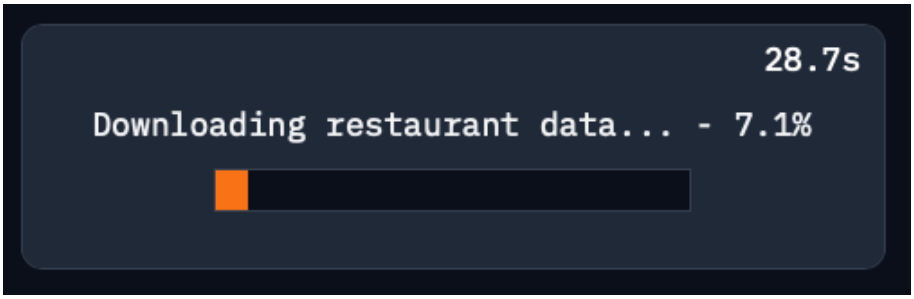


Figure 3.40: Progress Bar

Figure 3.40 shows the progress bar as the prediction is taking place. Gradio, by default, also provides a timer for these processes (as shown in the top right).

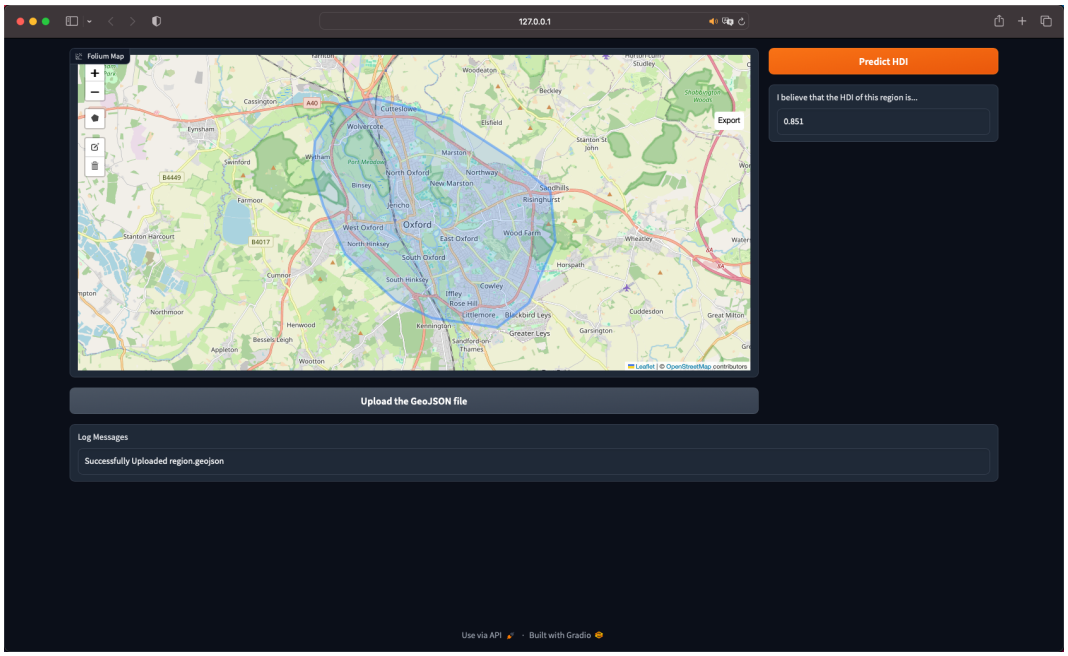


Figure 3.41: Successful Prediction from a Region

Figure 3.41 shows the prediction for Oxford, whici is pretty close.

No.	Test	Input	Type	Expectation	Output
$T_{6.1}$	Overall Flow	User Draws Region and then Predicts	Valid	Factors are calculated, HDI is predicted and displayed	As Expected
$T_{6.2}$	Overall Flow	User attempts to predict before uploading a region	Invalid	Error is thrown in the prediction output	As Expected

$T_{6.3}$	Overall Flow	User Draws Region and doesn't predict	Valid	Region is saved, Factors are never calculated and HDI is never predicted	As Expected

Table 3.66: T_6 Testing Table (for Code Snippets 3.77 & 3.78)

Unfortunately however, this took about 9 minutes to fully complete, which is too much time to wait. By looking at the progress bar, I found that the majority of the time taken was spent in calculating the average distance factors. Therefore, I plan to let the user change the `max_x_objects` variable with a slider, along with a new `max_y_objects` variable, which must first be implemented.

`calc_factors.py`

```
18 # finds the average distance between 2 types of objects, A(x,y)
19 def averageDistance(x_objects, y_objects, max_x_objects=2000,
   ↪  max_y_objects=2000):
```

`calc_factors.py / def averageDistance`

```
58         if len(y_objects) > max_y_objects:
59             y_objects_in_consideration = random.sample(y_objects,
               ↪  max_y_objects)
60         else:
61             y_objects_in_consideration = y_objects
62         for y_object in y_objects_in_consideration:
63             dist_to_ys.append(distBetween2Points(x_object, y_object))
```

Code Snippet 3.79: Implementing a Maximum for y-objects

In the `def averageDistance` function (in the new `calc_factors.py` file), we can pass in `max_y_objects` in as a parameter, with a default value of 2000. When we iterate for each `x_object`, we can random sample the `y_objects` if nescessary.

This means that we are no longer finding the absolute Average Shortest Distance from x to y, as the y-objects we select may not include the closest one to the current x-object. However, this may actually make the metric more realistic, for a reason which is best shown with an example. The closest school to my house is my primary school, which I no longer attend. Therefore, when I travel to school, I must travel further than the distance from my house to the closest school. This would be reflected in the code by the fact that my primary school may not be selected in the random sample of all the schools (y-objects in this example).

No.	Test	Input	Type	Expectation	Output
t_{53}	Maximum y-objects	Set of y-objects	–	Random sample of y-objects for each x-object	As Expected

Table 3.67: Testing Table for Code Snippet 3.79

app.py

```
28 # predict the HDI of the given region
29 def processPrediction(max_x_objects, max_y_objects,
   ↪  progress=gr.Progress()):
```

app.py / def processPrediction

```
48         allFactors.append(averageDistance(
           ↪  all_objects[averageDistanceFactor[0]],
           ↪  all_objects[averageDistanceFactor[1]], max_x_objects,
           ↪  max_y_objects))
```

Code Snippet 3.80: Passing in the `max_y_objects`

To then take these variables into account when calculating factors, we must pass the `max_x_objects` and `max_y_objects` variables into the `def processPrediction` function.

Then, inside that function, we must pass them into the the `def averageDistance` function.

app.py

```
93              predictHDIbutton = gr.Button(value='Predict HDI',
                ↪  variant='primary')
94              with gr.Group():
95                  max_x_objectsSlider = gr.Slider(minimum=10, maximum=5000,
                    ↪  value=500, label='Maximum x-buildings', info='When
                    ↪  calculating the average distance between 2 types of
                    ↪  building, it will find the average of this many pairs.
                    ↪  Increasing this value may make the prediction take
                    ↪  much longer, but may make the prediction more
                    ↪  acurate.')
96                  max_y_objectsSlider = gr.Slider(minimum=10, maximum=5000,
                    ↪  value=500, label='Maximum y-buildings', info='When
                    ↪  calculating the average distance between 2 types of
                    ↪  building, it will find the closest of the second type
                    ↪  of building out of this many options. Increasing this
                    ↪  value may make the prediction take much longer, but
                    ↪  may make the prediction more acurate.')
97              with gr.Group():
98                  HDIprediction = gr.Textbox(label='I believe that the HDI
                    ↪  of this region is...', value='', interactive=False)
99                  similarHDI = gr.Textbox(label='That\'s a similar HDI
                    ↪  to...', value='', interactive=False)
```

Code Snippet 3.81: Adding Sliders

To let the user decide the values for `max_x_objects` and `max_y_objects`, we can add some sliders. For each of them, I have set the minimum value to 10 and the maximum value to 5000, with a default value of 500. They also each come with an information label, as their function may not be clear to the user.

The use of the `gr.Group()` block will place these elements together, as shown in Figure 3.42. I have also added an output textbox for the similar HDI region, but I will not be implementing this until the 2nd prototype.

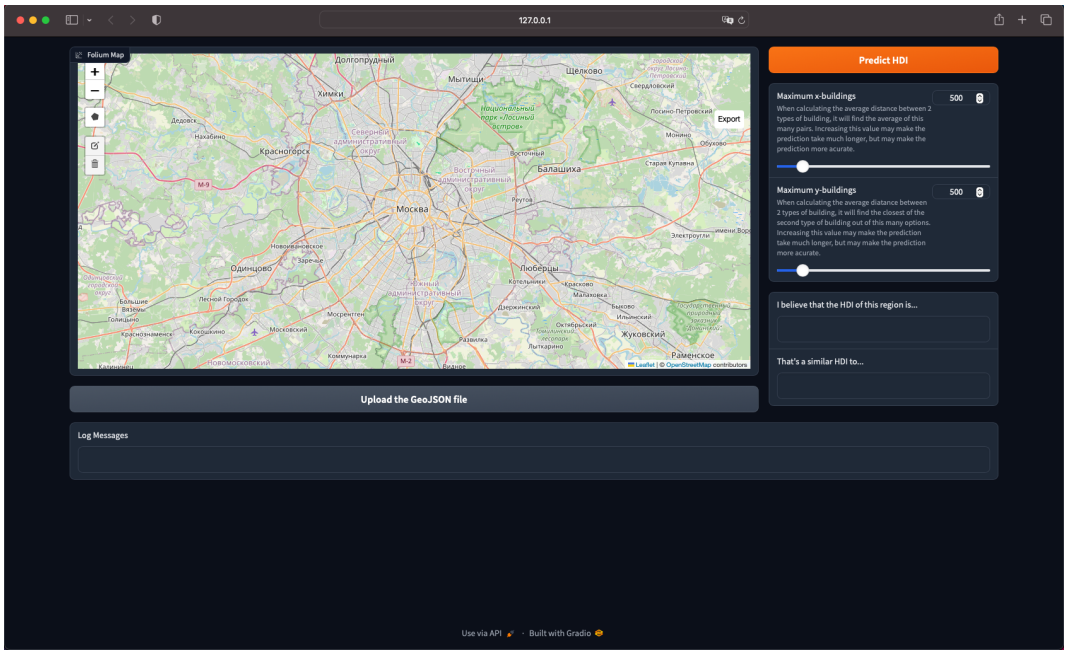


Figure 3.42: `max_x_objects` and `max_y_objects` Sliders

`app.py`

```
104     predictHDIbutton.click(processPrediction, inputs=[max_x_objectsSlider,
        ↪  max_y_objectsSlider], outputs=[HDIprediction])
```

Code Snippet 3.82: Updating Functionality for Predicting HDI Button

To take these slider values into account, we must now pass them as inputs to the `predictHDIbutton.click` method.

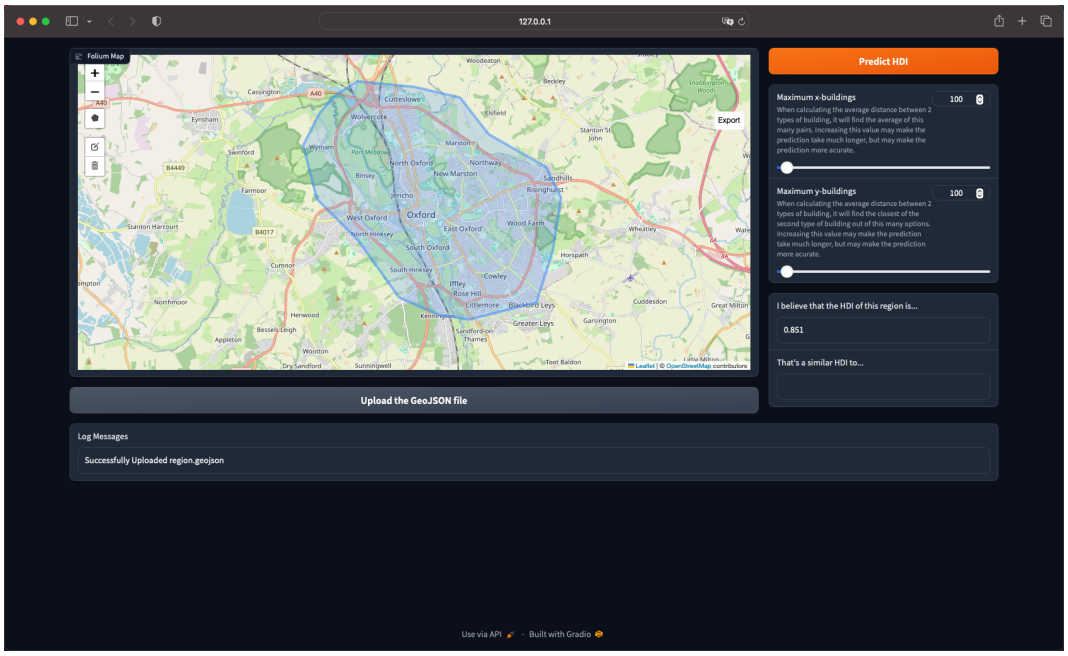


Figure 3.43: Prediction with lower `max_x_objects` and `max_y_objects` values

No.	Test	Input	Type	Expectation	Output
t_{54}	Prediction with lower `max_x_objects` and `max_y_objects` values	`max_x_objects`, `max_y_objects`	–	Less Accurate Values for $A(x, y)$, still gives accurate prediction	As Expected

Table 3.68: Testing Table for Code Snippets 3.80, 3.81 & 3.82

This prediction 1 minute instead of 9, 50 seconds of which were spent downloading the locations of all the objects, which I have no control over how long it takes.

3.3.4 Part 13 – Suggesting Improvements

...

3.3.5 Review

...

3.4 Feedback from Stakeholders

...

4. Design of the 2nd Prototype

4.1 Structure Diagram

...

4.2 Authentication System

4.2.1 Storing Data & Encryption

...

4.2.2 Creating an Account & Logging In

...

4.2.3 Saving Regions

...

4.2.4 List of Key Variables

...

4.3 User Interface

4.3.1 Interface Sketch

...

4.3.2 Input Validation

...

4.3.3 Usabilty Features

...

4.3.4 Analysis of how much Each Factor affects the Others

...

4.3.5 Manually Entering Factors

...

4.3.6 Comparison to Similar Regions

...

4.3.7 Improved Algorithm for Finding an Optimal Place for a New Building

...

4.3.8 List of Key Variables

...

4.4 Data

4.4.1 Data Structures

...

4.4.2 Testing Data for Development

...

4.4.3 Testing Data for Post-Development

...

5. Developing the Coded Solution of the 2nd Prototype

5.1 Milestone 4 – Authentication System

5.1.1 Success Criteria

...

5.1.2 Part 14 – The Next Bit

...

5.1.3 Review

...

5.2 Milestone 5 – Final User Interface

5.2.1 Success Criteria

...

5.2.2 Part 15 – The Next Bit

...

5.2.3 Review

...

5.3 Feedback from Stakeholders

...

6. Evaluation

...

List of Figures

List of Code Snippets

List of Tables

Bibliography

[1] sandeeoppondru. human-development-index. https://github.com/sandeepponduru/human-development-index, 2023.

[2] julieanneco. predictinghdi. https://github.com/julieanneco/predictingHDI, 2020.

[3] Human Development Reports. country-insights. https://hdr.undp.org/data-center/country-insights#/ranks, 2022.

[4] Global Data Lab. Subnational hdi version archive. https://globaldatalab.org/shdi/archive/, 2024.

[5] Global Data Lab. Gdlcode shapefiles. https://globaldatalab.org/mygdl/downloads/shapefiles/, 2024.

[6] Grant Sanderson. Neural networks – chapters 1-4. https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi, 2017.

[7] Micheal Nielsen. Neural networks and deep learning – chapter 2. http://neuralnetworksanddeeplearning.com/chap2.html, 2019.

[8] alexisdevarennes. Shapefile into geojson conversion python 3. https://stackoverflow.com/questions/43119040/shapefile-into-geojson-conversion-python-3, 2020.

[9] Micheal Nielsen. neural-networks-and-deep-learning. https://github.com/unexploredtest/neural-networks-and-deep-learning, 2022.

[10] GeeksforGeeks. Python - import from parent directory. https://www.geeksforgeeks.org/python-import-from-parent-directory/, 2024.