Report for

AstraZeneca AI Challenge

Shaastra 2021

Team AI Penguins

Omkar Nath

Shreya Nema

Uma T V

Contents

Problem Statement

The given problem statement is to build a program that is able to automatically reverse engineer a Kaplan Meier Chart. Kaplan Meier Chart is the visual representation of depreciating or appreciate probability of an event with respect to time interval.

The objective is that given a Kaplan Meier Chart, the program should be able to analyse the graph and output the raw data points from it. The input format is an image of the chart, and the expected output is a csv/excel file consisting of the x-y data points on the chart.

For the purposes of this, we are given five links (titled Image 1 to Image 5) which can be used to test the algorithm.

Approach

To solve the given problem, our approach is to use a combination of image analysis techniques, and machine learning techniques, each with its specific applications.

The broad approach is to identify key areas of the graph (axes, characters, data points) using image analysis, and use the machine learning for specific applications such as interpreting the textual data.

As per this, the broad steps involved in the algorithm are as follows

1. Identify the unique colour groups used in the graph
2. Locate the x and y axes in the image
3. Locate the intersection of the axes (not necessarily the origin)
4. Locate the various points along the axes where values are written using text.
5. Identify the distance and positions of these for getting the scale later.
6. Locate the regions of text corresponding to marked values on the x axis and the y axis
7. Isolate each individual value of text along the axes
8. Use some technique to read the text and obtain the values
9. Use the values to calculate the difference between the above markings
10. Also use values to locate the origin of the graph
11. Use 5,9,10 to define a transformation function between the two spaces.
12. Identify all possible curves along the graph
13. Track each curve to get all the data points corresponding to it
14. Filter the curve data
15. Apply the transforms to the curves to get the coordinates in x-y space
16. Save the data to a csv file for each curve.

Intuition behind the approach

Any image can be represented as a grid of pixels, where each pixel is a discrete unit that has (r,b,g) values, all varying between 0 and 255. These values represent the colour of the pixel. As it is a grid, each pixel has an associated location with it in a 2-dimensional space of its own, i.e., its own coordinates. Henceforth I will refer to this 2-dimensional space as "Pixel Space".

Similarly, the graph itself is meant to represent a 2-dimenstional space, as per the coordinate values of the two axes, x and y. Henceforth I will simply refer to this as "actual coordinate space", or coordinate space for short.

Most important aspect is that the proportions between the two spaces are kept constant when constructing such graphs, to maintain visual consistency. This fact is what is exploited in our approach.

In the given image, all the information is given in pixel space, through the use of colour values. If we can somehow interpret the information in this, we can use the positions of the curves in the pixel space to convert to coordinate space by creating a transformation between the two spaces.

That is what our approach tries to do. First, we analyse the image to understand various aspects of it. Subsequently we create a transformation to convert the information from pixel space to coordinate space. This is then used to give us our desired results.

Reasons for this Approach

There are many reasons for choosing this approach. These are as highly below:

- A graph in general is defined by a fixed set of rules. This can be exploited to create a system which is able to interpret the rules, to reverse engineer the original graph
- As many graphs are possible, there is a high chance of variation between graphs. This means that an algorithm trained on some may not work on others.
- An algorithm that utilizes the fundamental rules however will be able to generalize well to any graph, which means the graph is highly adaptable
- There is no need to train the algorithm and procure any training data
- This method will overall be much more efficient as we just need to do a minimalistic amount of processing.

Key Assumptions of the Algorithm

There are some key assumptions without which the algorithm won't work. These are as highlighted below:

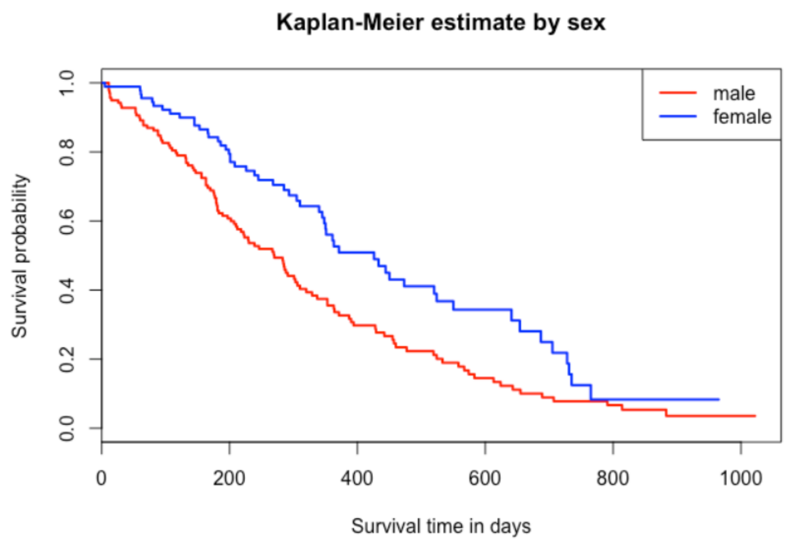
- The given image has x axis and y axis aligned with the borders of the image (so the image is not rotated basically)
- The graph occupies the majority of the image. Its not like the graph needs to be extracted from a lot of information in an image
- There are markings along each of the axes for some of the points on the axis
- There are textual values corresponding to the markings that highlight the value of that position

In addition to these, there are certain assumptions made in the current version of the algorithm. This can be resolved however by improving upon the algorithm:

- The background is white
- The axes are black
- The x axis text is horizontal
- The y axis text can be horizontal, or vertical with the base towards the right
- Each curve has a unique and different colour.
- The markings are connected to the axes
- There is a slight gap between the axes markings and their textual values

Loading the Image

For the purposed of this report, let us assume we are working with the following image. (Image 4-2)



The various steps of the algorithm will be visualized by running the code on this graph.

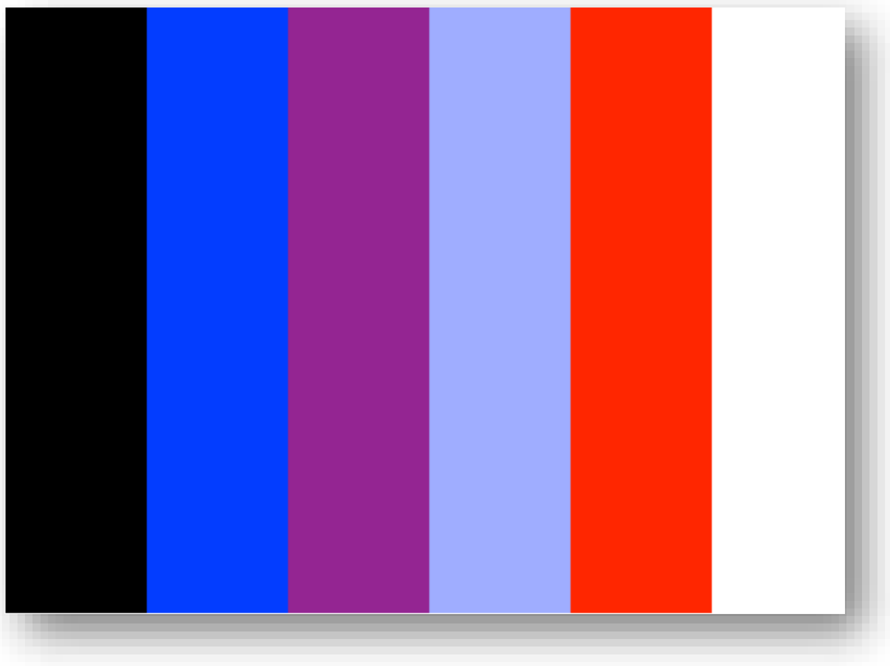
Analysing the Colours

One common problem noticed in a lot of the graphs is that each colour doesn't have a unique value. For example, even though one of the curves here is red, there are various shades of red that are used throughout the curve, which makes is more difficult to track. Similarly, for the background white etc.

To solve this problem, we want to identify the key colour "groups" that occur in the graph, around which most of the values cluster. Then based on which cluster the pixel is closer to, it can be classified as that colour.

The first step is to create a frequency distribution. This is done by reducing the 255x255x255 possible values into 6x6x6 possible values. Then a frequency distribution is created over the entire image, for every pixel values.

Local maximums are then taken as the colour centres for the colour groups. This can be visualised for our example as:



Then based on which colour group a pixel is closest towards, it is assigned that value. So all shades of red can be treated as a single red. This makes all our calculations from this point onwards much easier.

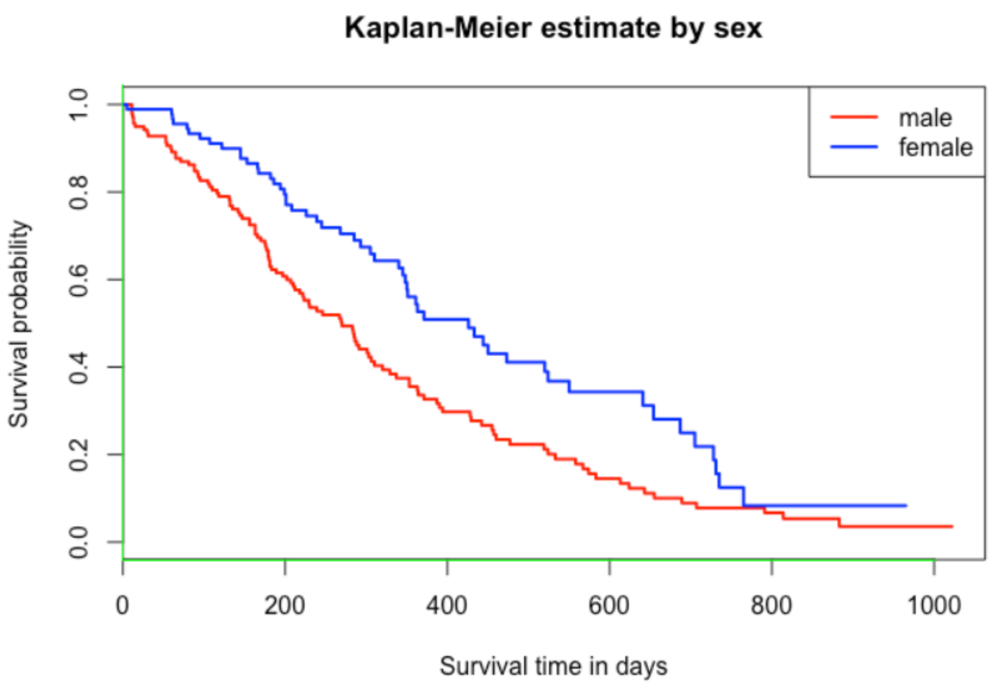
Locating the Axes

Next step is to locate the x axes and the y axes in the graph. This is done by finding a line along the x direction, which has a continuous strip of black pixels, of a minimum certain length based on the image dimensions. Similarly, for the y axis. The pixel coordinates of these axes are then noted.

To ensure that we don't capture any borders outside of the graph, the search starts from the centre of the graph.

Additionally, the length of these axes is also noted i.e., the start and the end points.

This is visualized in green as below:



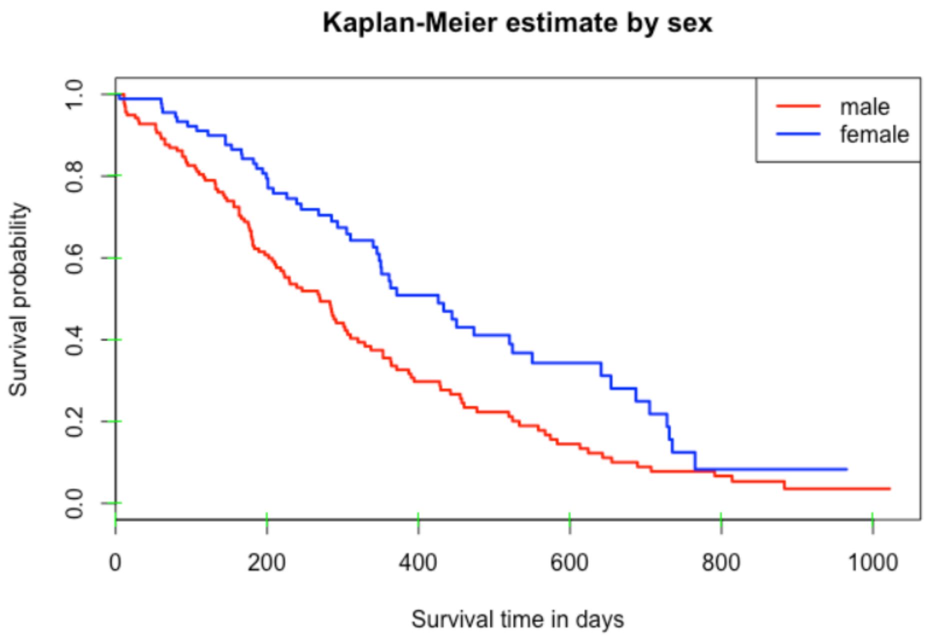
Finding the Axes Markings

Next step is to find the markings along the axes. For this, we start moving outwards from the axes until there are no more markings present (i.e., an entire row of white pixel space, before the textual values). Then we back trace a few pixels. Finally, the place where there are black pixels are noted and saved as the markings.

It is done in this manned so that we ignore any half markings that may be there, with no associated character values.

The pixel coordinates for each of these markings are saved for future use.

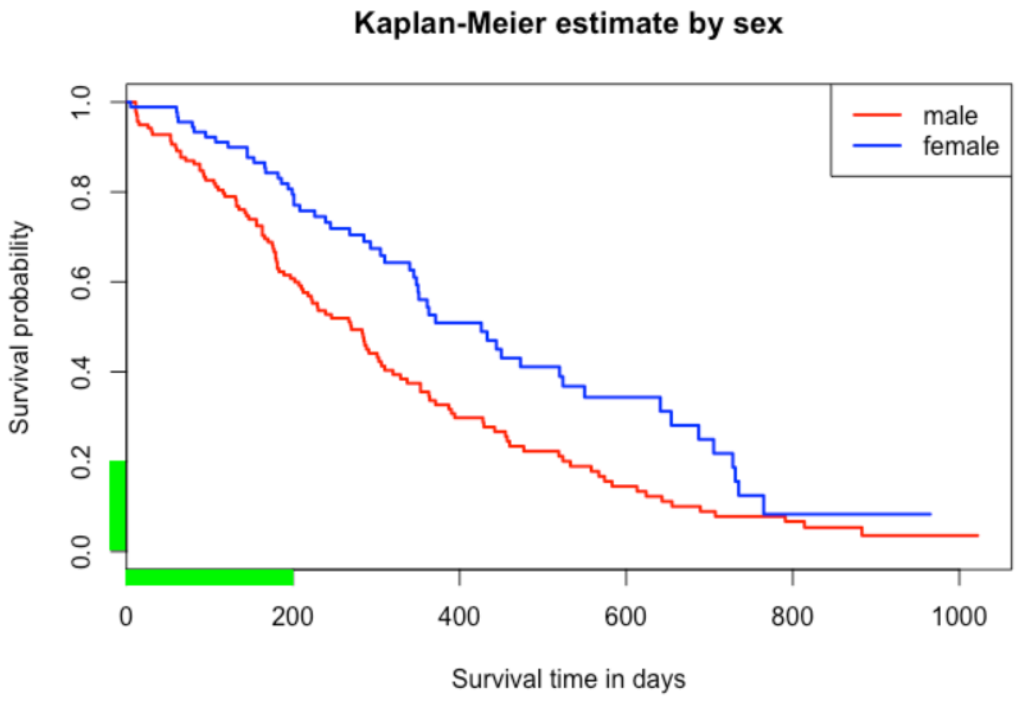
These can be visualized as below:



Pixel Space Distance for Scale

The distance between two divisions is taken by using the markings positions as above. This is taken by averaging over the entire axes, as taking difference between two markings may be inaccurate as pixels forcible have to take integer coordinates.

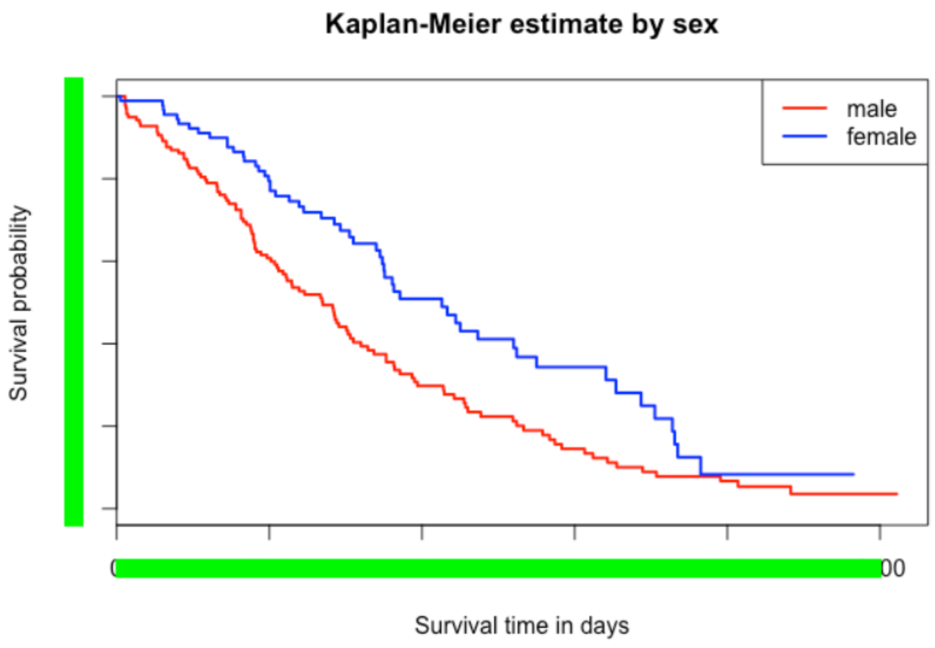
This is visualised as below:



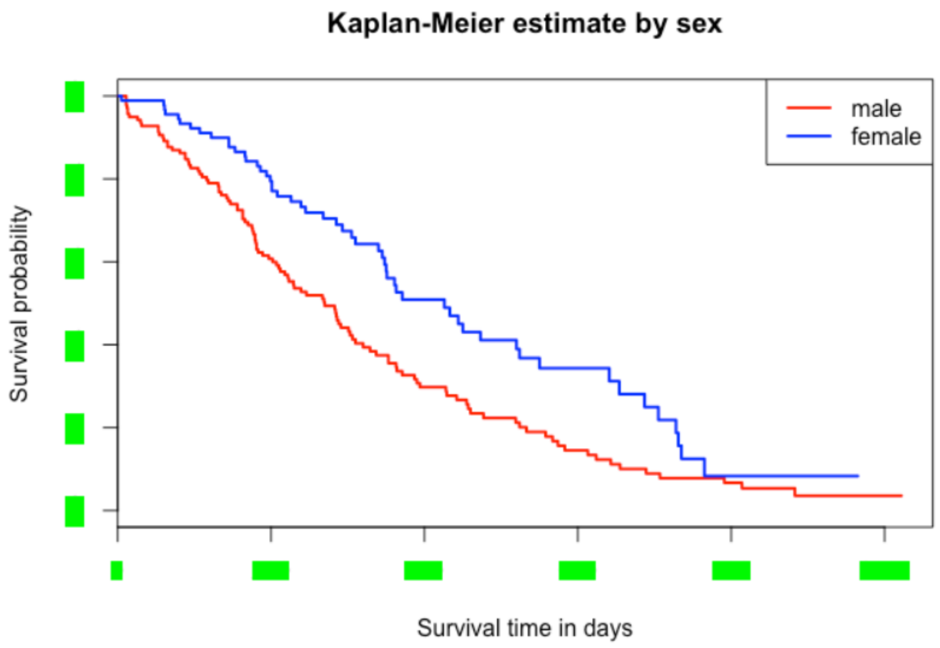
Locating the text of values along axes

Next step is to locate the text values along the x and the y axes. For this, it is assumed that just after the markings, there is white space located above and below the text. This white space is used as the guides for locating the top and bottom limits for the text.

A threshold value is considered in case there is also a separation between digits. The results of this have been visualized as below:



Next is the left and right limits. This again can be found by simply seeing where the black pixel text ends. This can even be taken a step further to isolate each unique value, as per the white space located in between. The isolated characters can be visualised as:



Optical Character Recognition

The next step is to read the text to be able to identify what the values are. There can be many ways to do this.

The method we have chosen here is to use pytesseract package. The textual data is fed as an image as show below to pytesseract.

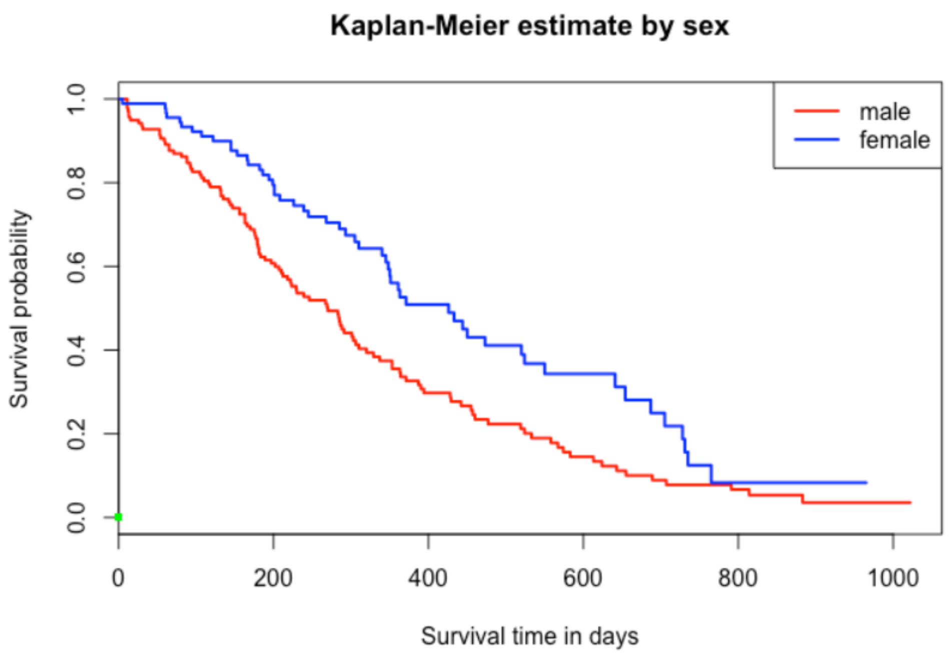
0.0 0.2 0.4 0.6 0.8 1.0

The numerical values are obtained from pytesseract. This is used to calculate two key things. Firstly, the difference between two consecutive values to establish a scale. Secondly location of a specific point to find the origin. While estimating the difference value, we calculate all possible pairs values and take the mode, as pytesseract results are not very accurate.

For y axis, as additional measure of rotating by 270 degrees is tried. Based on the results from pytesseract, the appropriate configuration is chosen.

Note that pytesseract is not the best method to use here. It would be better to use a more specific digit recognition model such as a CNN. This is even more easier considering the individual values have already been isolated.

Based on these, the origin is marked as shown below:

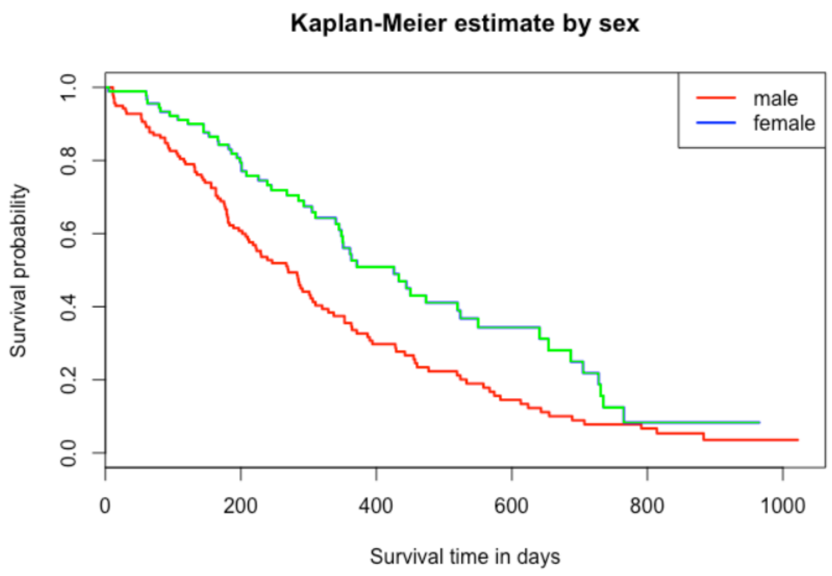


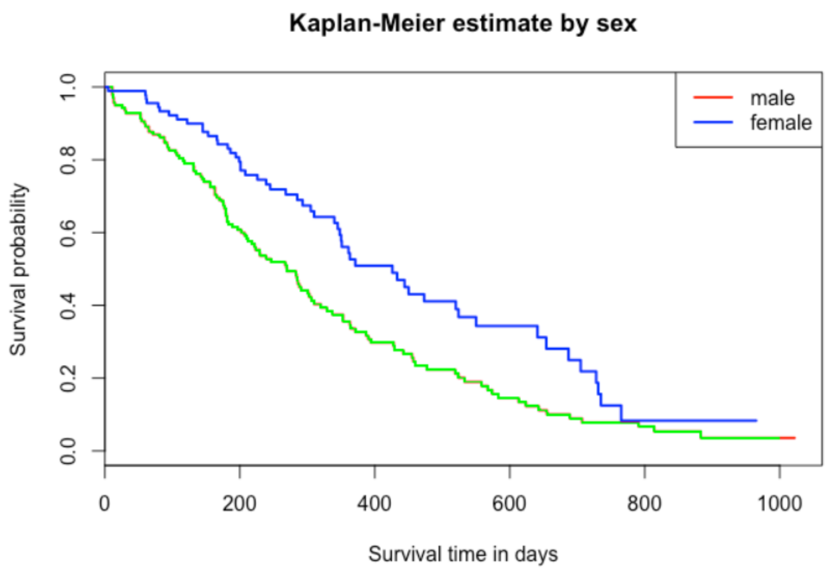
Transformation functions are then defined to map from pixel space to coordinate space and vice versa, using the obtained scale and origin.

Locating the Points on the Curve

Having all this, we need to identify the curves, and locate all the points of the curves in pixel space. This is done by "tracking" the curves in a sense. For every possible colour value that exist within the appropriate region of the graph, the curve is tracked to ensure continuity, starting from left to right. Conditions on number of data points and nature of the curve are applied to ensure that what we have is a curve. Jumps are also allowed for the overlap of curves. This means that even dashed curves can be tracked.

The results obtained are visualised as below:





All these coordinate points are stored in pixel space. Note that every single pixel with a matching colour is considered a point by this approach.

It is assumed that each curve is a different colour. A slight modification can allow for tracking of same colour curves.

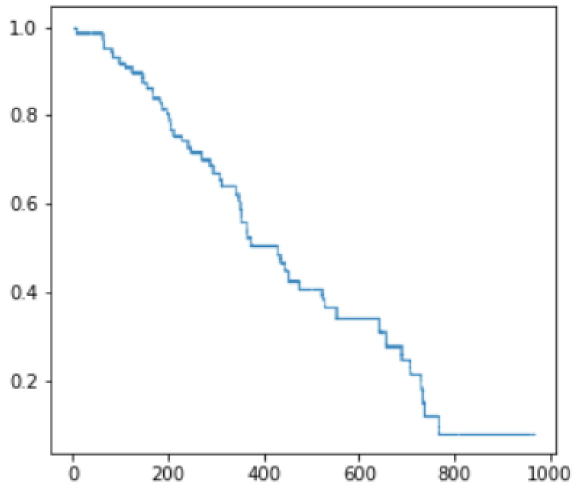
Converting Curves to Actual Coordinate Space

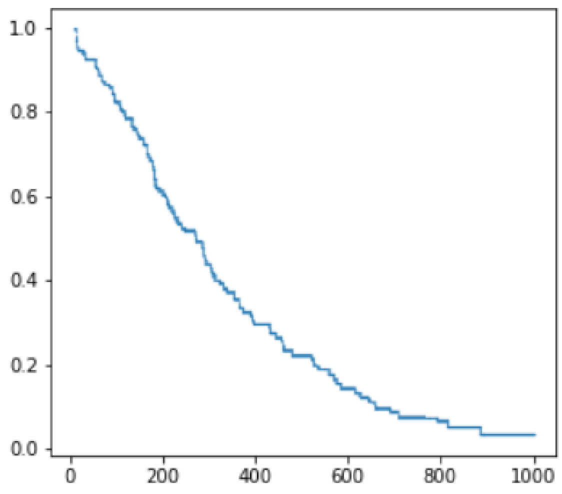
Now that the curves have been isolated in the pixel space, we need to convert it into actual coordinate space. This can be simply done by using the transformation functions defined earlier. With this, we have our data points in x-y space. This is then saved to a csv file.

A point to note here is that as each pixel is a data point, each x coordinate has multiple y coordinates across different data points. Additionally, the number of data points is rather large. In this code we have not modified this as there is no requirement of us to do so in the problem statement. However, if this is required, it can be done in the following manner:

- For single data point for every x, it is best to take the largest y coordinate for every x. This is simply to maintain consistency, as that is also how the axis coordinates were initially determined. This gives us a smooth curve.
- To reduce data points, we can simply consider every 5th data point. An alternative is to only save those data points which have a change in the y coordinate value, thus only reflecting change.

The extracted values are replotted using matplotlib to see if we can get the original curves back, just to verify that the entire process worked.





Advantages of this approach

Having implemented the algorithm, the advantages it is able to offer are as follows:

- Algorithm can generalize well to different types of graphs
- It is able to handle multiple curves, as well as multiple types of curves, such as dashed curves
- There is no need to train the algorithm
- There is no need to collect large amounts of training data
- The algorithm is reliable and controllable, so we can predict how it will react to different situations.
- It is highly efficient
- The basis of the algorithm is our intuitive understanding of graphs.

Further Possible Improvements

The solution has potential for being adopted as a general all-purpose algorithm for any graph. However, it is not there yet and can be improved upon:

- The character recognition technique can be something optimized for number recognition, rather than just using pytesseract. This can give better results.
- One issue is that some images colours are too widespread for them to be clustered together. Some kind of image cleaning/segmentation pre-processing can maybe be done to further improve the working efficiency of the algorithm.
- Code can be modified to work with graphs that have multiple curves of one colour.
- The colours of the background and the axes can be dynamically determined from the image, rather than assuming them the be white and black respectively.