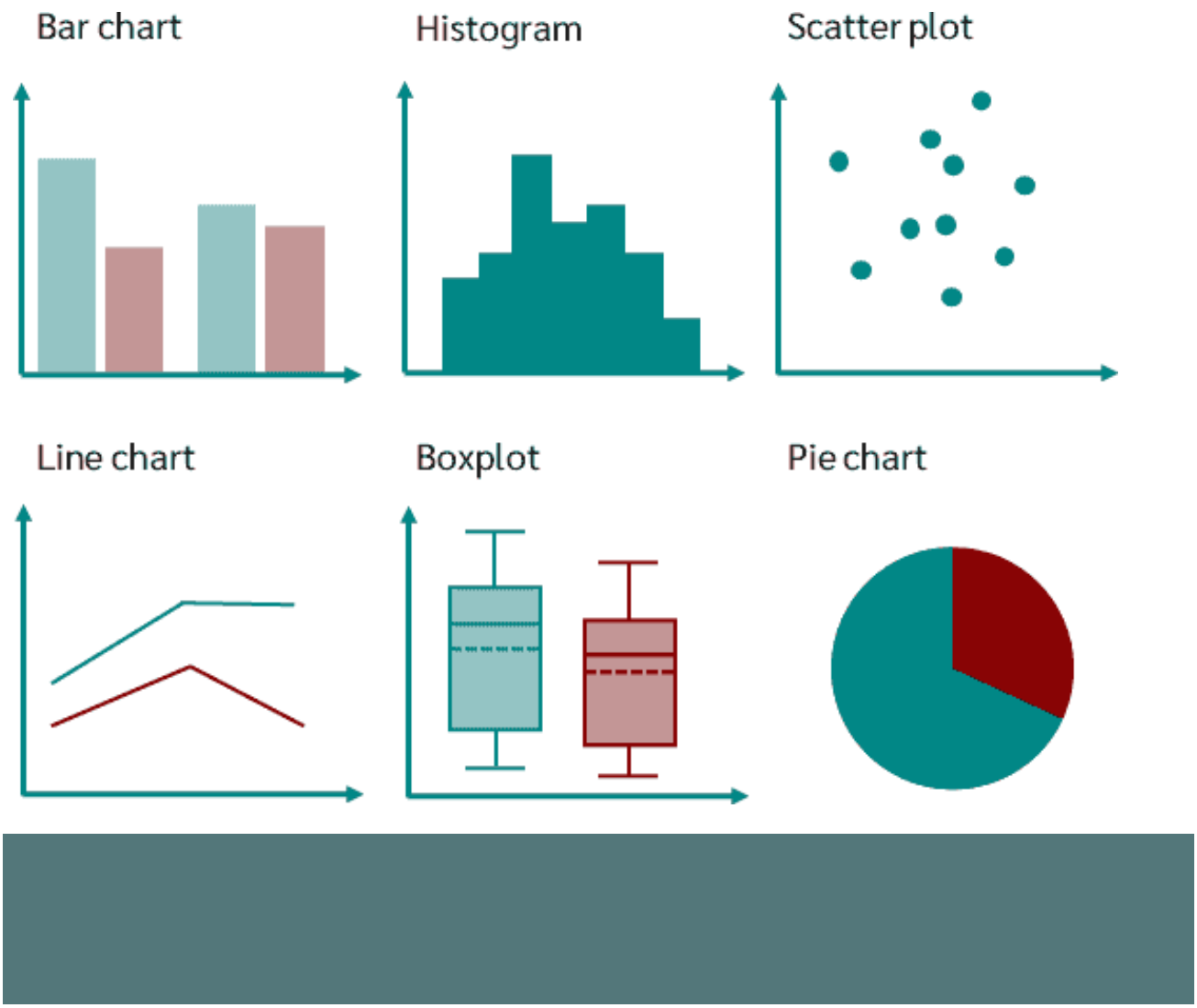
DIGITIZING GRAPH IMAGES

Matt Mascarelli

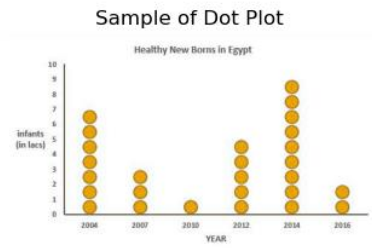


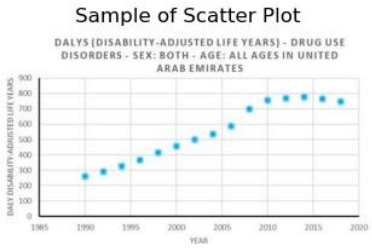
CONTEXT

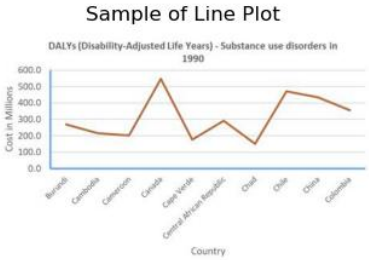
This project aims to address the educational barriers faced by millions of students with learning, physical, or visual disabilities preventing them from reading conventional print. Many STEM educational materials are inaccessible to these students, particularly when it comes to visuals like graphs. While technology can make the written word accessible, adapting visuals remains complex and resource intensive. Manual efforts to create accessible materials are costly and time-consuming, limiting their availability to schools without sufficient funding. To overcome these challenges, the project focuses on using machine learning to automate the process of digitizing various graph types, thereby making educational materials more accessible to learners with disabilities.

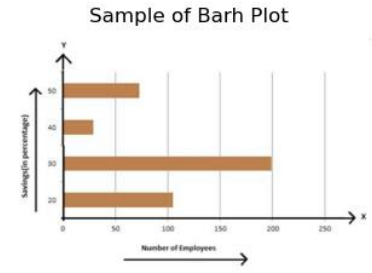
DATA WRANGLING

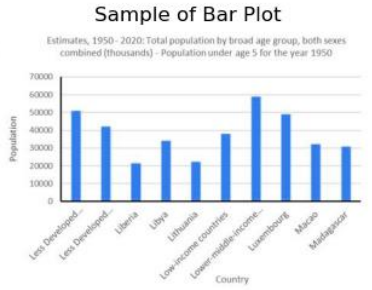
The data for this project originates from Kaggle and consists of 60,578 images and their corresponding annotations in json files. There are 24942 line plots, 19189 vertical bar plots, 11243 scatterplots, 5131 dot plots, and 73 horizontal bar plots. Due to the major class imbalance, specifically with the horizontal bar plots, 200 synthetic images were created by augmenting the original 73.











PROJECT PIPELINE

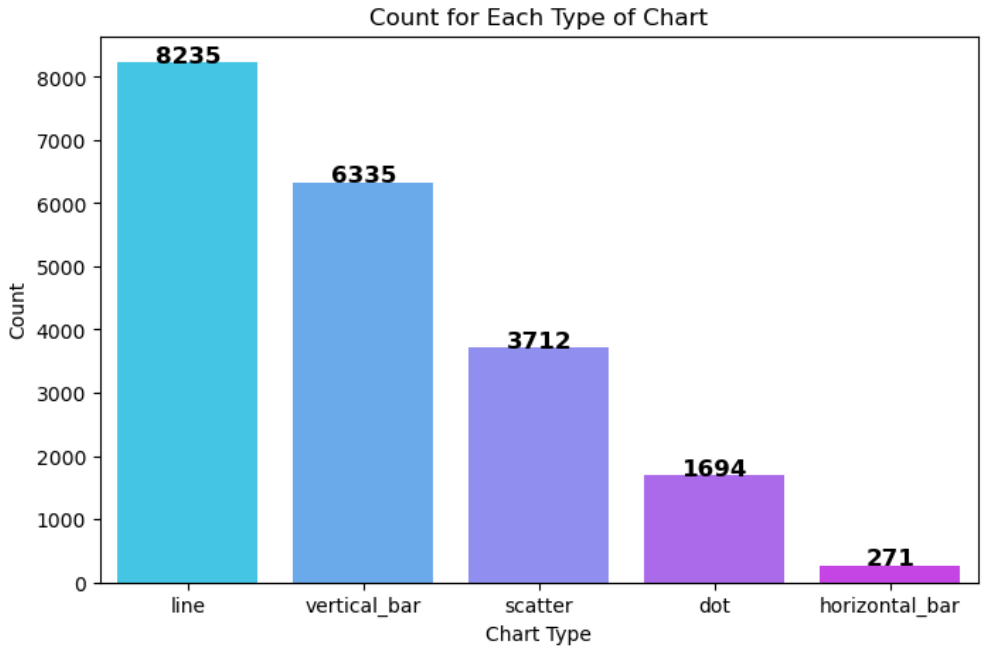
I. Train a convolutional neural network to classify the chart type.
   - Extracting data will be slightly different for each of the different chart types, so being able to correctly classify the type is a vital first step.

II. Train a custom object detector using the YOLOv8 architecture for each of the 5 chart types:
   - Scatter YOLO model
   - Vertical bar YOLO model
   - Horizontal bar YOLO model
   - Dot YOLO model
   - Line YOLO model

III. Detect XY plane within the image and rescale the data to be within the scale of the chart.
   - The YOLO models return bounding box coordinates of the detected objects on the scale of the image dimensions, but to successfully digitize the image it needs to be on the scale of the chart depicted in the image.

PART I: CLASSIFYING CHARTS

Image classification is a very common task in machine learning. Since our images are labeled, we can easily achieve this using deep learning. Due to the large number of images that we are working with, we will be training, validating, and testing this model with a subset of 20247 images, instead of all 60578 images.



TRAIN/VAL/TEST SPLIT

i. Training set: 13767 images

ii. Validation set: 3442 images

iii. Test set: 3038 images

iv. Note: Each split contains the same proportion of chart types as the original data

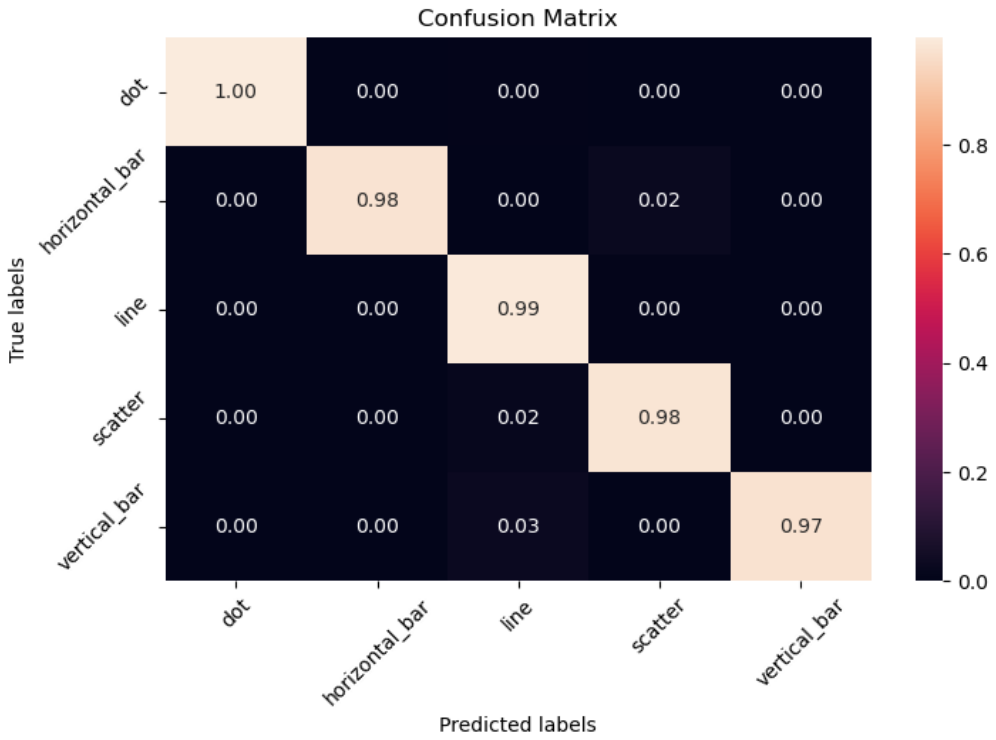
MODEL ARCHITECTURE:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 254, 254, 16)      448

 max_pooling2d (MaxPooling2D  (None, 127, 127, 16)     0
 )

 conv2d_1 (Conv2D)           (None, 125, 125, 32)      4640

 max_pooling2d_1 (MaxPooling  (None, 62, 62, 32)       0
 2D)

 conv2d_2 (Conv2D)           (None, 60, 60, 16)        4624

 max_pooling2d_2 (MaxPooling  (None, 30, 30, 16)       0
 2D)

 flatten (Flatten)           (None, 14400)             0

 dense (Dense)               (None, 128)               1843328

 dense_1 (Dense)             (None, 5)                 645

=================================================================
Total params: 1,853,685
Trainable params: 1,853,685
Non-trainable params: 0
_________________________________________________________________
```

RESULTS

Epoch	Train Loss	Train Accuracy	Val Loss	Val Accuracy
1	6.9239	0.8080	0.2123	0.9332
2	0.1676	0.9447	0.1022	0.9646
3	0.1943	0.9395	0.1986	0.9396
4	0.1888	0.9362	0.1323	0.9637
5	0.0474	0.9860	0.0823	0.9770

The model is achieving an overall accuracy of 98.6% on the training set and 97.7% on the validation set. It appears that the model could be slightly overfitting, but when the model is used on the test set, we get an accuracy of 98.4% which is on par with the training accuracy. The class-wise accuracies are displayed below:



The dot and line plots are performing the highest with 100% and 99% accuracy respectively, while horizontal bar, scatter, and vertical bar are getting 98%, 98%, and 97% respectively. It is quite possible that these results could be improved by utilizing the full data set for training, but these results are more than satisfactory, and this model will be used for the rest of the project pipeline.

PART II: YOLO OBJECT DETECTION

1) YOLO (You Only Look Once) is a popular set of object detection models used for real-time object detection and classification in computer vision. In this project we will be focusing on using YOLOv8, the latest version of the YOLO system.

2) Due to the computing power required to train a model using the YOLOv8 architecture, most of our models were trained on an AWS ec2 instance.

3) For training a model this way, the data will need to be setup in a very specific way.

    i) First, every image will need to have a .txt file created which will contain the class id, x-dimension center, y-dimension center, width, and height for the bounding box of each object in the image. These will also need to be standardized by dividing x-center and width by the image width, and y-center and height by the image height.
    ii) Second, a data.yaml file needs to be created which contains the absolute path to the images, the path to the training data, the path to the validation data, the number of classes, and names for each class.

EXAMPLE TXT FILE:

```
0 0.16071428571428573 0.43862275449101795 0.04365079365079365 0.3562874251497006
0 0.24603174603174602 0.5553892215568862 0.04365079365079365 0.12275449101796407
0 0.33134920634920634 0.49700598802395207 0.04365079365079365 0.23952095808383234
0 0.41865079365079366 0.44011976047904194 0.04365079365079365 0.3532934131736527
0 0.503968253968254 0.4625748502994012 0.04365079365079365 0.3083832335329341
0 0.5892857142857143 0.4655688622754491 0.04365079365079365 0.3023952095808383
0 0.6746031746031746 0.5658682634730539 0.04365079365079365 0.10179640718562874
0 0.7599206349206349 0.42664670658682635 0.04365079365079365 0.38023952095808383
0 0.8472222222222222 0.5449101796407185 0.04365079365079365 0.1437125748502994
0 0.9325396825396826 0.5299401197604791 0.04365079365079365 0.17365269461077845
```

EXAMPLE DATA.YAML FILE:

```yaml
path: /Users/matt/Desktop/graphs-capstone/data/processed/YOLO
train: Barplots/train
val: Barplots/test
nc: 1
names:
  0: bar
```

Note: The example above is for a bar plot. Because each of the chart types are being trained separately, there is only 1 class type in each image. In the txt file, the first column is all zeroes because each object is a bar, which is defined in the yaml file. This format will essentially be the same for all the chart types.

BAR PLOT MODELS

- There are 19189 vertical bar plots in the original data but will be using 500 with a 400/100 train/val split to build the model. YOLO works well on large objects, so due to the size of the bars in the images, using thousands of images will not be necessary. The remaining 18689 will be treated as a test set.

- There are only 73 horizontal bar plots in the original data, and we cannot use the synthetically generated images from part I, as they are not annotated. The train/val split is 58/15.

    - Note: Programs exists that allow you to manually draw the bounding boxes and get the coordinates as an output, but this is time consuming and will not be done for this project.

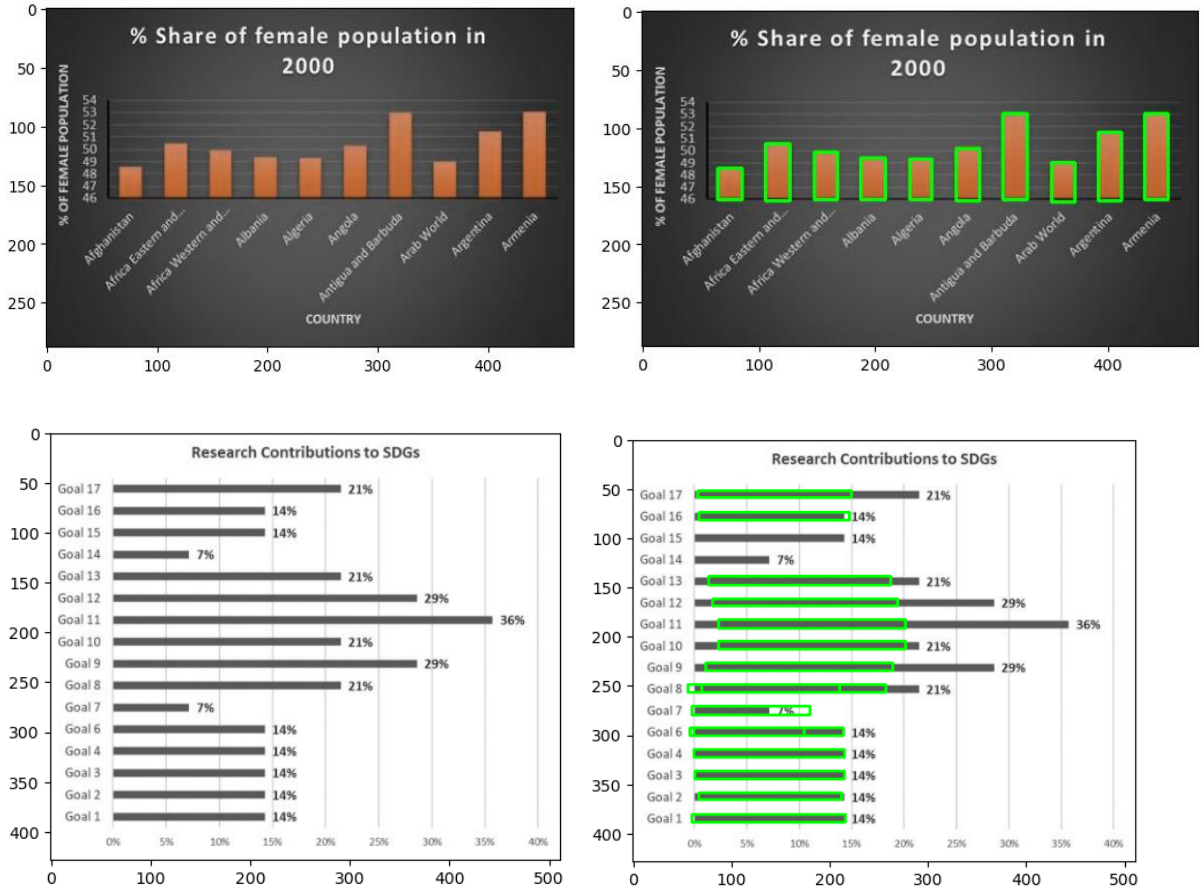
- The annotation data for each bar plot image contains the left x-coordinate, bottom y-coordinate, width, and height for each of the bounding boxes. To get the data in the correct YOLO format, some simple preprocessing needs to be done.

    - $x - center = \frac{x_0+(x_0+bbox\ width)}{2*image\_width}$
    - $width = \frac{bbox\ width}{image\_width}$
    - $y - center = \frac{y_0+(y_0+bbox\ height)}{2*image\ height}$
    - $height = \frac{bbox\ height}{image\ height}$

TRAINING RESULTS

Chart Type	**Precision**	**Recall**
Vertical Bar (Val)	0.994	0.904
Vertical Bar (Test)	0.963	0.936
Horizontal Bar	0.796	0.712

SAMPLE IMAGE PREDICTED BOUNDING BOXES



As we can observe from the examples above, the vertical bar model is performing as it should, detecting all of the bars in the image. The horizontal bar model however, is not performing well at all. It is completely missing some of the bars, and is not detecting the correct width for many of the bars. We can surmise that this poor performance is due to the small size of the training data. If we collected more images, and manually annotated the bounding boxes, we could retrain the model for a much more accurate performance.

SCATTER PLOT MODEL

- There are 11243 scatter plots in the original data but will be using 4000 with a 3200/800 train/val split to build the model. YOLO does no work well on small objects, so due to the small size of the scatter points in the images, using more images in training will be necessary. The remaining 7243 will be treated as a test set.

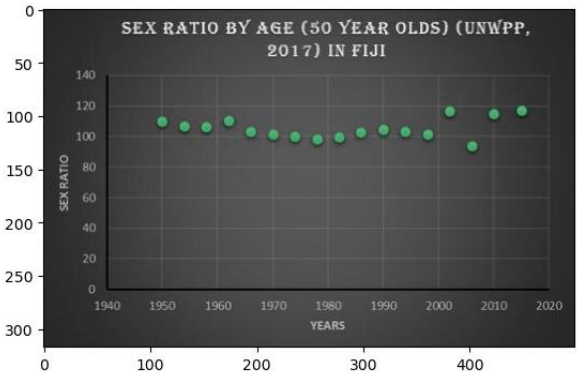
- The annotation data for each scatter plot image contains the x-center and y-center for the points. This reduces some of the preprocessing, as we need those for the YOLO format. However, we are not given any information for the width and height of the bounding boxes. This is tricky, as each scatterplot will have different sized scatter points. One way to deal with this would be to manually go through all the images and label the points as small, medium, or large and have different heights and widths for each group. This would be very time consuming, so I decided on using a standard width and height of 12 units for all of them. With that done, all that needs to be calculated are the standardized units required by the YOLO algorithm.

  - $x - center = \frac{x-center}{image\_width}$
  - $width = \frac{bbox\ width}{image\_width}$
  - $y - center = \frac{y-center}{image\ height}$
  - $height = \frac{bbox\ height}{image\ height}$

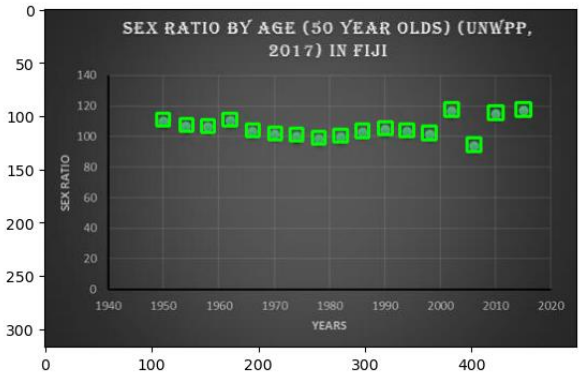
TRAINING RESULTS

Chart Type	Precision	Recall
Scatter (Val)	0.970	0.913
Scatter (Test)	0.965	0.918

SAMPLE IMAGE



PREDICTED BOUNDING BOXES



LINE PLOT MODEL

- There are 24942 line plots in the original data but will be using 5000 with a 4000/1000 train/val split to build the model. The remaining 19942 will be treated as a test set.

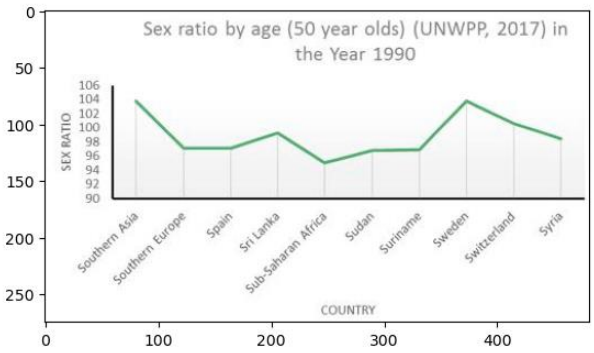
- The annotation data for each line plot image contains the x-center and y-center for the points at each tick mark. This might be tricky for the model to detect, but we will follow a similar method for setting up the bounding boxes as we did with the scatter plots. We will use a width and height of 12 units.

  - $x - center = \frac{x-center}{image\_width}$
  - $width = \frac{bbox\ width}{image\_width}$
  - $y - center = \frac{y-center}{image\ height}$
  - $height = \frac{bbox\ height}{image\ height}$

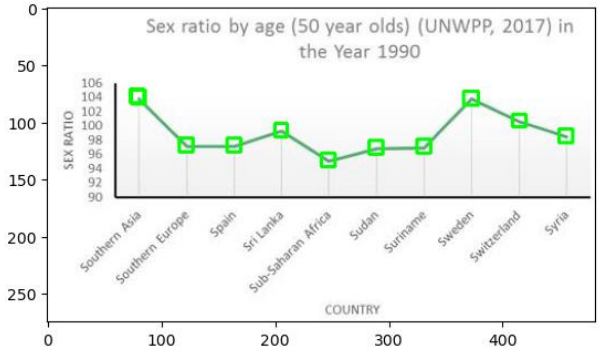
TRAINING RESULTS

Chart Type	**Precision**	**Recall**
Line (Val)	0.744	0.741
Line (Test)	0.728	0.743

SAMPLE IMAGE



PREDICTED BOUNDING BOXES



DOT PLOT MODEL

- There are 5131 dot plots in the original data but will be using 3000 with a 2400/600 train/val split to build the model. The remaining 2131 will be treated as a test set.

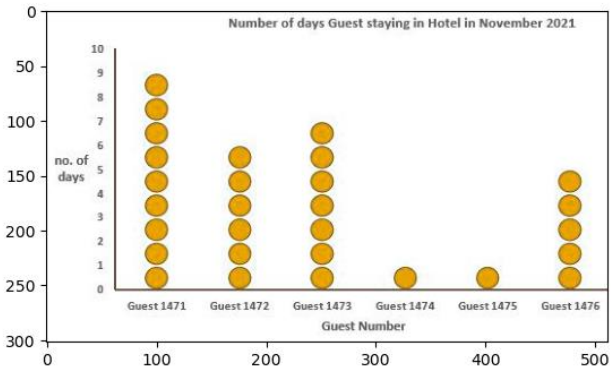
- The annotation data for each line plot image contains the x-center and y-center for each of the dots. This is similar to the way the data was set up for scatter plots. However, the dots are generally much larger than the scatter points. I used a standard width and height of 20 units for the bounding boxes.

  - $x - center = \frac{x-center}{image\_width}$
  - $width = \frac{bbox\ width}{image\_width}$
  - $y - center = \frac{y-center}{image\ height}$
  - $height = \frac{bbox\ height}{image\ height}$

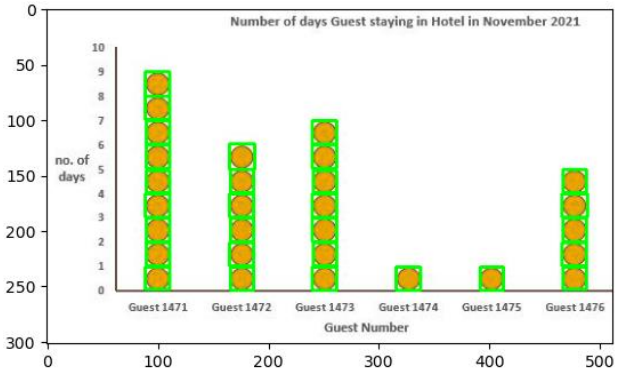
TRAINING RESULTS

Chart Type	Precision	Recall
Dot (Val)	0.995	0.981
Dot (Test)	0.996	0.975

SAMPLE IMAGE



PREDICTED BOUNDING BOXES

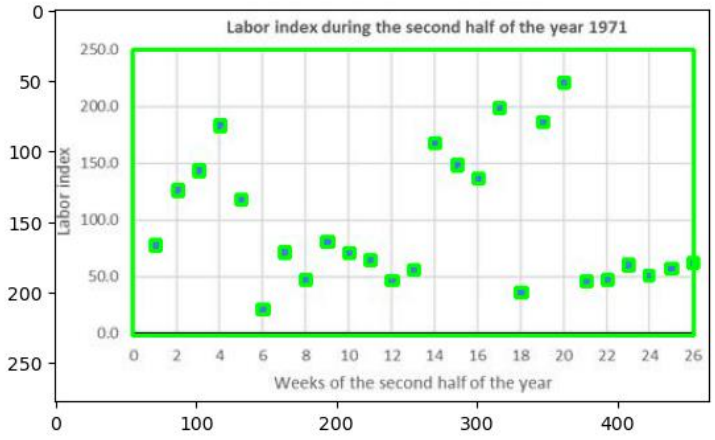


PART III: DETECTING XY PLANE AND RESCALING

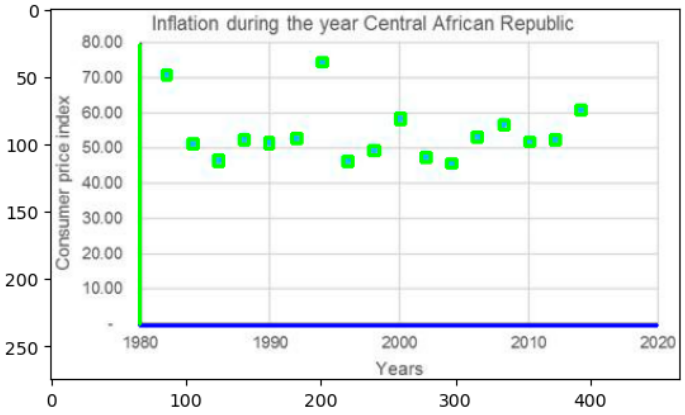
Now that we have algorithms for extracting data from the images, we need a way to get the data on the same scale of the charts depicted in the images. The first thing we need to do is detect the XY plane that our graph resides in. This can be achieved by using the OpenCV python library for computer vision.

STEP 1: DETECTING XY PLANE

Detecting the largest rectangle should result in being the total grid area of the XY plane. This can be done by using the 'findContours' function of cv2 and filtering out the smaller contours. In the image below you can see this at work. The XY plane has been detected and will allows us to have some boundary points when rescaling our points.



However, this method does not work on all the images. There is a back-up method in place to try and detect the XY axes instead using the 'HoughLinesP' function of cv2. This is done by detecting all the vertical lines (y-axis) and horizontal lines (x-axis) and then filtering out the lines that are too small/large and ones that are not within a certain boundary of the images. The image below shows an example of this working.



STEP 2: RESCALING

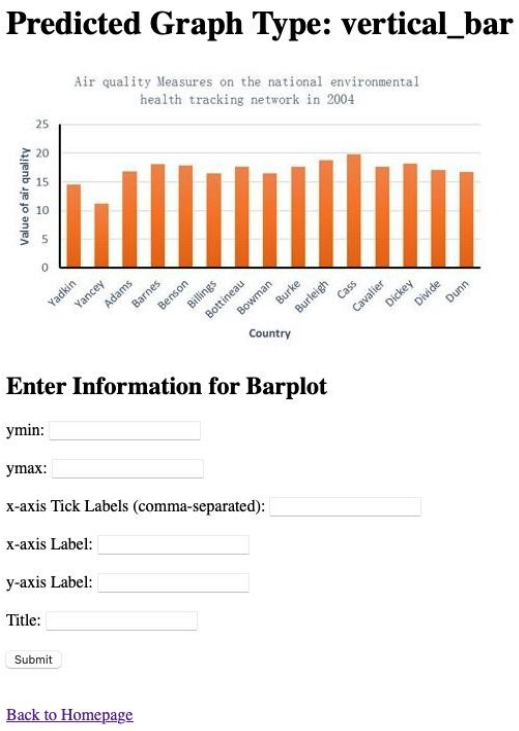
Rescaling the data is the final part of the process needed to digitize the chart depicted in the images. For example, the first image above has the x-axis from 0 to 26 and the y-axis from 0 to 250. Ideally this process would be automated and can be done with OCR (optical character recognition) algorithms. Unfortunately, the results from working with several different OCR libraries are less than desirable. I have no doubt that with more work this can be achieved, but for the time being the scales need to be inputted manually. Likewise for tick labels that are words, they need to be manually typed in.

PART IV: WEB APP

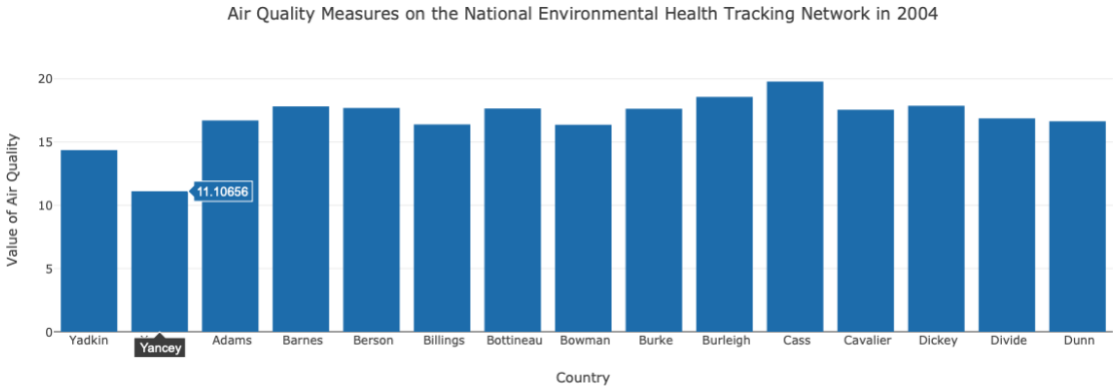
Now that the full project pipeline has been complete, we can see this work in a simple web application that I build using Flask.

A. First, you can upload the image of a graph:

B. Next, the program predicts the chart type and displays some forms that need to be filled out in order for the graph to be displayed in a digital form:



C. A plotly graph is then displayed for the user to explore. Plotly is a great data visualization library that produces interactive charts. It allows you to hover over data points with your mouse to get the actual number depicted by the bar. For example, the bar at the tick label 'Yancey' looks like it a little above 10 on the y-axis, but when you hover over the bar you can see the exact value is 11.10656.



PART V: CONCLUSIONS AND FURTHER WORK

The aim of this project was to digitize images of different chart types in the hopes of making charts more accessible for people who have barriers that make it difficult for them to read the information portrayed in a mathematical chart. The results of this project, specifically the web application that I built, can be used by schools, students, textbook companies, math apps, and others. There is still much more work that can be done to improve these results, but it is a great first step towards digitizing charts and making them more accessible.

NEXT STEPS:

1. Improving accuracy by including more training data.
2. Collect more horizontal bar plots and manually annotate the bar bounding boxes.
3. Automating the rescaling process by using OCR algorithms to extract xy tick labels.
4. Currently, the web app only works for scatter, vertical bar, and line plots. I would like to update it to work for horizontal bar and dot plots.

REFERENCES

[1] Data Source: https://www.kaggle.com/competitions/benetech-making-graphs-accessible/overview

[2] YOLO: https://ultralytics.com/yolov8

[3] OpenCV: https://opencv.org

[4] Flask: https://flask.palletsprojects.com/en/2.3.x/

[5] Plotly: https://plotly.com/