

Algorithmic Intelligence The Joy of Coding Workbook #3 Pre-trained Models

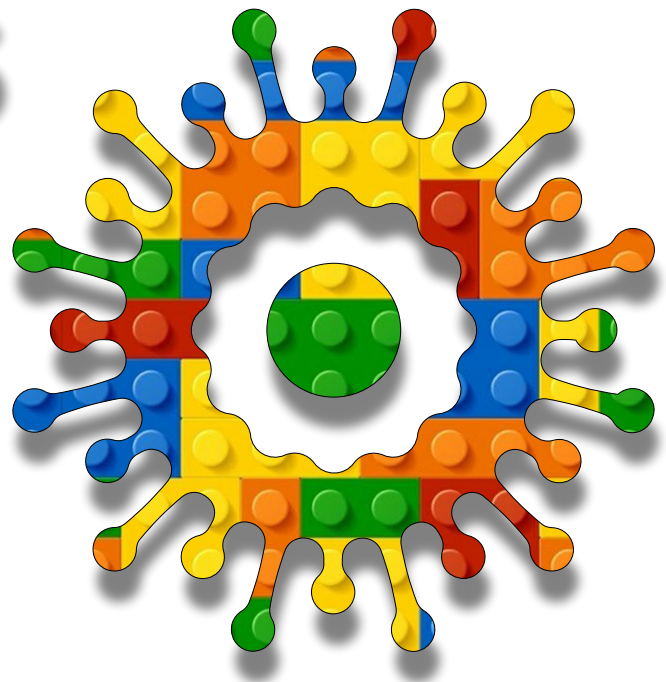




Table of Contents

MIT Licence	6
Workbook #3 Quick Content Summary	7
Module B Unit #1: p5.js code snippets #3	9
Sketch B1.1 drawing a rectangle	10
Sketch B1.2 drawing an ellipse	12
Sketch B1.3 drawing triangles	14
Sketch B1.4 measuring the distance	16
Sketch B1.5 in the name of the colour	18
Module B Unit #2: faceMesh with ml5.js	21
FaceMesh points	22
The index.html file	24
Sketch B2.1 our starting sketch	25
Sketch B2.2 creating a video from your webcam	26
Sketch B2.3 video on the canvas	27
Sketch B2.4 video flipped	29
Sketch B2.5 faceMesh	31
Sketch B2.6 callback	32
Sketch B2.7 the faces array	35
Sketch B2.8 collecting the lips	37
Sketch B2.9 face mirrored	39
Sketch B2.10 adding eyes	41
Drawing the data points	44
Sketch B2.11 a slimmed down version	45
Sketch B2.12 a new array of objects	46
Sketch B2.13 keypoint array	48
Sketch B2.14 finally remove the background image	50
Module B Unit #3: handPose with ml5.js	53
The index.html file	54
Sketch B3.1 our starting sketch	55
Sketch B3.2 video	56
Sketch B3.3 putting you on the canvas	57
Sketch B3.4 flipping the video	58
Sketch B3.5 the handPose model	59
Sketch B3.6 start detecting	60
Sketch B3.7 callback	61
Sketch B3.8 an array of hands	63
Sketch B3.9 the keypoints	64
Sketch B3.10 mirroring the yellow dots	66
Sketch B3.11 reference keypoints	68

Sketch B3.12 index and thumb	71
Sketch B3.13 circle size	73
Sketch B3.14 centring the circle	75
Sketch B3.15 using two hands	77
Module B Unit #4: bodyPose with ml5.js	80
The index.html file	81
Sketch B4.1 let us begin	82
Sketch B4.2 preload	83
Sketch B4.3 get the results	84
Sketch B4.4 keypoints	86
Sketch B4.5 confidence	88
Sketch B4.6 skeleton	90
Module B Unit #5: soundClassifier with ml5.js	94
The index.html file	95
Sketch B5.1 the preload starting sketch	96
Sketch B5.2 classifying	97
Sketch B5.3 the got the results	98
Sketch B5.5 a moving ball	100
Sketch B5.6 a splash of colour and text	103
Module B Unit #6: body segmentation with ml5.js	108
Sketch B6.1 our basic sketch	110
Sketch B6.2 preload	111
Sketch B6.3 masking the body parts	112
Sketch B6.4 start detecting your body parts	113
Sketch B6.5 gotResults()	114
Sketch B6.6 detecting your segments	115
Sketch B6.7 selecting a body part	117
Module B Unit #7: image classification	120
Sketch B7.1 index.html	121
Sketch B7.2 callback	122
A banana	123
Sketch B7.3 the appearance of the banana	126
Sketch B7.4 the classification	128
Sketch B7.5 get the results	130
Sketch B7.6 display	131
Module B Unit #8: video classification	133
Sketch B8.1 index.html	134
Sketch B8.2 starting sketch	135
Sketch B8.3 classifying	137
Sketch B8.4 display the results	139
Sketch B8.5 a bit of a mug	141

Module B Unit #9: bounding box image classification	144
Sketch B9.1 index.html	145
Sketch B9.2 the cocossd model	146
Sketch B9.3 let's go detecting	147
Sketch B9.4 resizing	149
Sketch B9.5 rearrange the image	151
Sketch B9.6 bounding box	153
Sketch B9.7 labelling the objects	155
Module B Unit #10: bounding box video classification	158
Sketch B10.1 index.html	159
Sketch B10.2 getting the video stream	160
Sketch B10.3 now to detect	162
Sketch B10.4 drawing the bounding box	163
Sketch B10.5 labelling	165
Module C Unit #1: introduction to transformers	168
Small Language Models	170
Hugging Face	171
Models	172
Transformer.js models	173
Pipeline API	174
The p5.js version	176
The import problem	179
Sentiment analysis	180
Specifying a model	181
Sketch C1.1 starting sketch	183
Sketch C1.2 text box and button	185
Sketch C1.3 sentiment	186
Sketch C1.4 text input	187
Module C Unit #2: image classification	190
Choosing the task	191
The task tab	192
Select one	193
Their example	194
Upload the dog	195
Sketch C2.1 the code	197
Module C Unit #3: depth analysis	200
Sketch C3.1 the code	201
Module C Unit #4: image detection	204
Sketch C4.1 image detection	205
Sketch C4.2 adding the label	207
Sketch C4.3 multiple objects	209



Copyright ©2026 Warren George

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Brief summary:

Permissions (what you can do)

Commercial use
Modification
Distribution
Private use

Limitations (what is not covered)

Liability
Warranty



Workbook #3 Quick Content Summary

For this unit we are using the latest version of p5.js (2.2.2 as of writing), also the latest version of ml5.js.

This unit covers pre-trained models, a pre-trained model is one that has (as the name suggests) already been trained on large amounts of data. For instance it may have been trained on 10s of 1000s of images which have been labelled. This is probably outside of your ability, so these models do offer something useful.

However, it is worth pointing out that there are large gaps in the datasets and the model are not perfect, hence why companies are using ever bigger datasets, employing large numbers of people to draw boxes round objects in the images and labelling. It is an expensive and labour intensive job getting the right data into a model.

As you may well be aware of, there is a sudden rise in the use of AI such as chatGPT or Gemini to name two, these are Large Language Models (LLMs) and they use staggering amounts of data scraped from the internet and are then trained on models with parameters that are in the billions. The machines that train these models (and run them) are so large they occupy warehouses the size of several football pitches.

As well as the big LLMs that cost billions of dollars to train and run there is an alternative Small Language Models (SLMs). These are, obviously, smaller, they are totally free, open source and run in your browser. We can use them to do numerous tasks. They are based on the same transformer architecture as the their big brothers. In this unit we will make use of some of these models and how they can be incorporated into p5.js.

Algorithmic Intelligence Module B Unit #1 p5.js code snippets #3



Module B Unit #1: p5.js code snippets #3

We are going to cover some essential coding snippets relevant to the next section on pre-trained models. If you have already completed the tutorials in Algorithmic Art then you could give this section a miss although it is often a good refresher.



Sketch B1.1 drawing a rectangle

We have covered most of the shapes you will use or need except for this one, the **rectangle**. A rectangle has four arguments. The first two are similar to those of the square, and they are the co-ordinates of the rectangle; the third is the width of the rectangle, and the fourth is the height of the rectangle.

```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  rect(100, 100, 200, 50)
  rect(100, 200, 20, 100)
  rect(200, 200, 100, 150)
}
```

Notes

Drawing three rectangles, the arguments are:

- A) x position
- B) y position
- C) horizontal dimension
- D) vertical dimension

Challenge

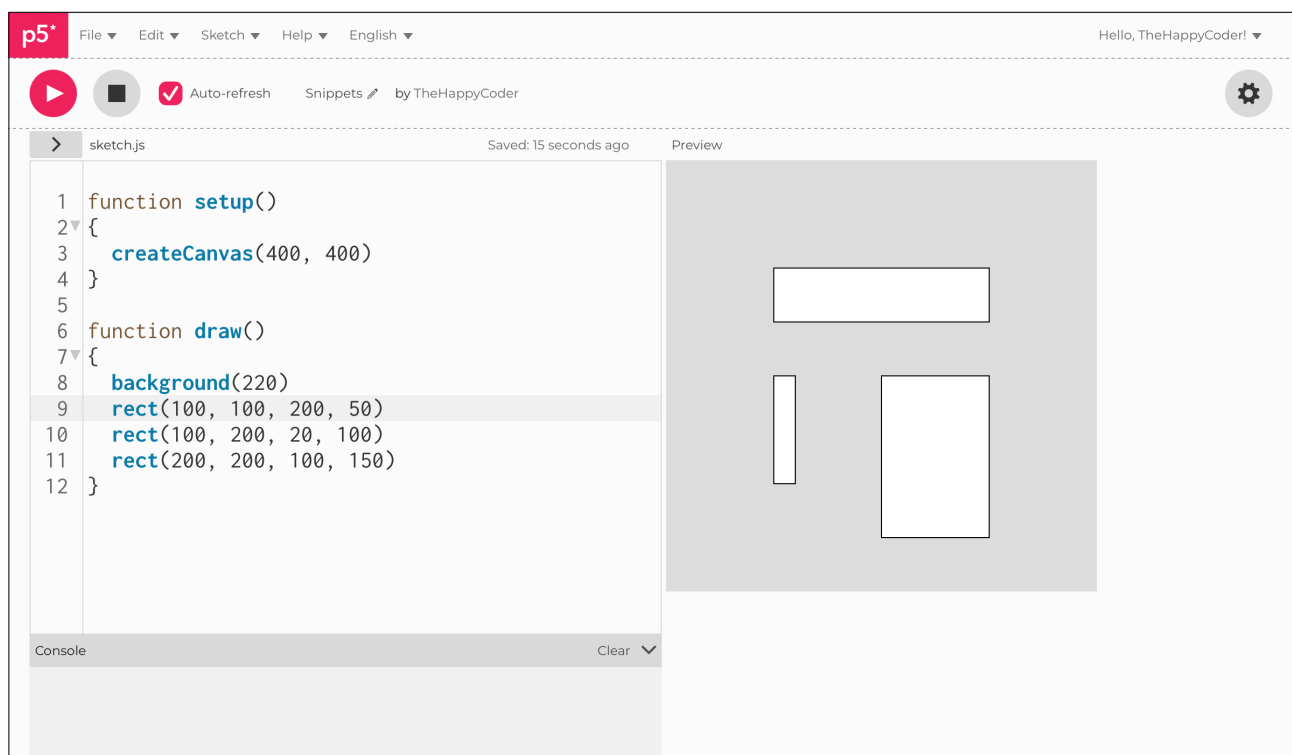
Add the `rectMode()` to put the co-ordinates in the centre of the rectangle.

Code Explanation

```
rect(100, 100, 200, 50)
```

Draws a rectangle at top left hand corner 100 from the left, 100 down and with a width of 200 and height of 50

Figure B1.1 drawing rectangles





Sketch B1.2 drawing an ellipse

Another shape that we can draw is an **ellipse**, which is similar to a circle with an extra argument.

```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  ellipse(200, 100, 150, 50)
  ellipse(100, 200, 20, 100)
  ellipse(200, 250, 100, 200)
  ellipse(300, 200, 20, 100)
}
```

Notes

We draw four ellipses, the arguments are:

- A) x position
- B) y position
- C) horizontal dimension
- D) vertical dimension

Challenge

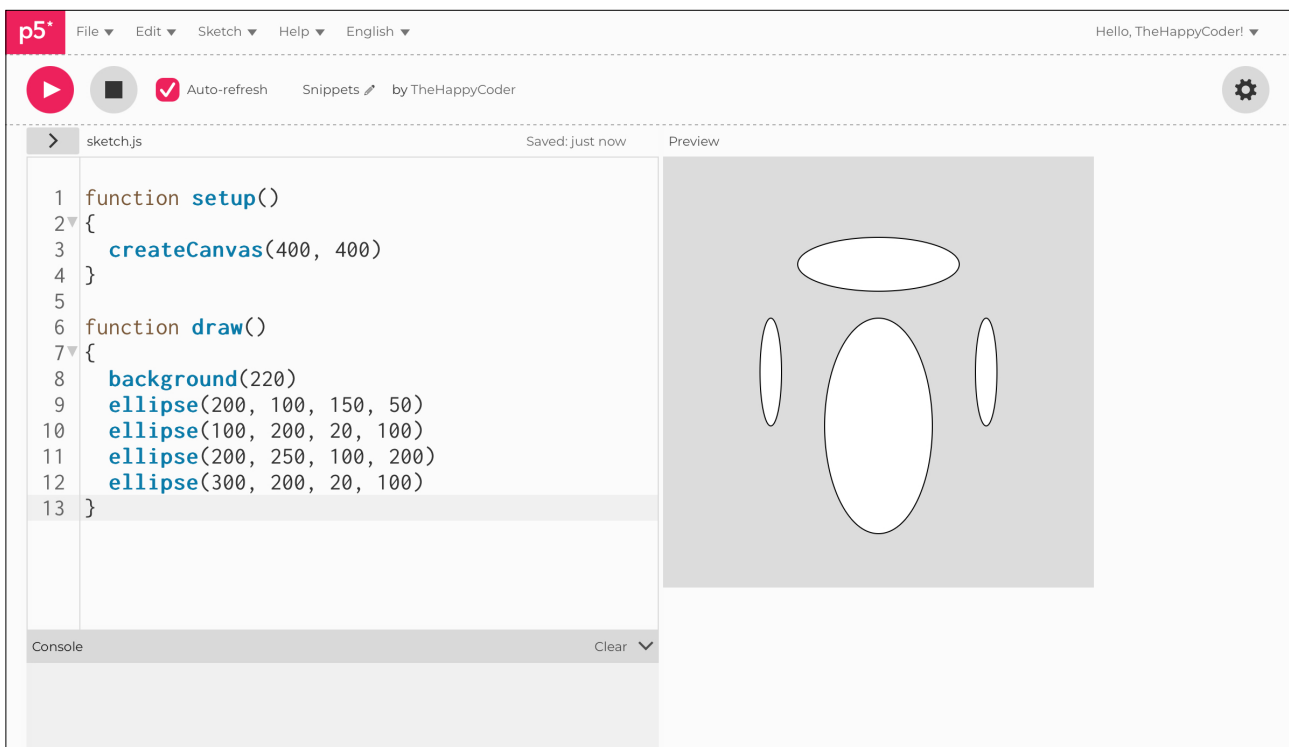
Make a pattern with them

Code Explanation

```
ellipse(200, 100, 150, 50)
```

First two arguments are the x and y co-ordinates, third is the width of ellipse and the fourth is the height of the ellipse

Figure B1.2





Sketch B1.3 drawing triangles

The triangle has six arguments for the three co-ordinates at each corner.

```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background(220)
  triangle(200, 100, 100, 190, 300, 190)
  triangle(200, 300, 100, 210, 300, 210)
}
```

Notes

The triangle requires a bit more thought organising the co-ordinates. I often sketch them out first on a piece of paper to help plan them out. The arguments are:

- A) x1 position
- B) y1 position
- C) x2 position
- D) y2 position
- E) x3 position
- F) y3 position

Challenge

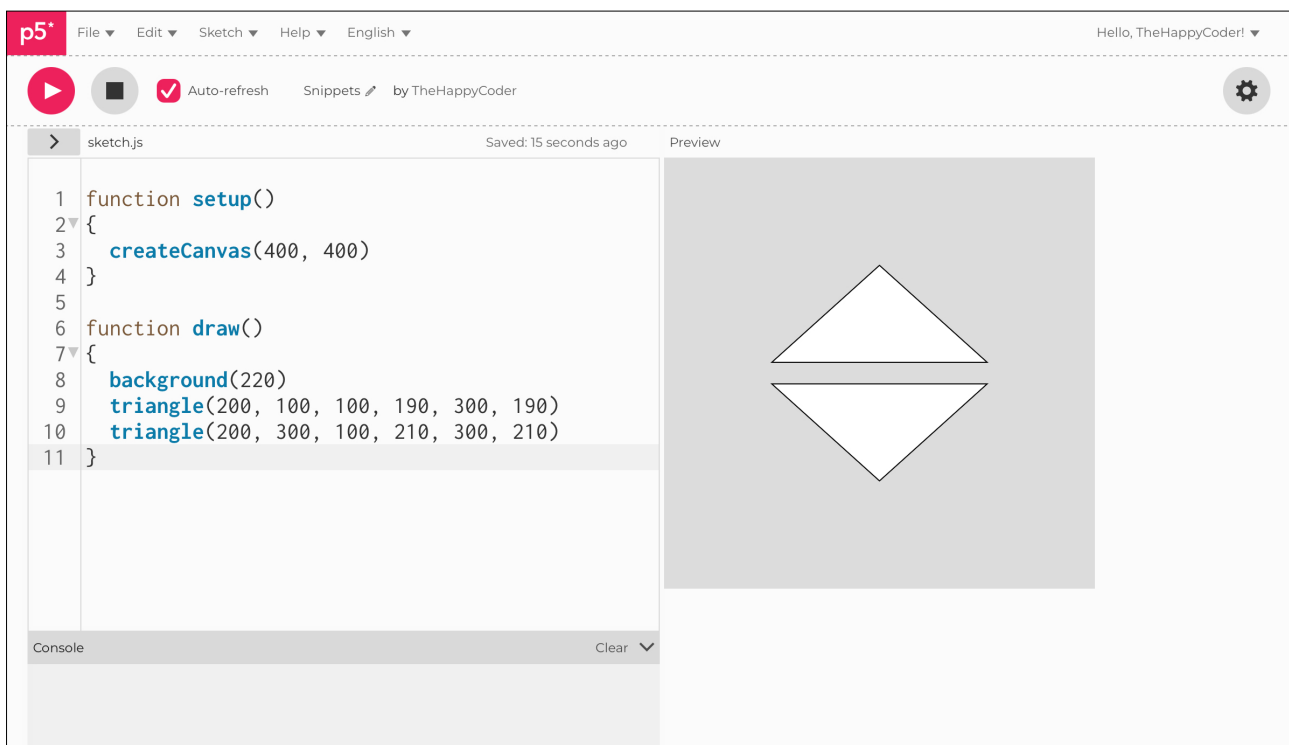
Make some more

Code Explanation

```
triangle(200, 100, 100, 190, 300, 190)
```

The three co-ordinates for the three corners of the triangle

Figure B1.3





Sketch B1.4 measuring the distance

! Starting a new sketch

If we want to know the distance between two points, for instance, the centre of two circles, we can use a `dist()` function that calculates the distance between those two points. This is useful when looking at collisions or in our ml5.js example later using it to draw a circle. Here we are measuring the distance between two circles and writing the value on the canvas.

```
let x1
let y1
let x2
let y2

function setup()
{
  createCanvas(400, 400)
  textSize(32)
}

function draw()
{
  background(200)
  x1 = width/2
  y1 = height/2
  x2 = mouseX
  y2 = mouseY
  let d = dist(x1, y1, x2, y2)
  text(d, 50, 50)
  circle(x1, y1, 25)
  circle(x2, y2, 25)
}
```


Notes

Calculates the distance between two points. The version of `dist()` with four parameters calculates distance in two dimensions. The version of `dist()` with six parameters calculates distance in three dimensions; this is handy if you are using 3D shapes:

x1 position of point 1
y1 position of point 1
x2 position of point 2
y2 position of point 2

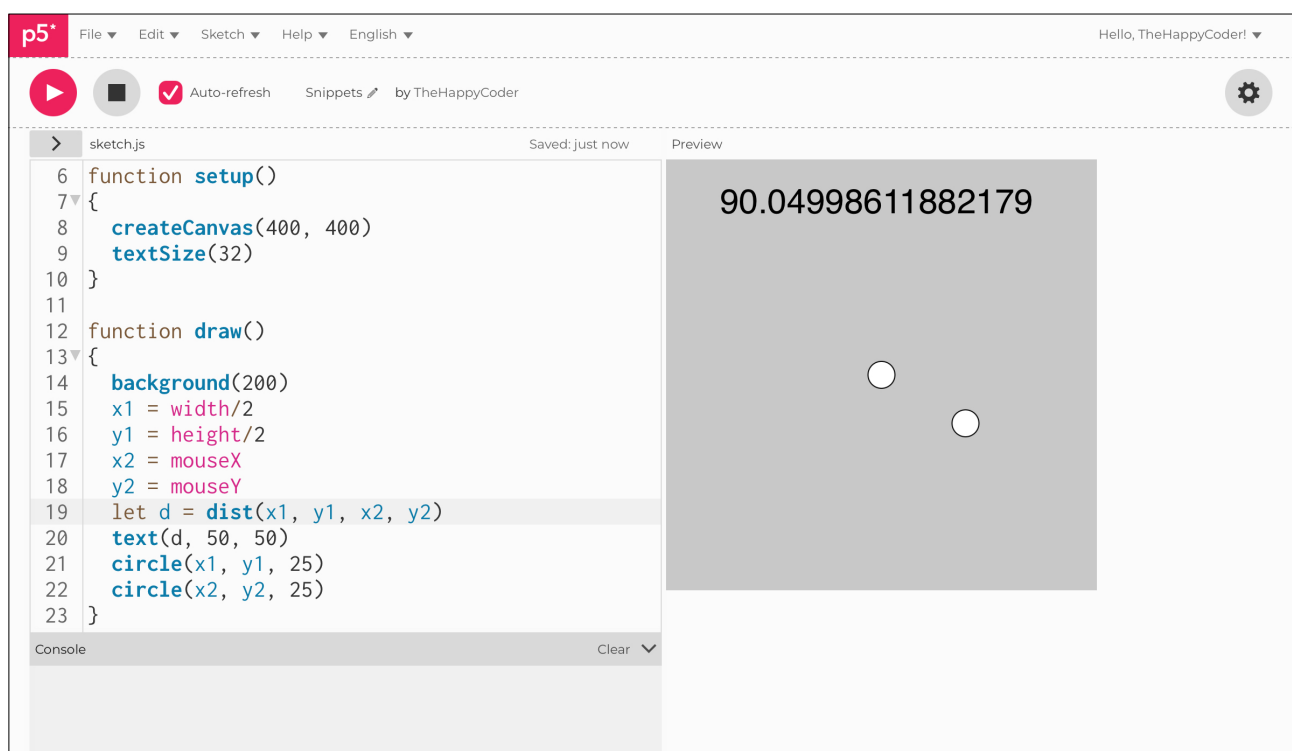
To make it just an integer replace with: `text(int(d), 50, 50)`

Code Explanation

```
let d = dist(x1, y1, x2, y2)
```

Calculating the distance (d) between two pairs of co-ordinates (x1, y1) and (x2, y2)

Figure B1.4





Sketch B1.5 in the name of the colour

! Starting another new sketch.

We can, if we wish, just use the name of the colours. There is a wide range of colours, too numerous to mention them here. The name must be in speech marks.

```
function setup()
{
  createCanvas(400, 400)
}

function draw()
{
  background('orange')
  fill('green')
  circle(100, 100, 100)
}
```

Notes

When you put the name of the colour in speech marks, it gives you a square indicator to the colour. There are quite a few colour names. If you use 'lightgreen' with no gap, you get light green. Some colours will take the word dark, for instance, **darkred**. There are other names such as **teal**, **magenta**, and so on. You can use single or double quotes. It is useful if you just want a simple colour rather than trying to remember the **RGB** values.

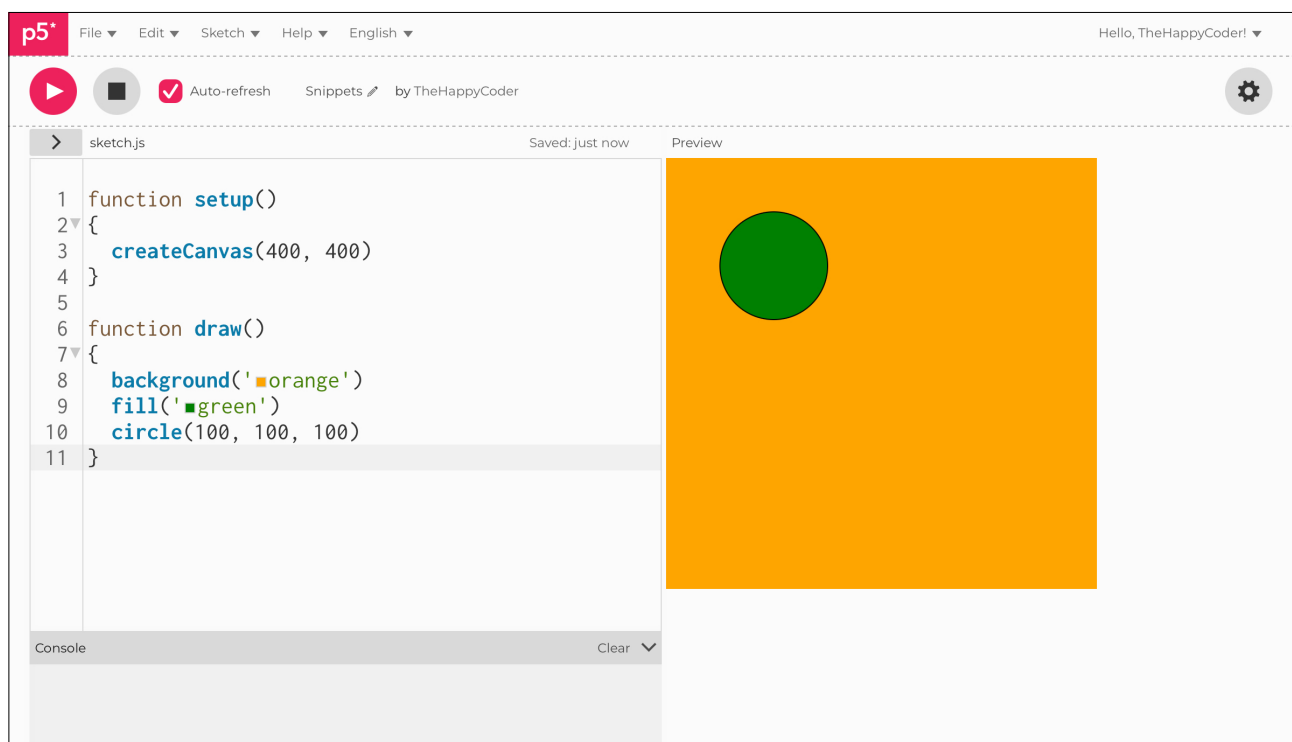
Challenge

Just experiment with the names of other colours (I have a list of all of them in my Algorithmic Art workbooks)

Code Explanation

<code>background('orange')</code>	Gives you an orange background
<code>fill('green')</code>	Fills the circle green

Figure B1.6



Algorithmic Intelligence Module B Unit #2 pre-trained faceMesh model

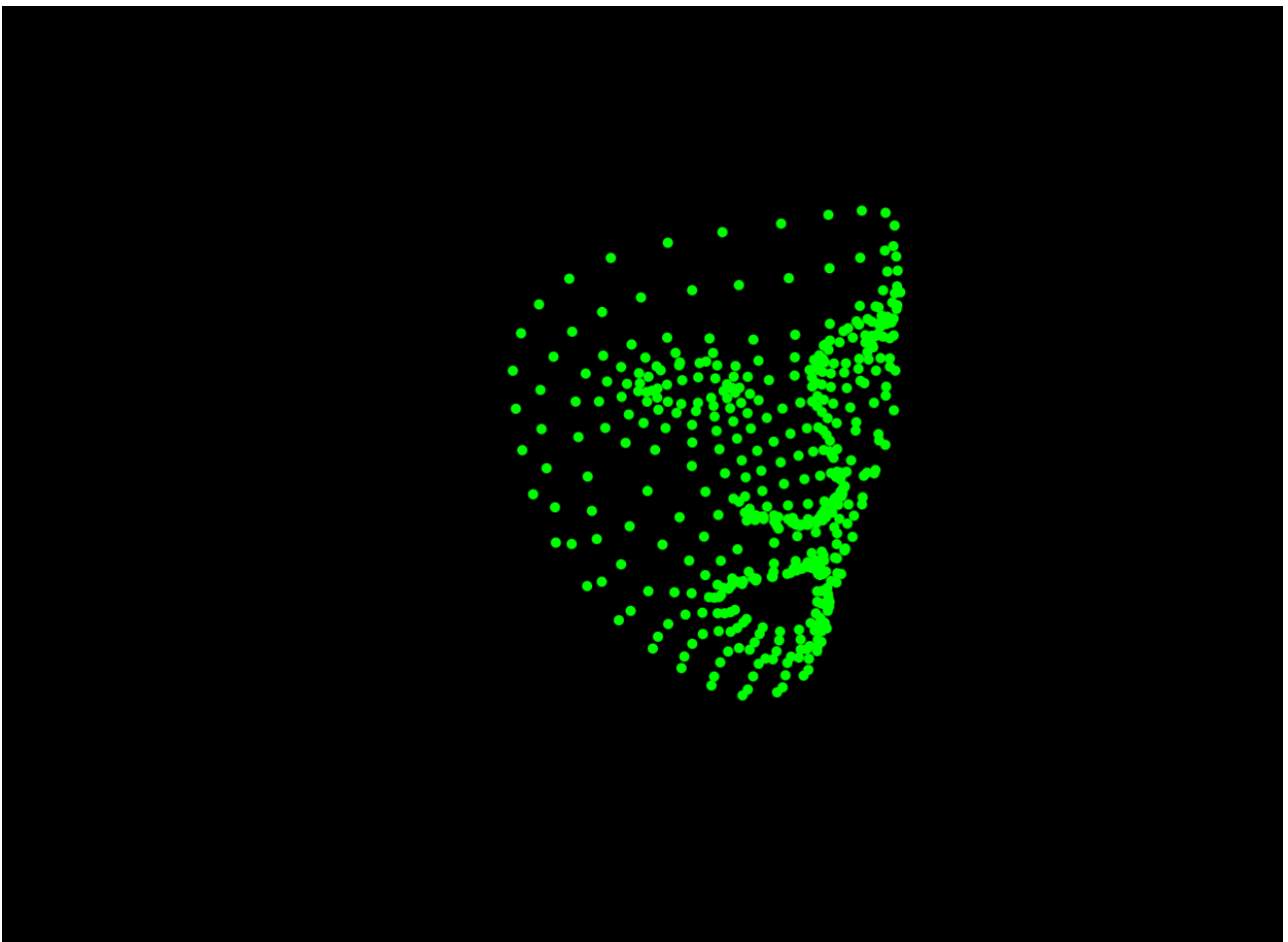


Module B Unit #2: faceMesh with ml5.js

You will need a **webcam** for this unit. Much of what we are going to do is covered in previous units and so will not spend time elaborating on how to create the video input.

Embedded within the ml5.js library there is a model called **faceMesh** which has been trained to identify over **400** points on any face. These points can identify the mouth, the eyes, eyebrows, etc. We can access these points if we want to focus on those particular features. The data points of the faceMesh have **3D** values but we will just use the 2D co-ordinates. In this example, we are going to draw a box around the mouth using **faceMesh**, add circles for eyes and draw all the points.

Figure 1: all of the data points of my face





FaceMesh points

The points on the faceMesh are numbered. This means you can reference specific points on the mesh. You can see the reference points and their associated numbers. I will include a button link to this on the website.

Figure 2: faceMesh reference points

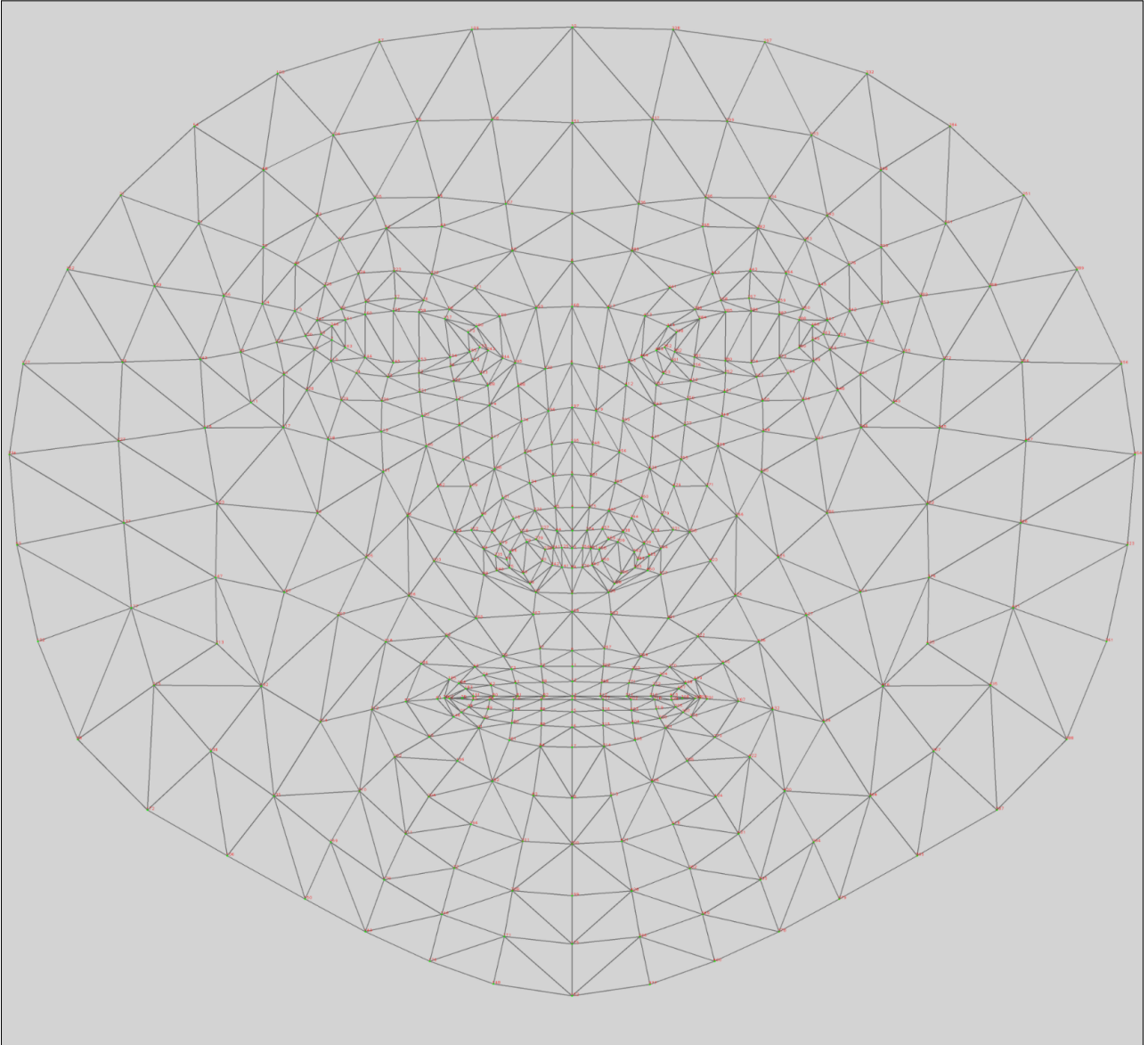
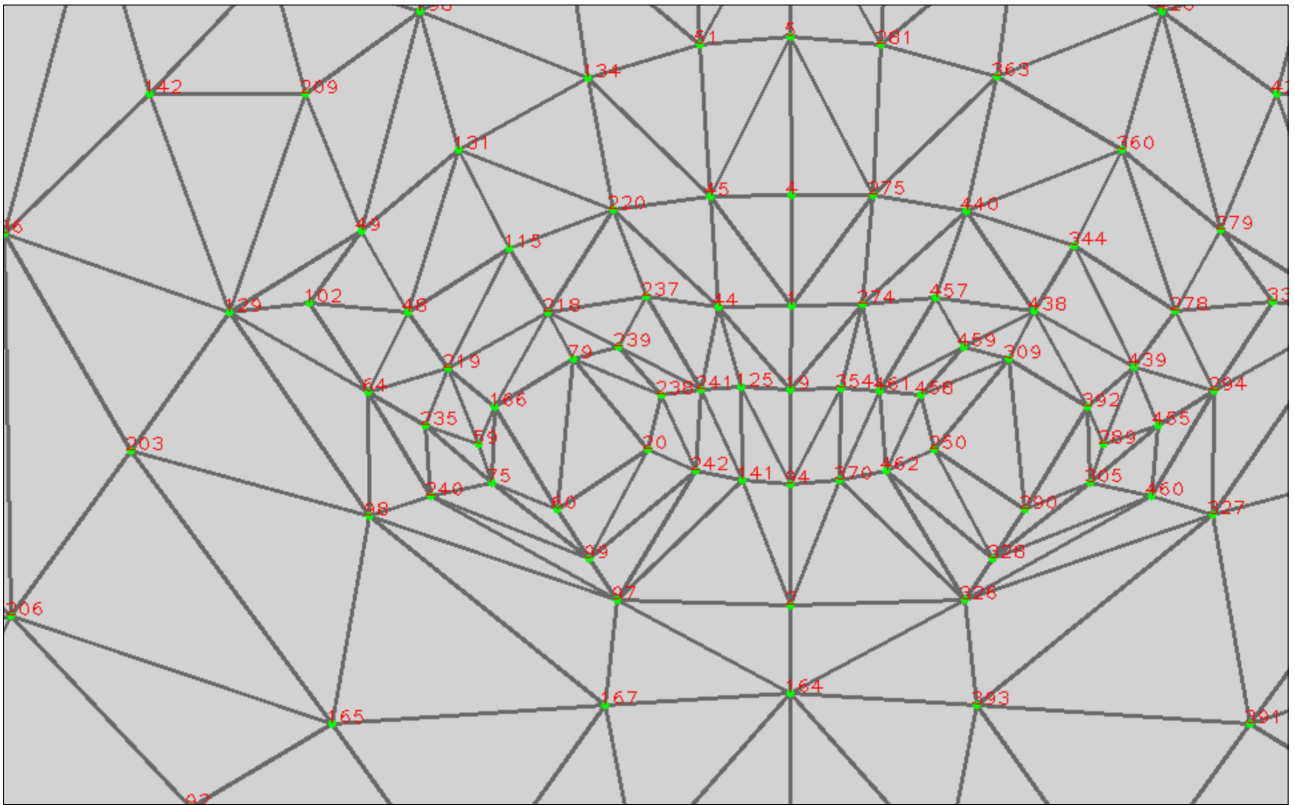


Figure 3: zooming in





The index.html file

Adding the `ml5.js` to the `index.html` file, just as you have done before.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```




Sketch B2.1 our starting sketch

Starting sketch as per normal, except note the canvas size.

```
function setup()
{
  createCanvas(640, 480)
}

function draw()
{
  background(220)
}
```

Notes

You should have a grey canvas **640** by **480**. We will use this size of canvas because we will be using a video feed. You might need to slide the canvas, code divide by clicking and dragging it so you can see the whole canvas.



Sketch B2.2 creating a video from your webcam

We added a video capture function to our basic sketch.

```
let video

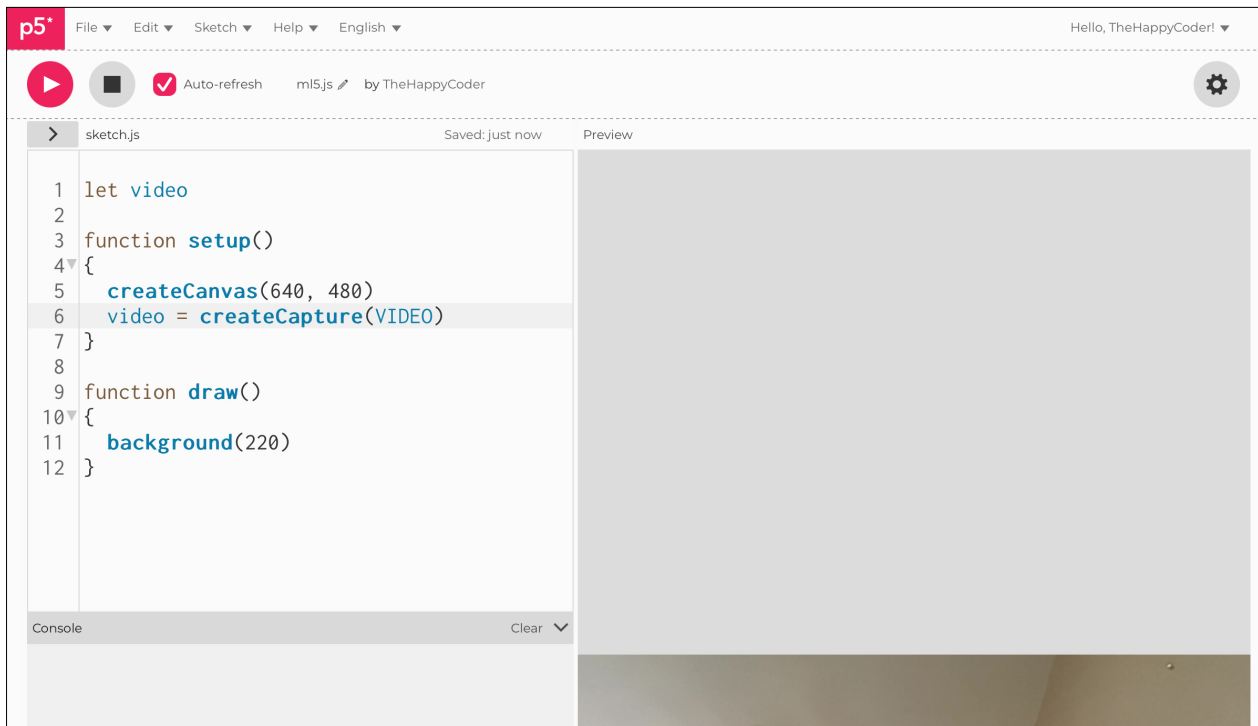
function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO)
}

function draw()
{
  background(220)
}
```

Notes

You should have the grey canvas, and underneath it there will be a video feed from your webcam. You will probably need to give your computer permission to allow p5.js to access the webcam.

Figure B2.2





Sketch B2.3 video on the canvas

As we want the video in the canvas, we will hide the live stream and create an `image()` of the video on the canvas.

```
let video

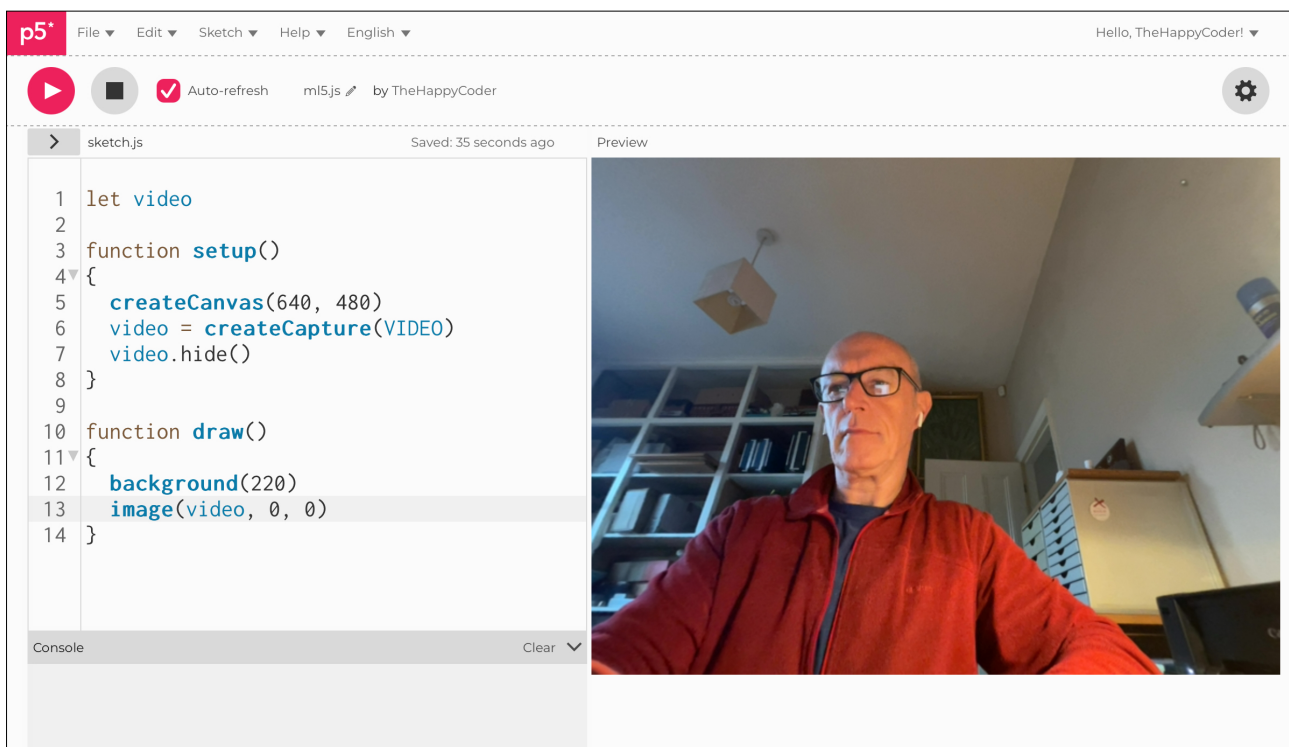
function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

The video stream has disappeared from below the canvas and should now appear on the canvas in the dimensions we are using. The full size is (640, 480).

Figure B2.3





Sketch B2.4 video flipped

To mirror the image, we will flip the video.

```
let video

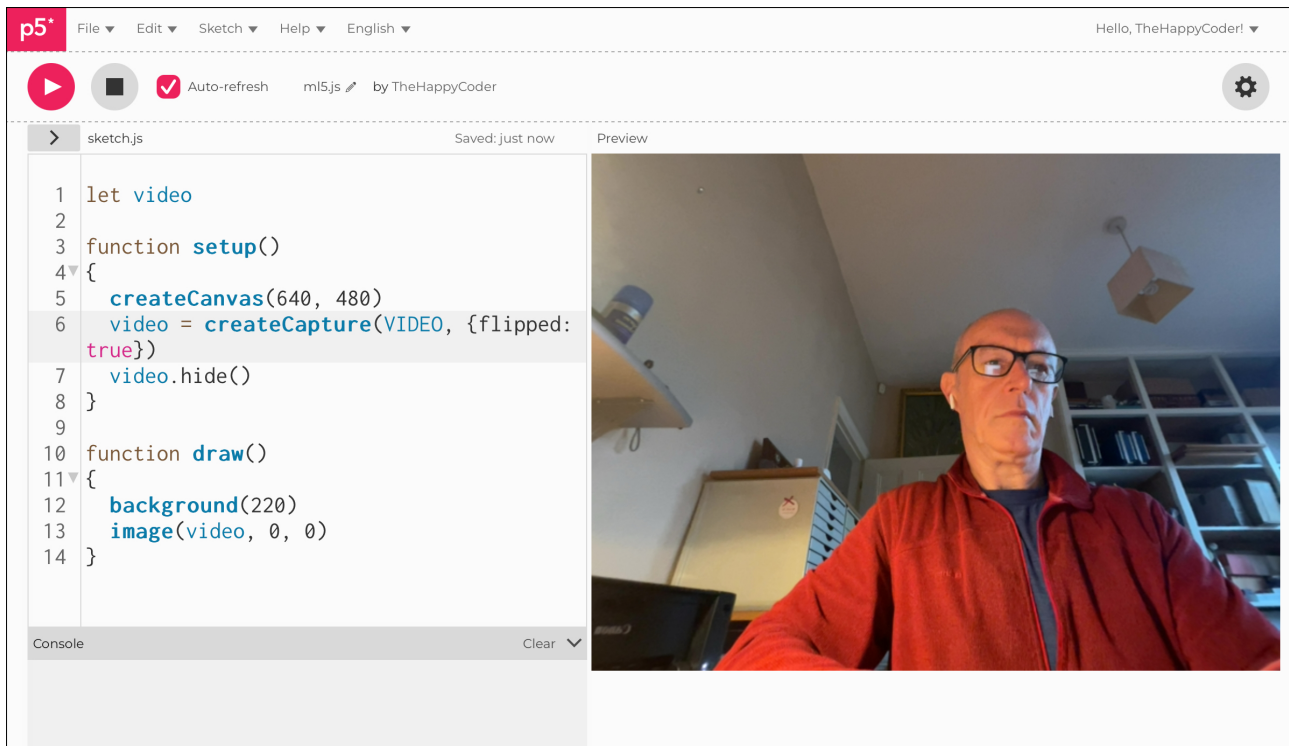
function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

The image should now be reversed or mirrored.

Figure B2.4





Sketch B2.5 faceMesh

Now we want to use the **faceMesh** model, so we need to upload the model asynchronously that is built into ml5.js.

```
let video
let faceMesh

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

Nothing will happen on the canvas just yet; we need to connect the **faceMesh** to the video.

Code Explanation

let faceMesh	Create a variable to hold the face mesh
faceMesh = ml5.faceMesh()	Call the pretrained model faceMesh in ml5.js



Sketch B2.6 callback

We can use a built-in function for detecting a video and any face(s) in it called `detectStart()`. We can specify how many faces to detect; for a single face, use the function `maxFaces` and make it equal to `1`. We will also need a callback function; we will give it the name `gotFaces()`, which will hold the data (`results`) from the `faceMesh`. We will console log the results, which will give us all the data points.

```
let video
let faceMesh

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(220)
  image(video, 0, 0)
}

function gotFaces(results)
{
  console.log(results)
}
```


Notes

This only works for one face. If you want multiple faces, you need to specify the number. For example, five faces would be:

Notice also that it gives an x, y, and z value; this means it is a three-dimensional model of your face.

Challenge

Have a drill down into the results in the console log. You will need to stop the code running to open it up.

Code Explanation

<pre>faceMesh = ml5.faceMesh({maxFaces: 1})</pre>	Option to only pick one face at a time or multiple faces
<pre>faceMesh.detectStart(video, gotFaces)</pre>	A function that detects whether there is a video and if so is there a face in it, then calls the callback function with the data points
<pre>function gotFaces(results)</pre>	The data points are sent to the callback function gotFaces()

Figure B2.6a: console log of data points

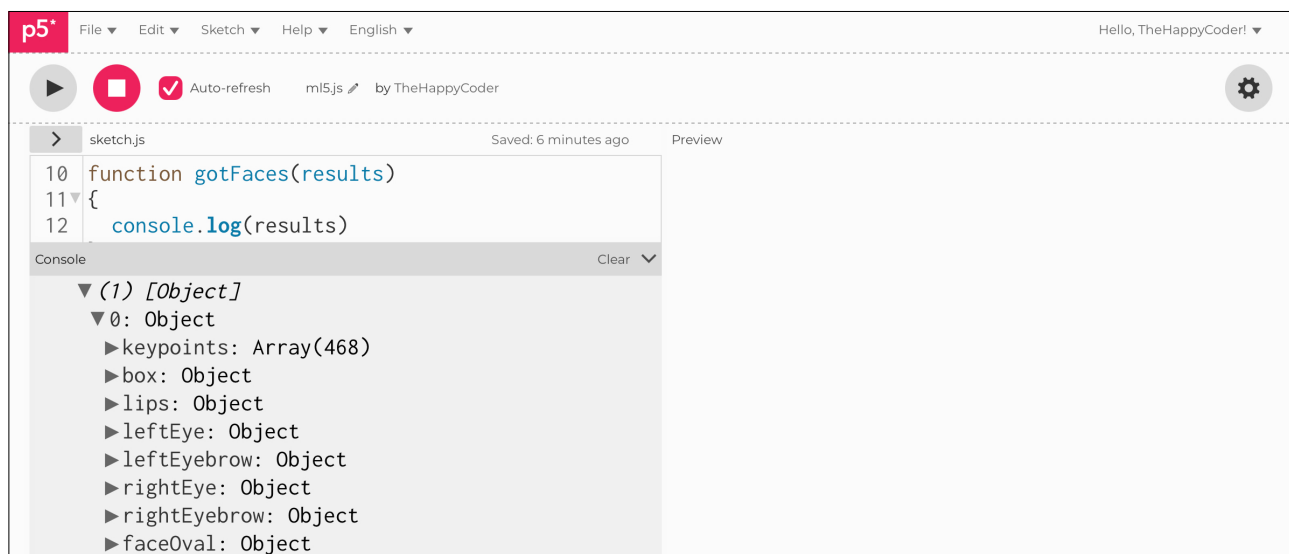
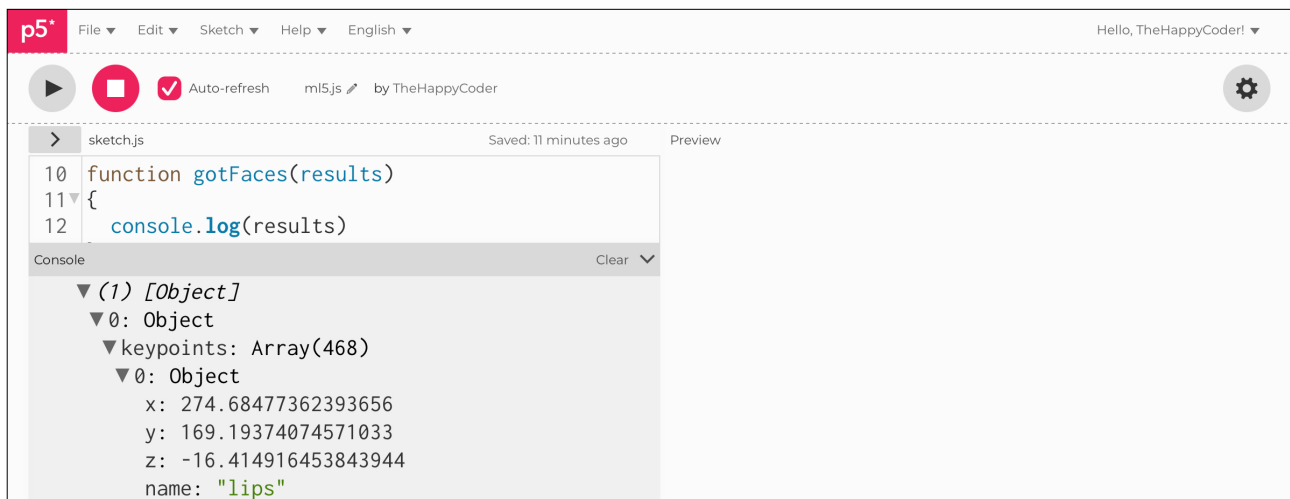


Figure B2.6b: drilling down a bit more





Sketch B2.7 the faces array

We want to save the relevant data in an empty array called **faces**.

! Remove: **console.log()** and replace it by filling the **faces[]** empty array with the data points (**results**).

```
let video
let faceMesh
let faces = []

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(220)
  image(video, 0, 0)
}

function gotFaces(results)
{
  // console.log(results)
  faces = results
}
```

Notes

We are just collecting the data points first before we draw anything on the canvas.

Challenge

Try `console.log(faces)` to see that it now holds the data points, but remember to move that line of code to after this line of code: `faces = results`; otherwise, you might get an error.

Code Explanation

<code>let faces = []</code>	Empty array called faces
<code>faces = results</code>	Filling the array with those data points in the results



Sketch B2.8 collecting the lips

If you open an object in the console, you will see that there is a **keypoints** array with **468** points. These are the total number of points on your face. We only want the lips, and to do that, we create an **if()** loop to collect them and create a bounding box.

```
let video
let faceMesh
let faces = []
let lips

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(220)
  image(video, 0, 0)
  if (faces.length > 0)
  {
    lips = faces[0].lips
    rect(lips.x, lips.y, lips.width, lips.height)
  }
}

function gotFaces(results)
{
  faces = results
}
```

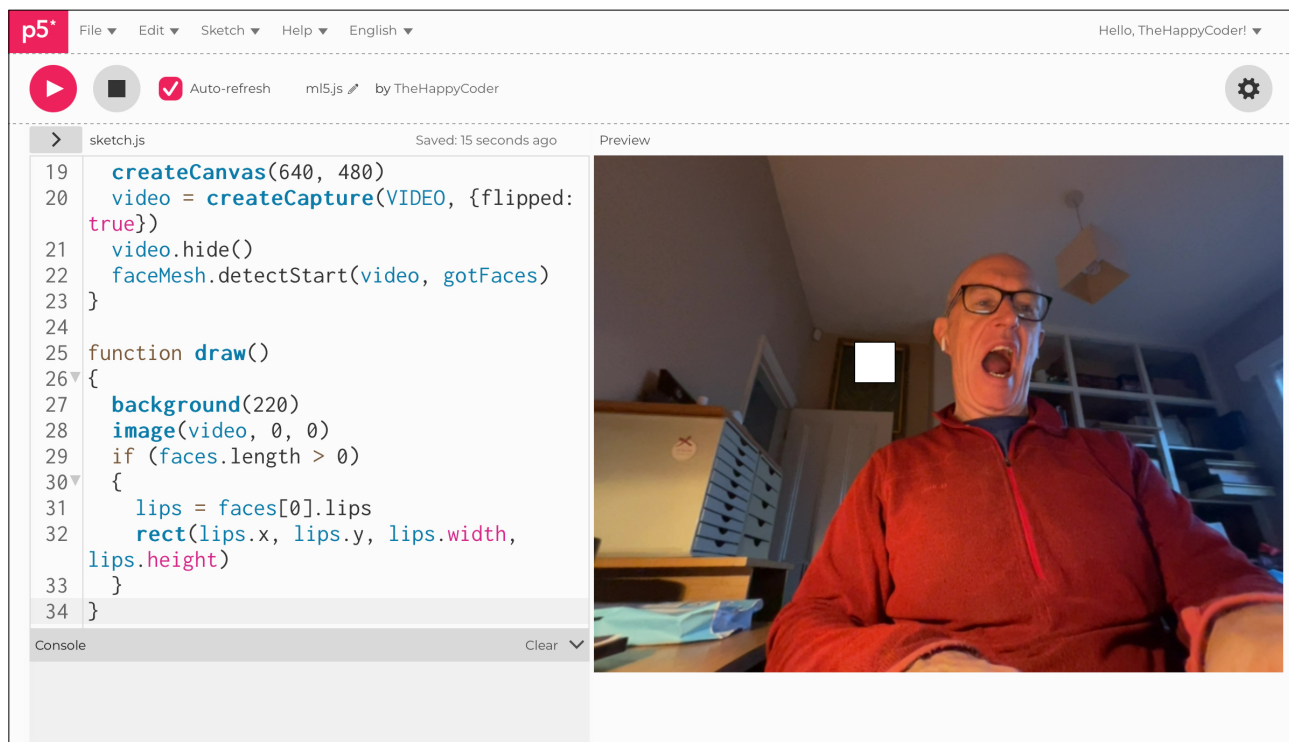
Notes

You get a white box that responds to you opening and closing your mouth. You will notice that the box is not mirrored. We will resolve that in the next sketch.

Code Explanation

<code>if (faces.length > 0)</code>	Checks to see if a face has been detected
<code>lips = faces[0].lips</code>	The lips are the first element in the array
<code>rect(lips.x, lips.y, lips.width, lips.height)</code>	Draws a rectangle with dimensions width and height of lips

Figure B2.8





Sketch B2.9 face mirrored

Here we will be resolving the not mirrored issue and have a rectangle instead of a box.

```
let video
let faceMesh
let faces = []
let lips

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1, flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(220)
  image(video, 0, 0)
  if (faces.length > 0)
  {
    lips = faces[0].lips
    noFill()
    strokeWeight(3)
    stroke(255)
    rect(lips.x, lips.y, lips.width, lips.height)
  }
}

function gotFaces(results)
{
  faces = results
}
```

Notes

You should have a white rectangle following the shape and movement of your mouth.

Challenge

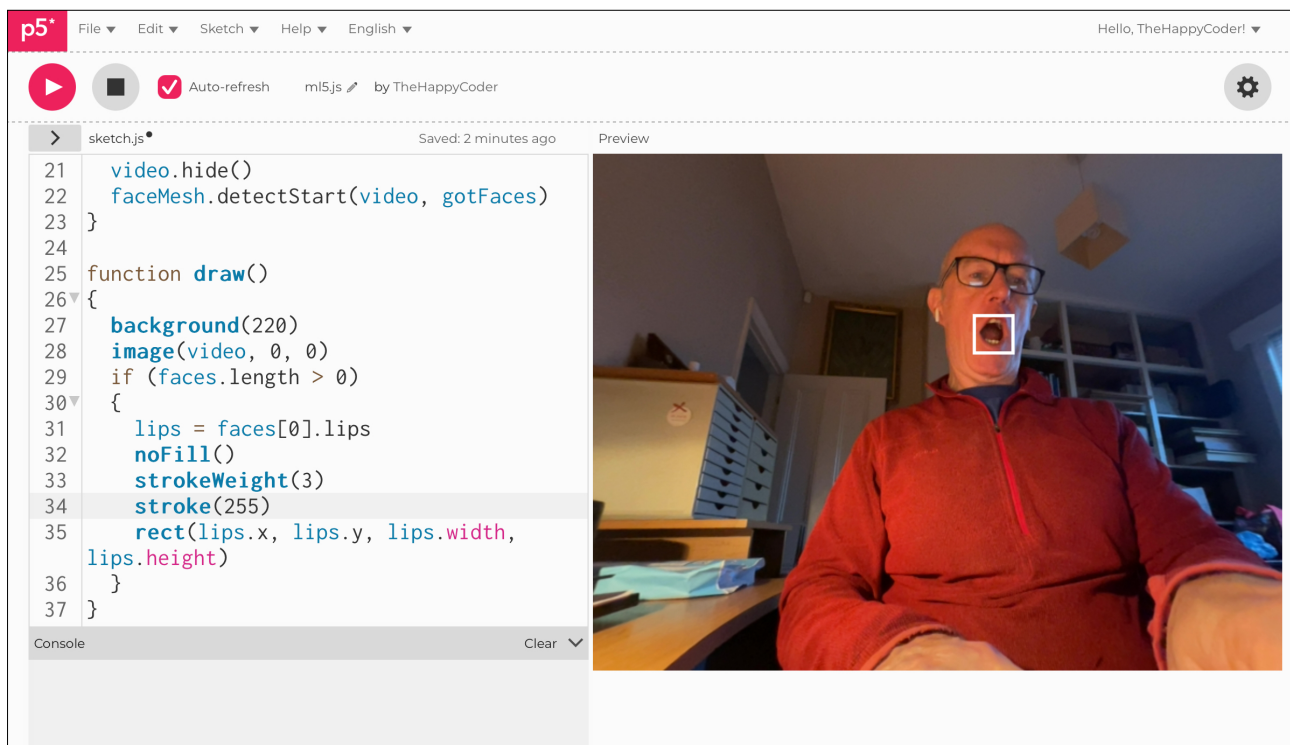
Fancy trying some other shapes?

Code Explanation

```
faceMesh = ml5.faceMesh({maxFaces: 1,  
  flipped: true})
```

faceMesh is mirrored

Figure B2.9





Sketch B2.10 adding eyes

Let's add the eyes; there are labels for the left and right eye called `leftEye` and `rightEye`, respectively. If we look at a `console.log()` of the `results`, we notice that it gives us a `centreX` and a `centreY` value, which are the centres of the respective eyes. If you look at [figure B2.10a](#) below, you can see their reference. We can add them to our sketch and draw circles to those data points.

```
let video
let faceMesh
let faces = []
let lips

let leftEye
let rightEye

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1, flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(220)
  image(video, 0, 0)
  if (faces.length > 0)
  {
    lips = faces[0].lips
    leftEye = faces[0].leftEye
    rightEye = faces[0].rightEye

    noFill()
    strokeWeight(3)
    stroke(255)
    rect(lips.x, lips.y, lips.width, lips.height)
    circle(leftEye.centerX, leftEye.centerY, leftEye.width)
    circle(rightEye.centerX, rightEye.centerY, rightEye.width)
```

```

    }
  }

  function gotFaces(results)
  {
    faces = results
  }

```

Notes

The console log comes in very useful for navigating through all this information. It is often a good idea to comment it out afterwards (or delete it) as it can slow it down.

Challenges

1. Add other features to the image.
2. Can you remove the background image and have just a plain background?

Code Explanation

<code>leftEye = faces[0].leftEye</code>	The variable <code>leftEye</code> takes all the data for that eye
<code>circle(leftEye.centerX, leftEye.centerY, leftEye.width)</code>	We pull out the data for the <code>centerX</code> and <code>centerY</code> for the left eye and draw the circle.

Figure B2.10a: looking at the console.log for the eyes

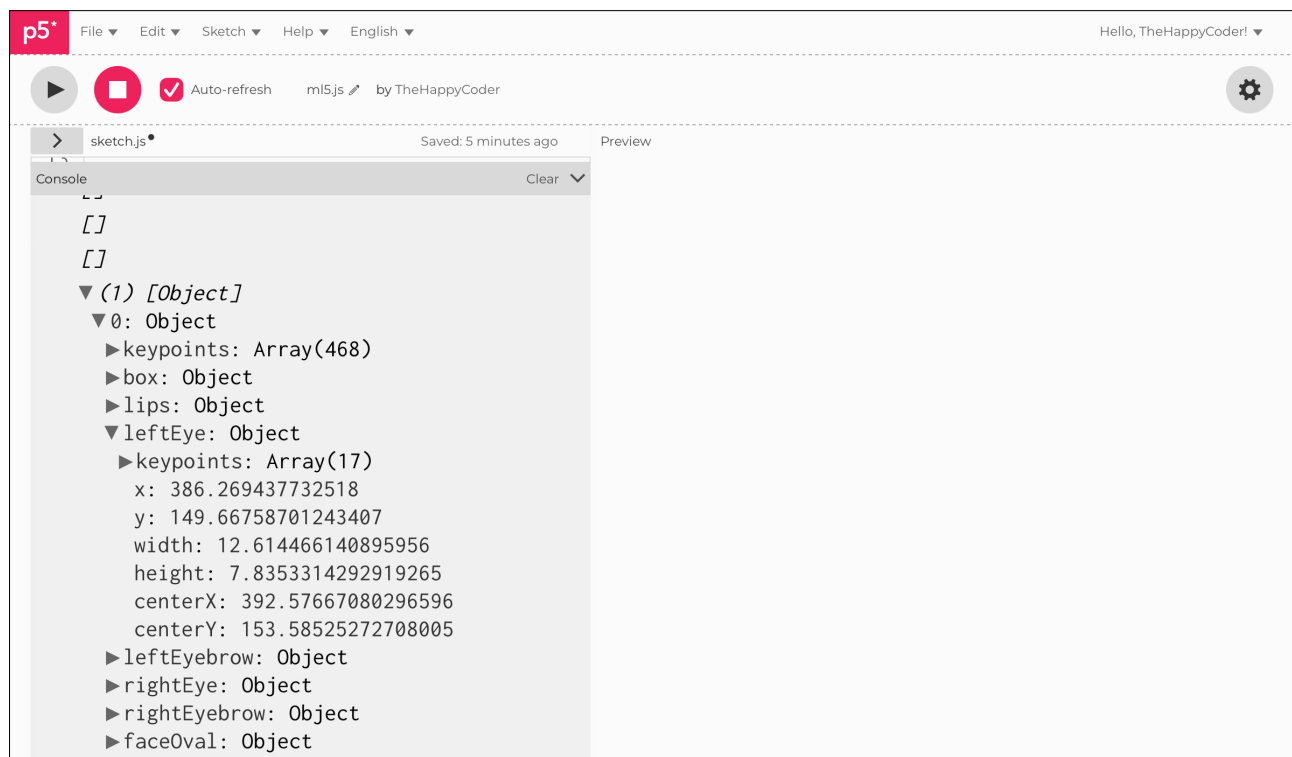
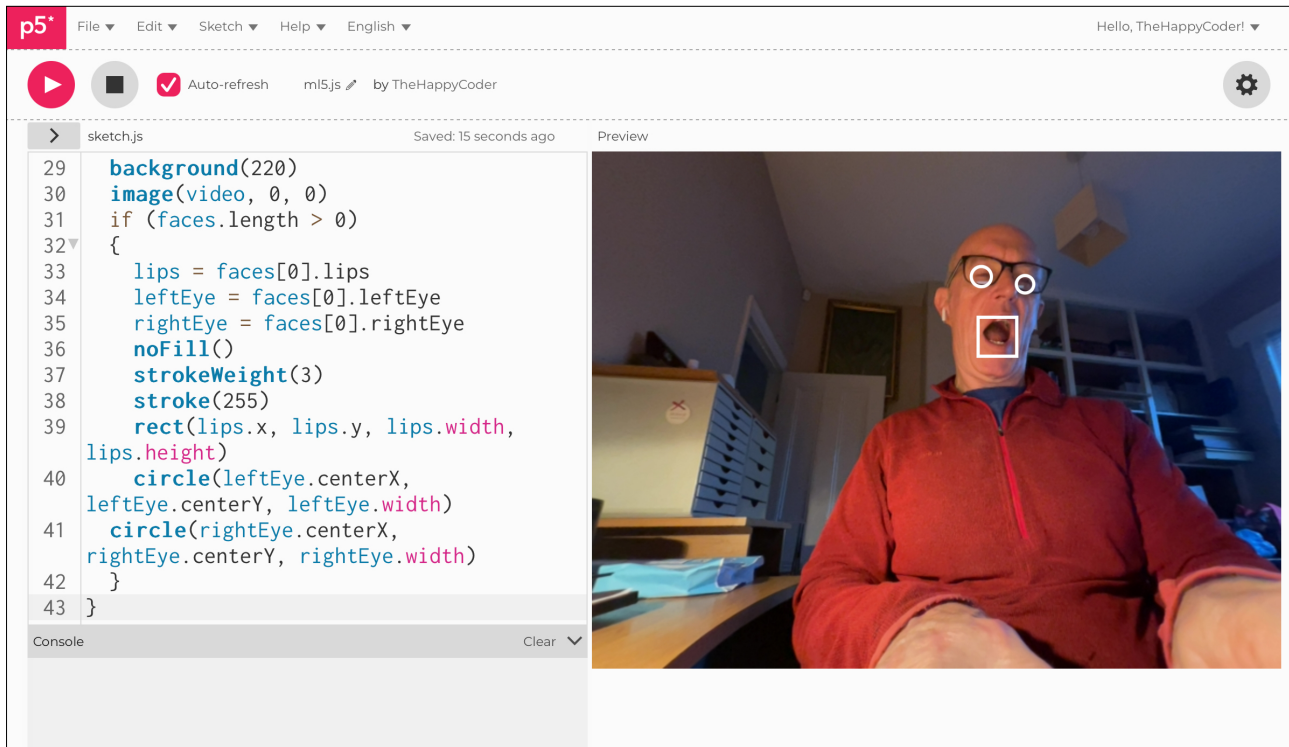


Figure B2.10b





Drawing the data points

In this final part of the unit, we will draw all the data points for the face mesh. We will use the above sketch but first remove much of the code in the `draw()` function. We will be drawing `points()`, not circles; in case you have forgotten, points are the size of pixels, so we will enlarge them slightly.

When you do a `console.log()` of the results, you can see there are **468 keypoints**; these are the values we are after. See **figure 4** below:

Figure 4: `console.log()` of the keypoints

The screenshot shows the p5.js web editor interface. At the top, there's a menu bar with 'File', 'Edit', 'Sketch', 'Help', and 'English'. The user is logged in as 'Hello, TheHappyCoder!'. Below the menu bar, there's a toolbar with a play button, a red square button, a checkmark button, and 'Auto-refresh'. The file name is 'ml5.js' by 'TheHappyCoder'. The editor shows a sketch.js file with the following code:

```
1 let faceMesh
2 let faceMesh
3 let faces = []
```

The console shows the output of a `console.log()` statement. The output is a large object with the following structure:

```
{keypoints: Array(468), box: Object, lips: Object, leftEye: Object, leftEyebrow: Object...}
  ▼ keypoints: Array(468)
    ▼ 0: Object
      x: 384.90838660254184
      y: 209.24978808140517
      z: -15.997210704816007
      name: "lips"
    ▼ 1: Object
      x: 386.5294634202798
      y: 192.2116903279196
      z: -20.54022902607433
    ▼ 2: Object
      x: 385.50599638724185
      y: 198.78253336708528
      z: -13.002282310166411
    ▼ 3: Object
      x: 383.7160663713105
```



Sketch B2.11 a slimmed down version

Delete the lines of code not used in the `draw()` function below as well as the reference to lips and eyes.

```
let video
let faceMesh
let faces = []
// let lips
// let leftEye
// let rightEye

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1, flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(220)
  image(video, 0, 0)
}

function gotFaces(results)
{
  faces = results
}
```

Notes

We have an almost empty `draw()` function.

Sketch B2.12 a new array of objects

We are going to loop through all the data points in the **faces** array. This will give us an array of objects; each object carries with it a bunch of data.

```
let video
let faceMesh
let faces = []

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1, flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(220)
  image(video, 0, 0)
  for (let i = 0; i < faces.length; i++)
  {
    let face = faces[i]
  }
}

function gotFaces(results)
{
  faces = results
}
```

Notes

We have created another array of objects called **face**.

Code Explanation

<code>for (let i = 0; i < faces.length; i++)</code>	Loop through the faces array of objects
<code>let face = faces[i]</code>	Move the objects into a new array called face



Sketch B2.13 keypoint array

Next, we want another loop to go through the face array and pull out all the **keypoints** one by one and draw them as points.

```
let video
let faceMesh
let faces = []

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1, flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(220)
  image(video, 0, 0)
  for (let i = 0; i < faces.length; i++)
  {
    let face = faces[i]
    for (let j = 0; j < face.keypoints.length; j++)
    {
      let keypoint = face.keypoints[j]
      strokeWeight(2)
      stroke(255, 255, 0)
      point(keypoint.x, keypoint.y)
    }
  }
}

function gotFaces(results)
{
  faces = results
}
```



```
}
```

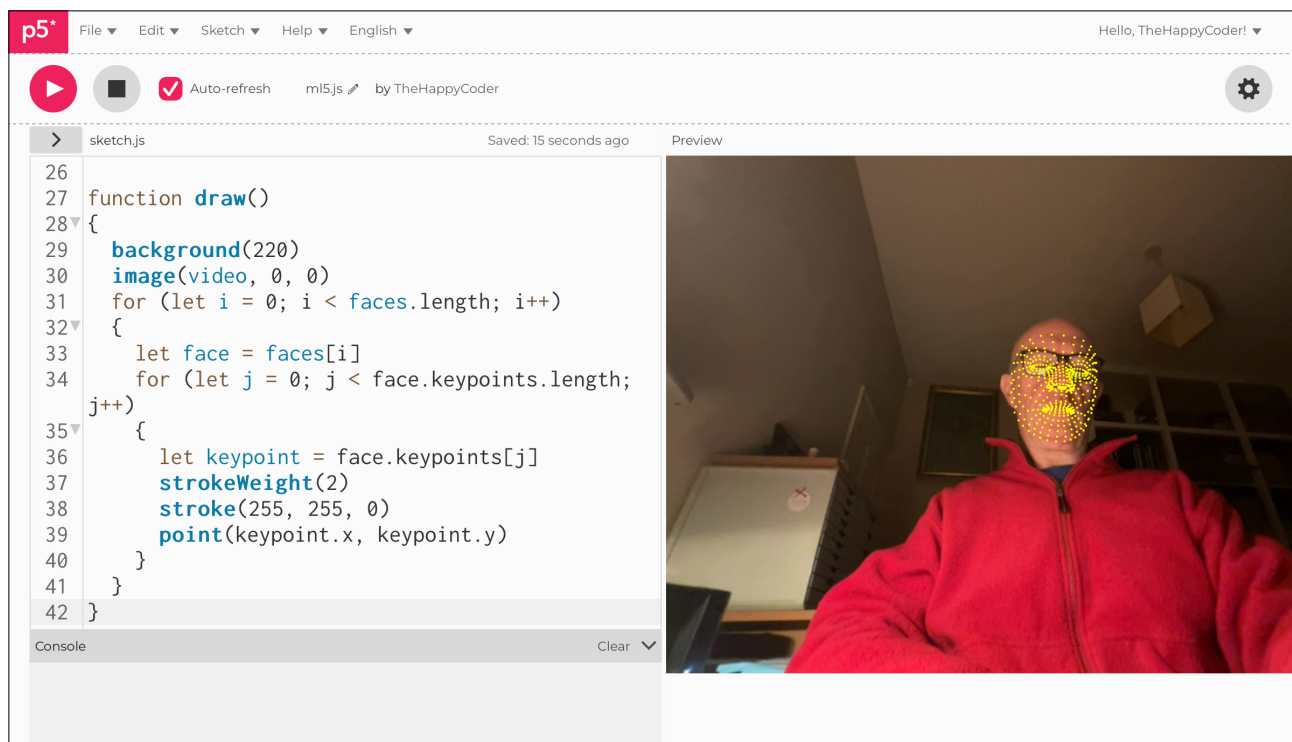
Notes

We cycle through all the **keypoints** and draw a point at each of their **x** and **y** coordinates.

Code Explanation

<pre>for (let j = 0; j < face.keypoints.length; j++)</pre>	Cycle through just the keypoints part of the array
<pre>let keypoint = face.keypoints[j]</pre>	Give those values to a new array called keypoint
<pre>point(keypoint.x, keypoint.y)</pre>	Draw the x and y components of each keypoint as a point (pixel)

Figure B2.13





Sketch B2.14 finally remove the background image

We want a nice, stark contrast.

```
let video
let faceMesh
let faces = []

async function setup()
{
  ml5.setBackend("webgl")
  faceMesh = await ml5.faceMesh({maxFaces: 1, flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  faceMesh.detectStart(video, gotFaces)
}

function draw()
{
  background(0)
  // image(video, 0, 0)
  for (let i = 0; i < faces.length; i++)
  {
    let face = faces[i]
    for (let j = 0; j < face.keypoints.length; j++)
    {
      let keypoint = face.keypoints[j]
      strokeWeight(2)
      stroke(255, 255, 0)
      point(keypoint.x, keypoint.y)
    }
  }
}

function gotFaces(results)
{
  faces = results
}
```

```
}
```

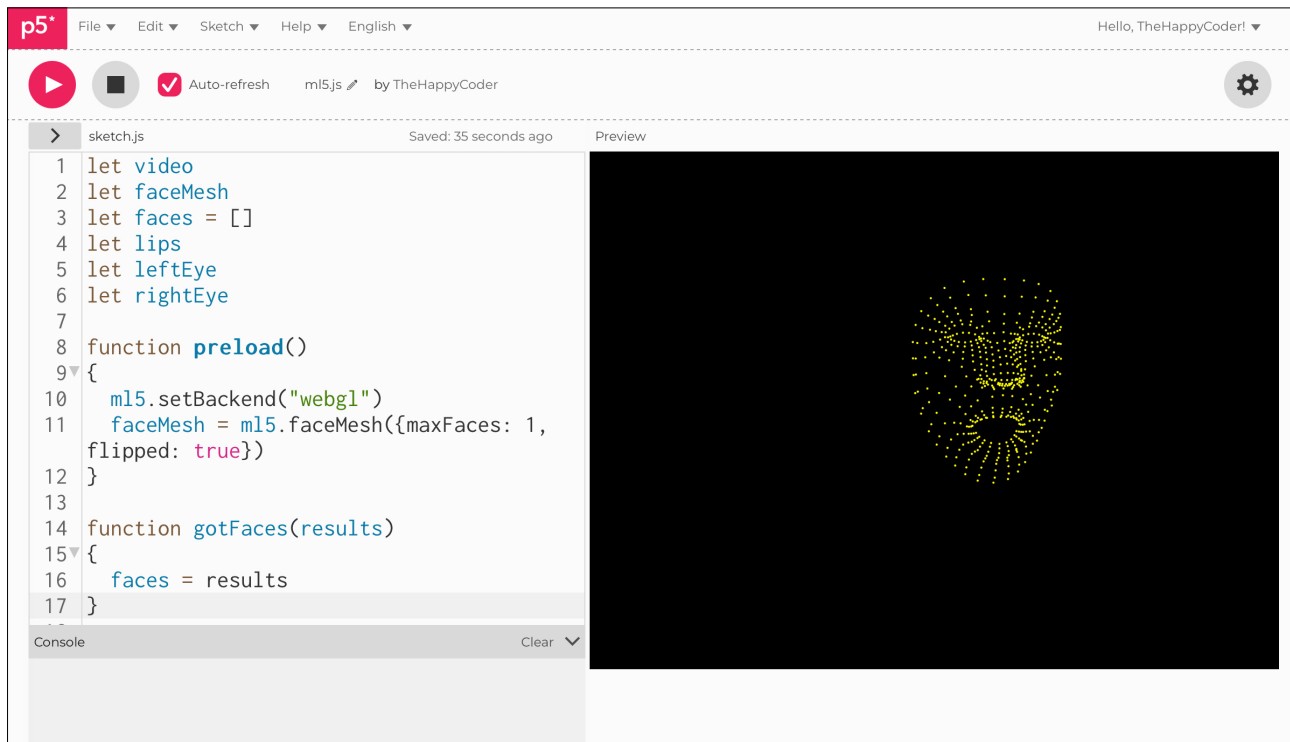
Notes

Our final effect is quite something.

Challenge

What else could you develop or create? 3D, perhaps.

Figure B2.14



Algorithmic Intelligence Module B Unit #3 pretrained handPose

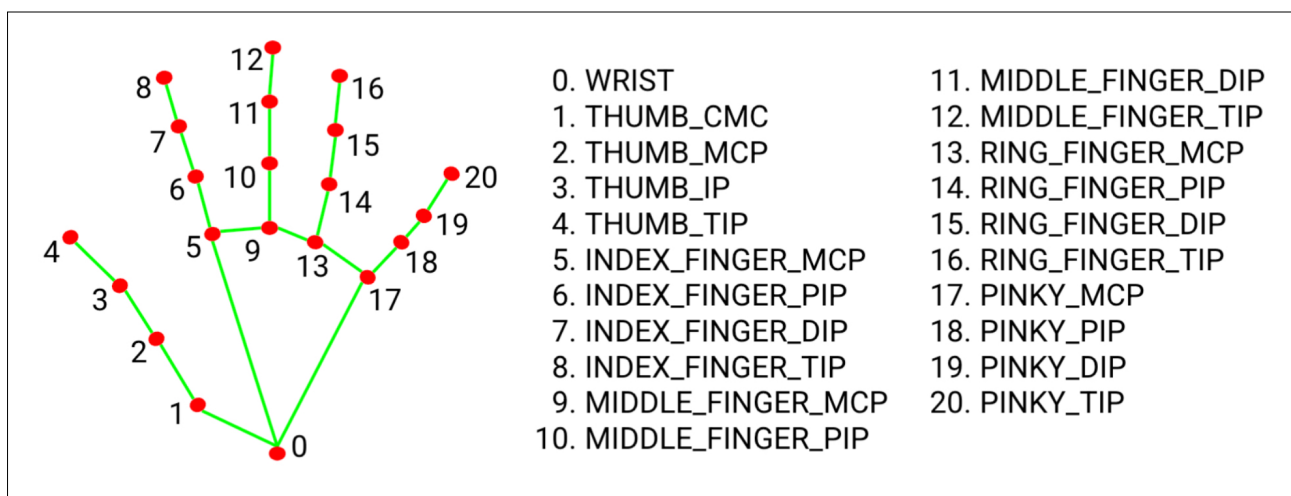


Module B Unit #3: handPose with ml5.js

The **handPose** model is another pre-trained model that is part of the ml5.js library. We can get all the data points of your hands and use some of them to control the size and position of a circle.

The **handPose** model can detect **21** keypoints on a hand (see **figure 1** below). It can detect both hands at the same time and determine the left from the right hand. Also, the keypoints are given as **3D** co-ordinates, but we will focus on the **2D** co-ordinates for now.

Figure 1 all 21 keypoints for handPose





The index.html file

Adding the ml5.js file to the index.html file.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch B3.1 our starting sketch

The process is similar to the previous unit, but we will still go through it step by step.

```
function setup()
{
  createCanvas(640, 480)
}

function draw()
{
  background(220)
}
```



Sketch B3.2 video

Getting the video feed from the webcam.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO)
}

function draw()
{
  background(220)
}
```

Notes

You should get a grey canvas and underneath it, a larger video image of you.



Sketch B3.3 putting you on the canvas

We hide the video stream and use the `image()` function. Notice that we include the width and height, not entirely necessary in this case, but if you wanted to match the canvas dimensions, this is the way to go.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

Now you should have a video image of yourself where the canvas was, but if you lift up your hand, it is not mirrored.



Sketch B3.4 flipping the video

We can flip the video using a function in p5.js.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

You will now have a mirror image from the camera.



Sketch B3.5 the handPose model

There is a pre-trained model within ml5.js, and all we need to do is call it. We will preload it before we try to use it; otherwise, we may get error messages.

```
let video
let handPose

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

Nothing will happen when you hold your hand up to the camera because we haven't connected the two together.



Sketch B3.6 start detecting

We use a function called `detectStart()`, and this will connect the model to the video feed.

```
let video
let handPose

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

! If you run this now, you will get an error message. It will tell you that you need a callback function. This callback function will hold the data.



Sketch B3.7 callback

Let's create a callback function called `gotHands()` and `console.log()` the `results` as we did before.

```
let video
let handPose

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}

function gotHands(results)
{
  console.log(results)
}
```

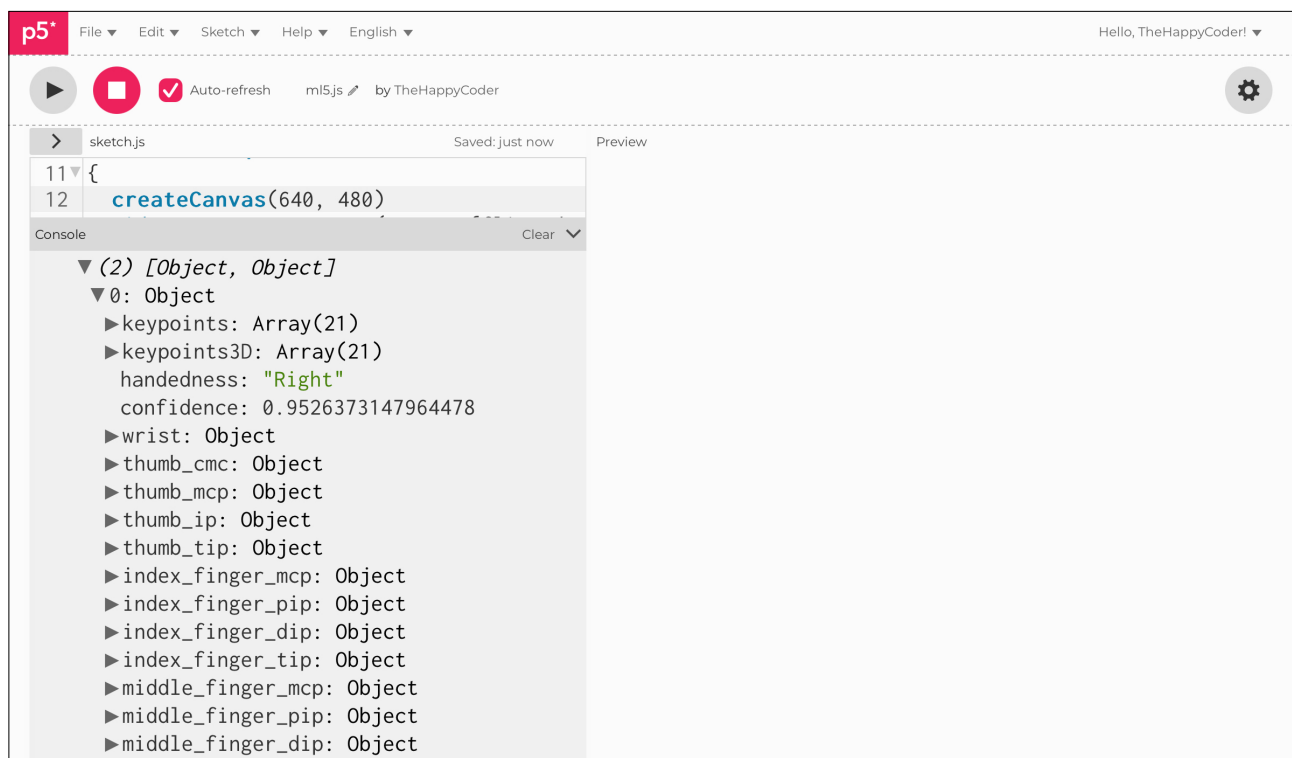
Notes

You will get an empty array when it doesn't see any hands, one array object when you hold up one hand, and two array objects when it can see two hands. If you pause the programme, you can examine the content of the arrays. You will notice how many points there are and the labelling of your fingers, plus other parts of your hand. You have **21 keypoints** for each hand.

Challenge

Highly recommend that you explore the results in the console.

Figure B3.7





Sketch B3.8 an array of hands

We need to store this data and access the array of your hands.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}

function gotHands(results)
{
  hands = results
}
```

Notes

We create an empty array to put the data in and fill it with the results in the callback function.



Sketch B3.9 the keypoints

We want to cycle through all the **keypoints** in the hands array and draw them. For this, we use a nested loop.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose()
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

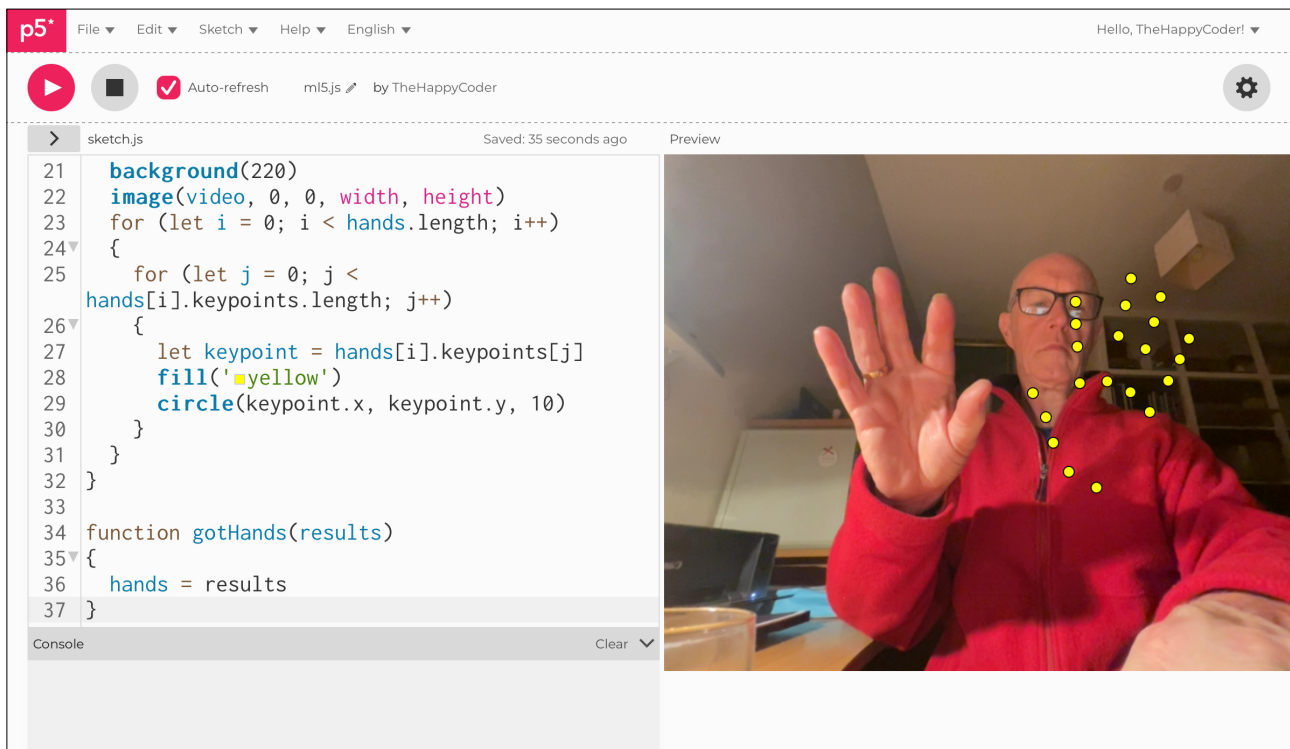
function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  for (let i = 0; i < hands.length; i++)
  {
    for (let j = 0; j < hands[i].keypoints.length; j++)
    {
      let keypoint = hands[i].keypoints[j]
      fill('yellow')
      circle(keypoint.x, keypoint.y, 10)
    }
  }
}

function gotHands(results)
{
  hands = results
}
```


Notes

Notice that the yellow dots are not mirrored!

Figure B3.9





Sketch B3.10 mirroring the yellow dots

We can add the **flipped** is **true**.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

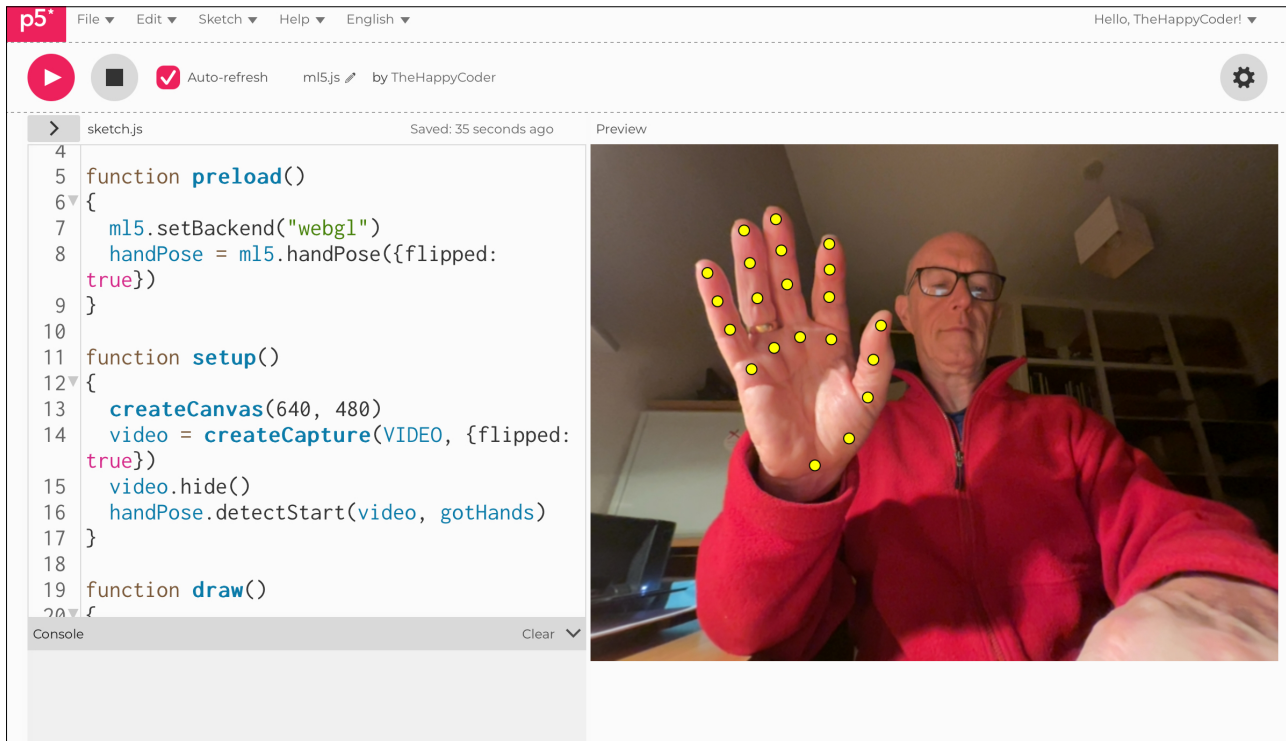
function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  for (let i = 0; i < hands.length; i++)
  {
    for (let j = 0; j < hands[i].keypoints.length; j++)
    {
      let keypoint = hands[i].keypoints[j]
      fill('yellow')
      circle(keypoint.x, keypoint.y, 10)
    }
  }
}

function gotHands(results)
{
  hands = results
}
```

Notes

You can now see the points for both hands.

Figure B3.10





Sketch B3.11 reference keypoints

Instead of points, let's get the index reference for each point, because then we can use specific **keypoints**.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  for (let i = 0; i < hands.length; i++)
  {
    for (let j = 0; j < hands[i].keypoints.length; j++)
    {
      let keypoint = hands[i].keypoints[j]
      fill('yellow')
      // circle(keypoint.x, keypoint.y, 10)
      text(i, keypoint.x -10, keypoint.y)
      text(j, keypoint.x, keypoint.y)
    }
  }
}

function gotHands(results)
{
  hands = results
}
```

```
}
```

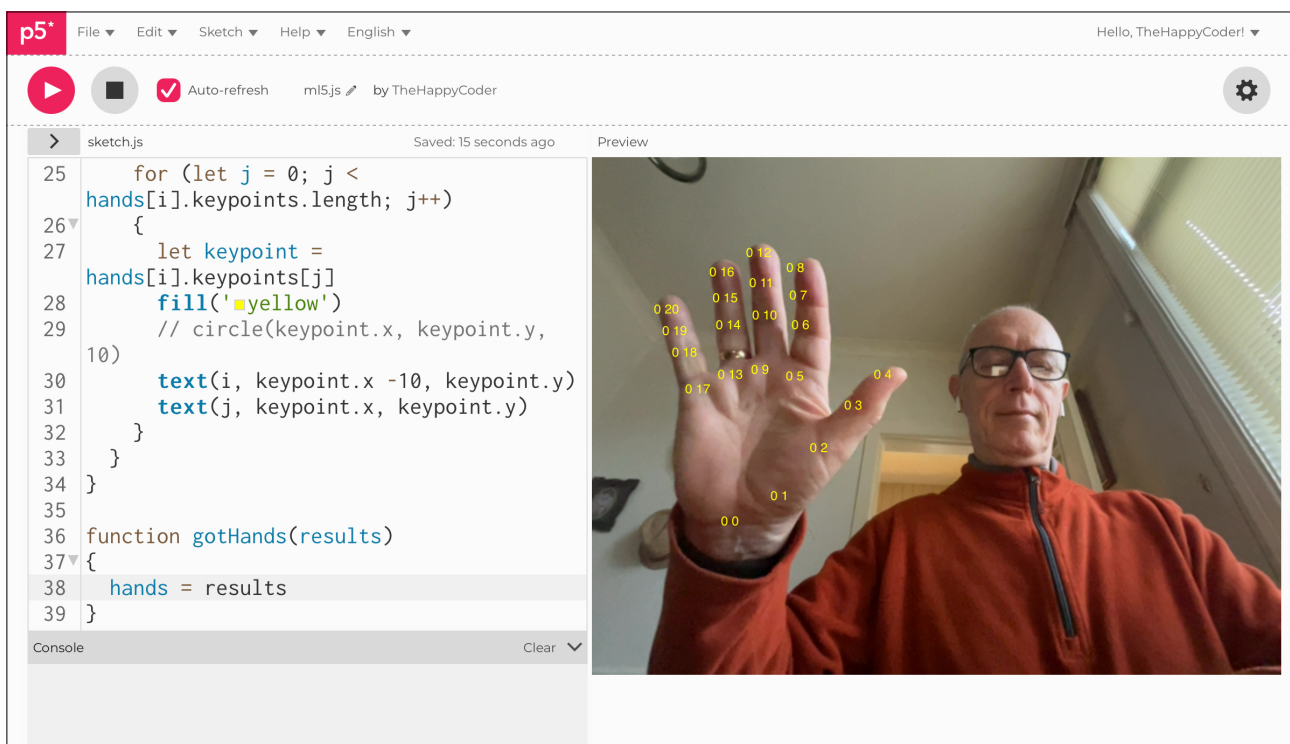
Notes

You will notice that when you hold up two hands, you get a **1** for the left hand followed by the reference number on that hand, and a **0** for the right hand followed by the corresponding reference number on the other hand.

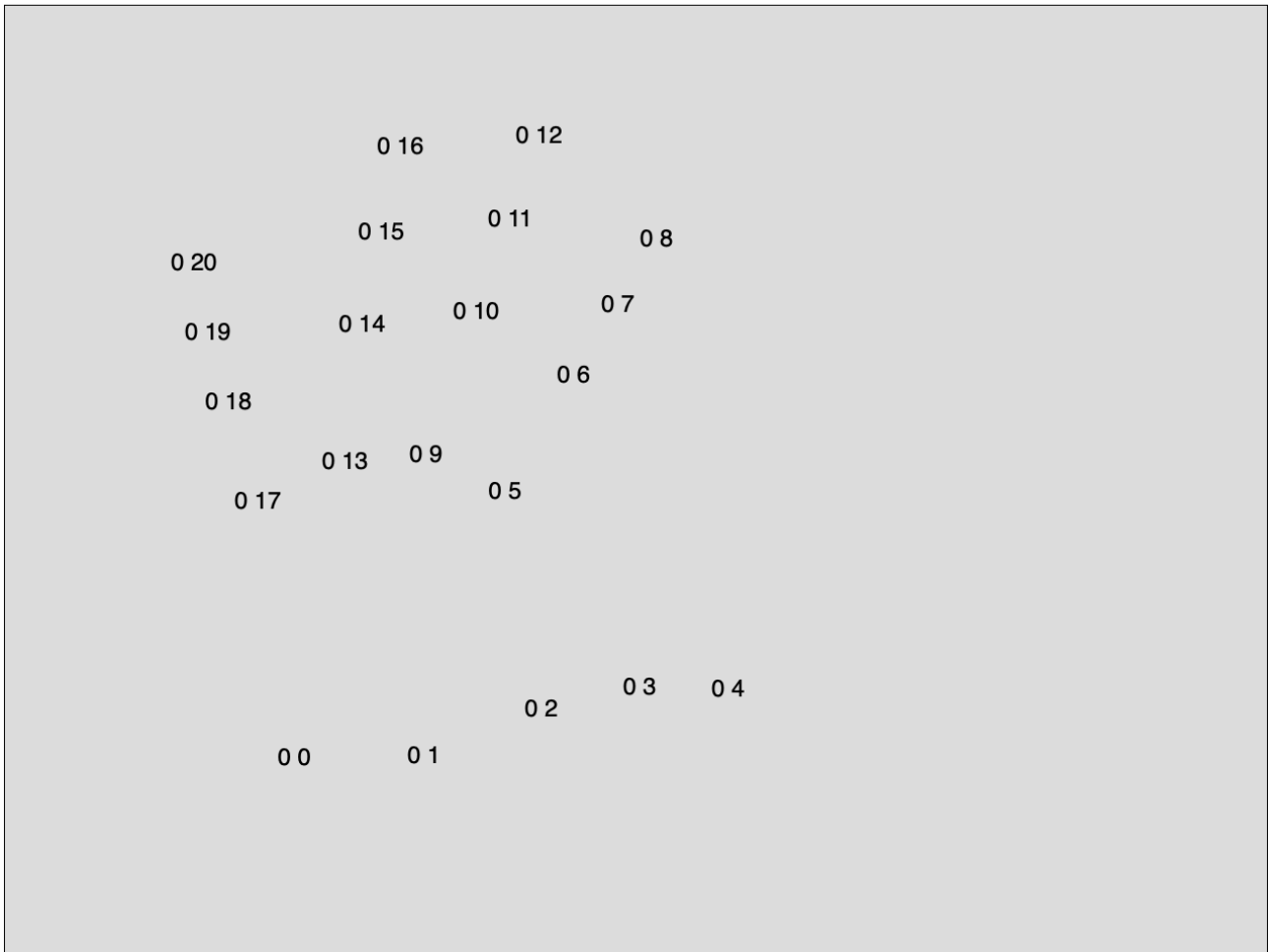
We want the index finger and thumb (below are the reference numbers):

Index finger: keypoint **8**
Thumb: keypoint **4**

Figure B3.11



If you remove the video





Sketch B3.12 index and thumb

Just to prove we have the right keypoint reference, we can write the words onto the canvas. We are only interested in one hand at a time. We can remove the commented-out nested loops.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  if (hands.length > 0)
  {
    fill('white')
    let index = hands[0].keypoints[8]
    let thumb = hands[0].keypoints[4]
    text('index', index.x, index.y)
    text('thumb', thumb.x, thumb.y)
  }
  // for (let i = 0; i < hands.length; i++)
  // {
  //   for (let j = 0; j < hands[i].keypoints.length; j++)
  //   {
  //     let keypoint = hands[i].keypoints[j]
  //     fill('yellow')
  //     // circle(keypoint.x, keypoint.y, 10)
  //   }
  // }
}
```

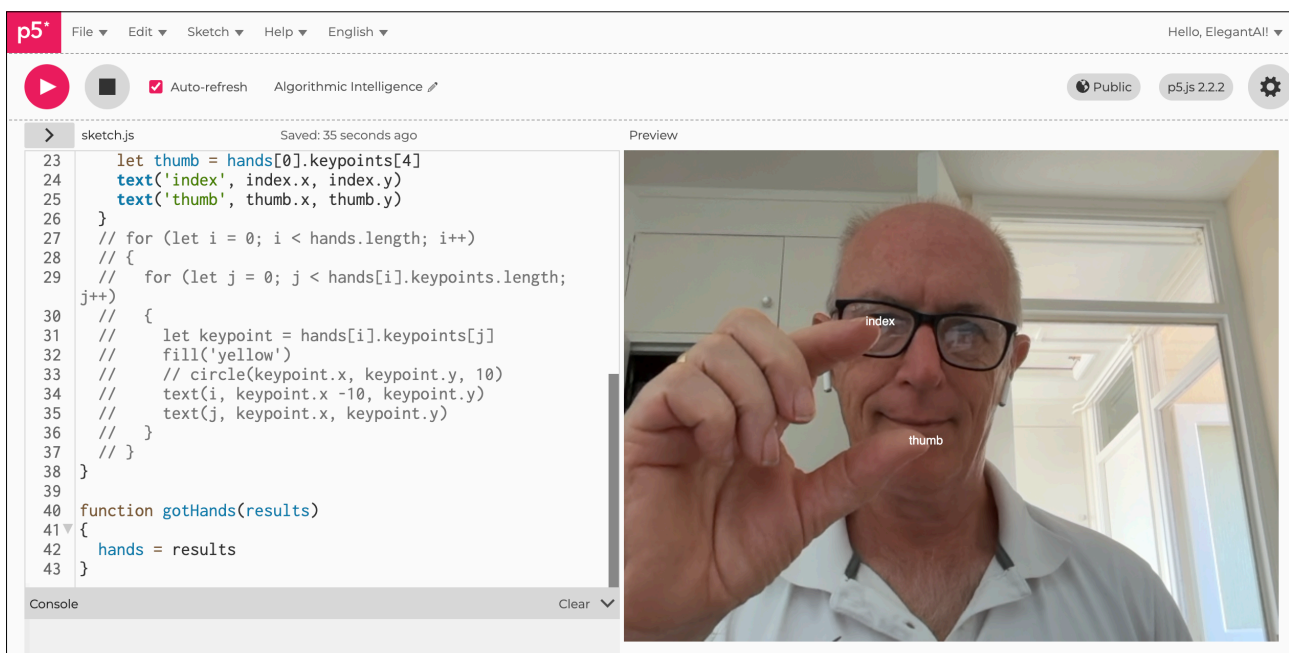
```
//    text(i, keypoint.x -10, keypoint.y)
//    text(j, keypoint.x, keypoint.y)
//  }
// }
}
```

```
function gotHands(results)
{
  hands = results
}
let video
```

Notes

You should have the index finger and thumb correctly identified using either hand.

Figure B3.12





Sketch B3.13 circle size

Instead of writing text, we will draw a circle that changes depending on the distance (**d**) between those two fingers.

! We have completely removed the nested loops as we don't need that information anymore.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  if (hands.length > 0)
  {
    // fill('white')
    noFill()
    strokeWeight(5)
    stroke('yellow')
    let index = hands[0].keypoints[8]
    let thumb = hands[0].keypoints[4]
    let d = dist(index.x, index.y, thumb.x, thumb.y)
    circle(index.x, index.y, d)
    // text('index', index.x, index.y)
    // text('thumb', thumb.x, thumb.y)
  }
}
```

```
function gotHands(results)
{
  hands = results
}
```

Notes

What you get is the circle changing size as you open and close your thumb and index finger. However, I have left the centre on the index finger for now. We want it halfway between the index finger and the thumb.

Challenge

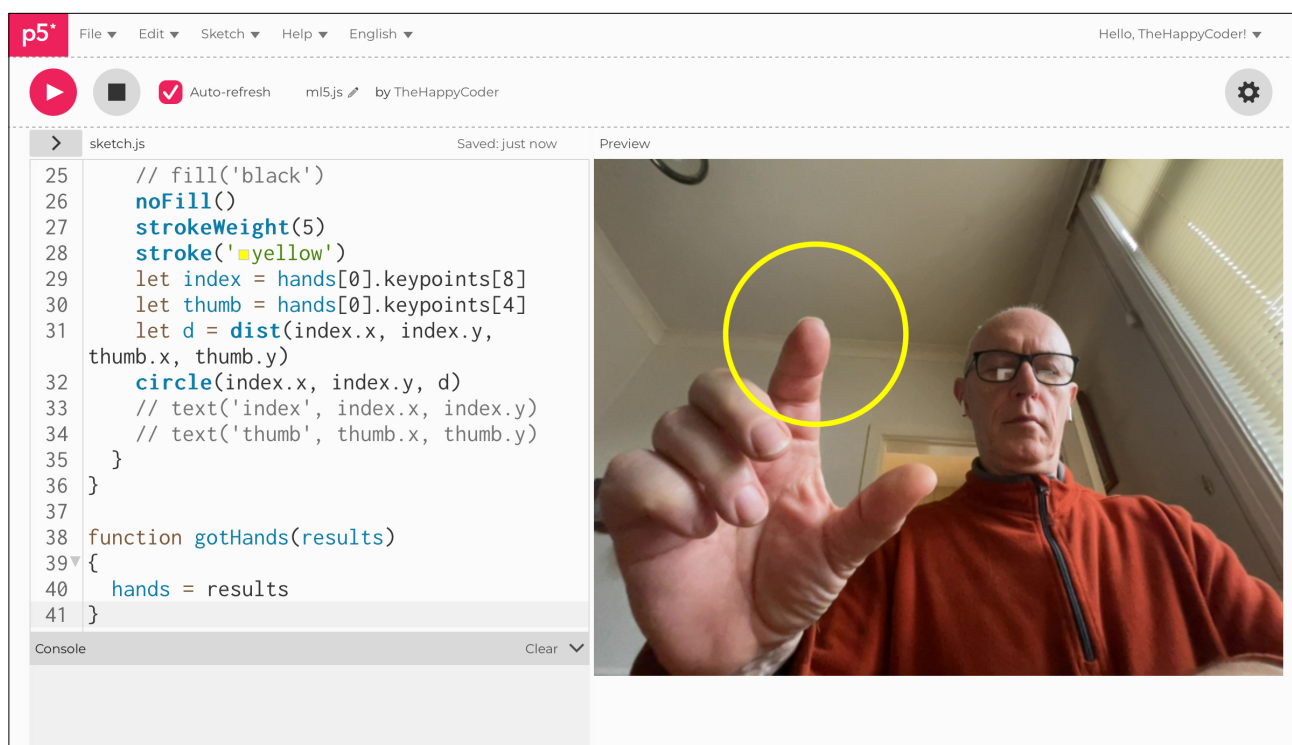
You could try other fingers.

Code Explanation

```
let d = dist(index.x, index.y,
thumb.x, thumb.y)
```

Measures the distance between the x, y co-ordinates of the index finger with the x, y co-ordinates of the thumb

Figure B3.13





Sketch B3.14 centring the circle

Now we have the centre of the circle between our thumb and index finger.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  if (hands.length > 0)
  {
    noFill()
    strokeWeight(5)
    stroke('yellow')
    let index = hands[0].keypoints[8]
    let thumb = hands[0].keypoints[4]
    let posX = (index.x + thumb.x)/2
    let posY = (index.y + thumb.y)/2
    let d = dist(index.x, index.y, thumb.x, thumb.y)
    circle(posX, posY, d)
  }
}

function gotHands(results)
{
  hands = results
}
```

```
}
```

Notes

As you open and close your thumb and index finger, you should get the circle opening and closing with it.

Challenge

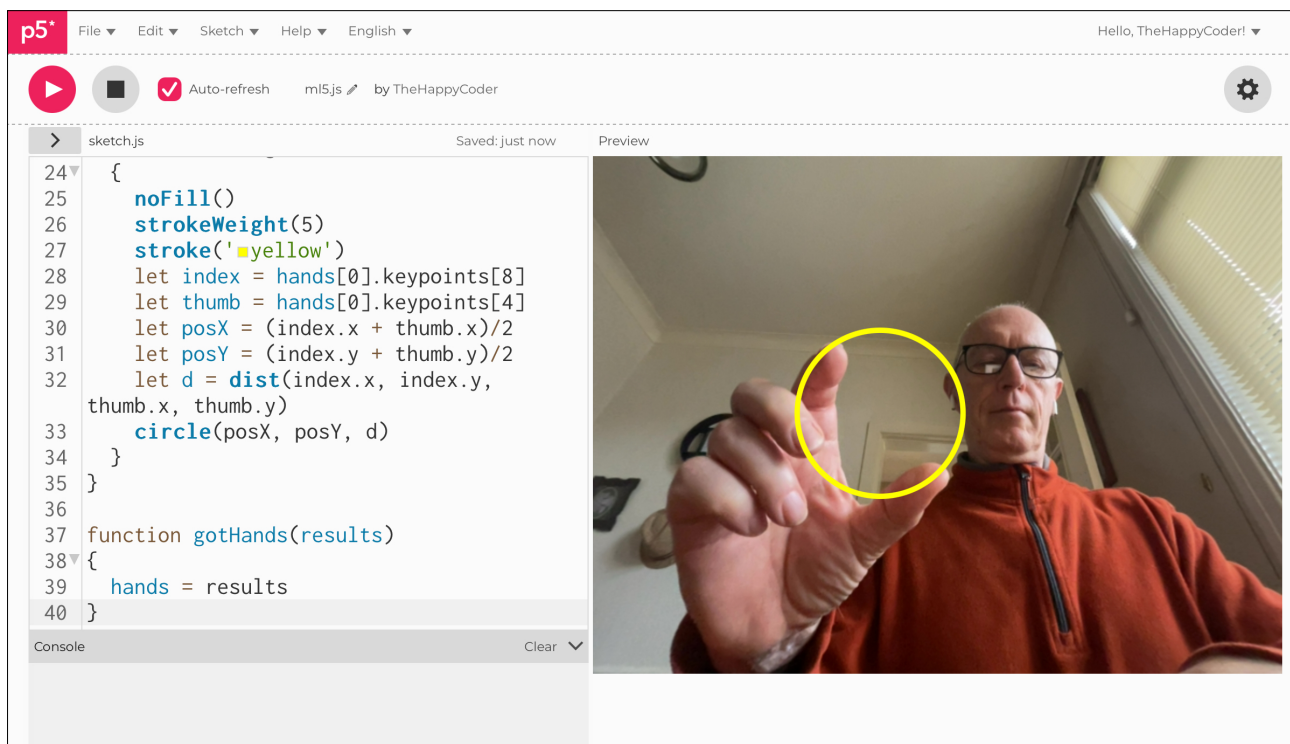
How about a square, rectangle, or ellipse?

Code Explanation

```
let posX = (index.x + thumb.x)/2
```

Halving the distance between them to get the centre

Figure B3.14





Sketch B3.15 using two hands

Just for the fun of it. Let's see if we can control a circle with our two index fingers.

```
let video
let handPose
let hands = []

async function setup()
{
  ml5.setBackend("webgl")
  handPose = await ml5.handPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.hide()
  handPose.detectStart(video, gotHands)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  if (hands[0] && hands[1])
  {
    noFill()
    strokeWeight(5)
    stroke('yellow')

    let indexL = hands[0].keypoints[8]
    let indexR = hands[1].keypoints[8]
    let posX = (indexL.x + indexR.x)/2
    let posY = (indexL.y + indexR.y)/2
    let d = dist(indexL.x, indexL.y, indexR.x, indexR.y)

    circle(posX, posY, d)
  }
}

function gotHands(results)
{
  hands = results
}
```

```
}
```

Notes

You will need to go through the array for both hands, then pull out the relevant reference for the index fingers. Again, this is where the console log can help us.

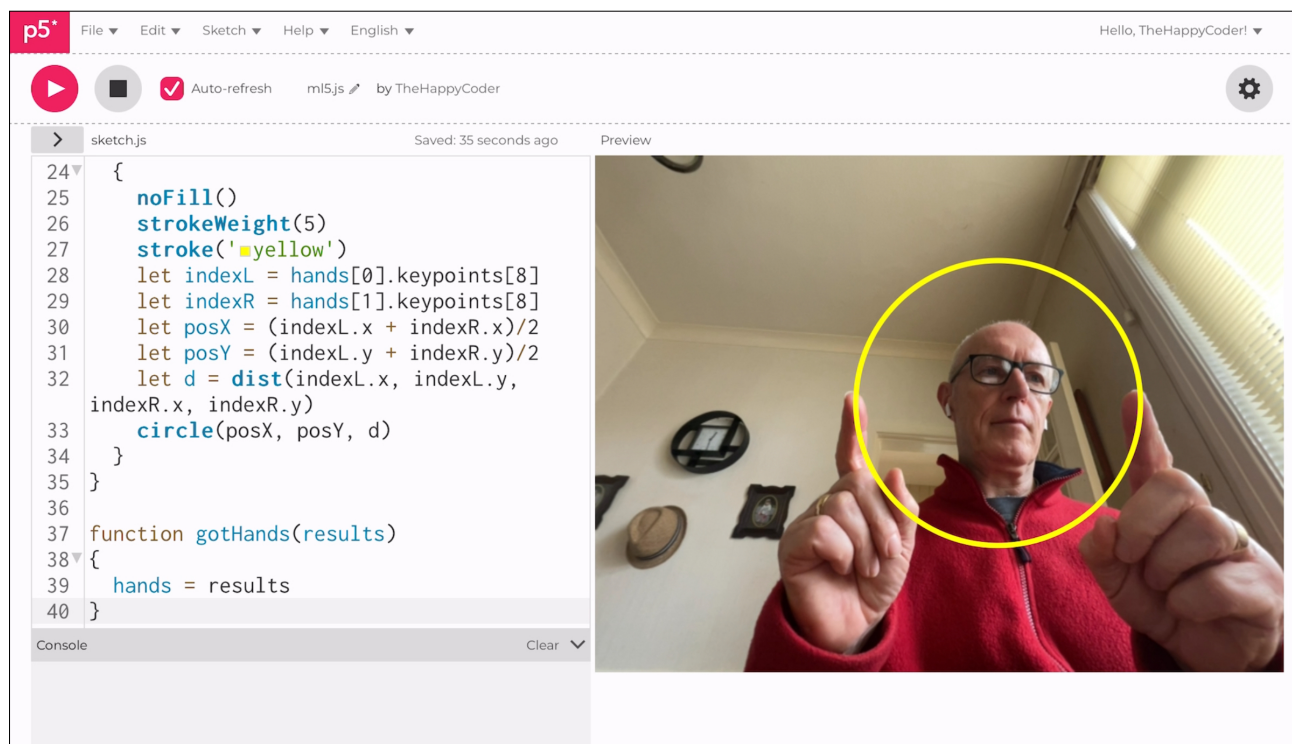
Challenge

What other effects could you create?

Code Explanation

<code>if (hands[0] && hands[1])</code>	Check to see if there are two hands
<code>let indexL = hands[0].keypoints[8]</code>	Get the key point for the left hand
<code>let indexR = hands[1].keypoints[8]</code>	Get the key point for the right hand

Figure B3.15



Algorithmic Intelligence Module B Unit #4 pre-trained bodyPose



Module B Unit #4: bodyPose with ml5.js

This is another pre-trained model to use. It picks out the main body points, wrist, knee, elbow, eyes, and so on. You can draw them and watch them move with you; you can also connect them as a skeleton.

One of the uses of this is to analyse movement, especially in sport where you can compare the running action of one runner versus another to identify differences, or in football, golf, or tennis where there may be tweaks that can be made to improve performance.

By default, the **bodyPose** model uses the **MoveNet** version; this has **17** body keypoints, as shown in the diagram below. If you don't specify the version, it defaults to MoveNet. The alternative is **BlazePose**, which has **33** body keypoints and also has a **3D** option (x, y, z). We will use the **MoveNet** for this unit.

Figure 1 BodyPose('moveNet')

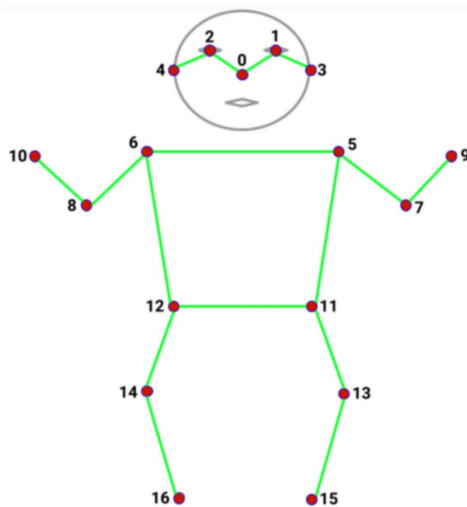
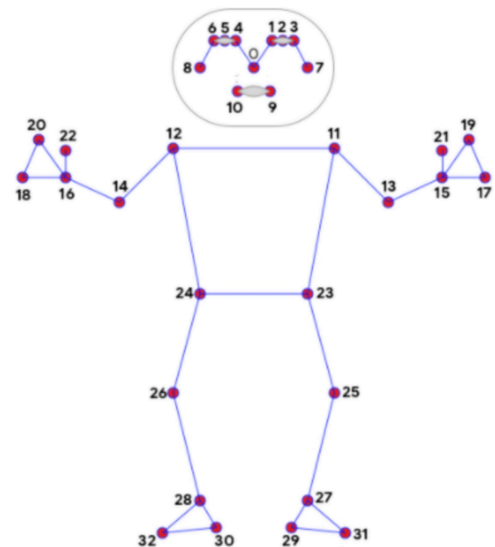


Figure 2 BodyPose('blazePose')





The index.html file

Adding the line of code for ml5.js

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch B4.1 let us begin

You should be pretty familiar with this now. We start with our webcam video stream, as we have done in the previous units.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

You should have an image on your screen where the canvas should be. The canvas will be **640 x 480**. You can adjust the separating line between the code and the canvas by dragging it.



Sketch B4.2 preload

We want to preload the pre-trained model so that it is running before we attempt to do anything with it; otherwise, if there is a slight lag, you will get error messages because it cannot find it. We flipped the **bodyPose** model as well as the video.

```
let video
let bodyPose

async function setup()
{
  ml5.setBackend("webgl")
  bodyPose = await ml5.bodyPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}
```

Notes

Nothing new, just missing a few steps we have covered already. Still won't see anything (except yourself) yet.



Sketch B4.3 get the results

We want the results of the `bodyPose`. We will console log the results into the `poses[]` array.

```
let video
let bodyPose
let poses = []

async function setup()
{
  ml5.setBackend("webgl")
  bodyPose = await ml5.bodyPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
  bodyPose.detectStart(video, gotPoses)
}

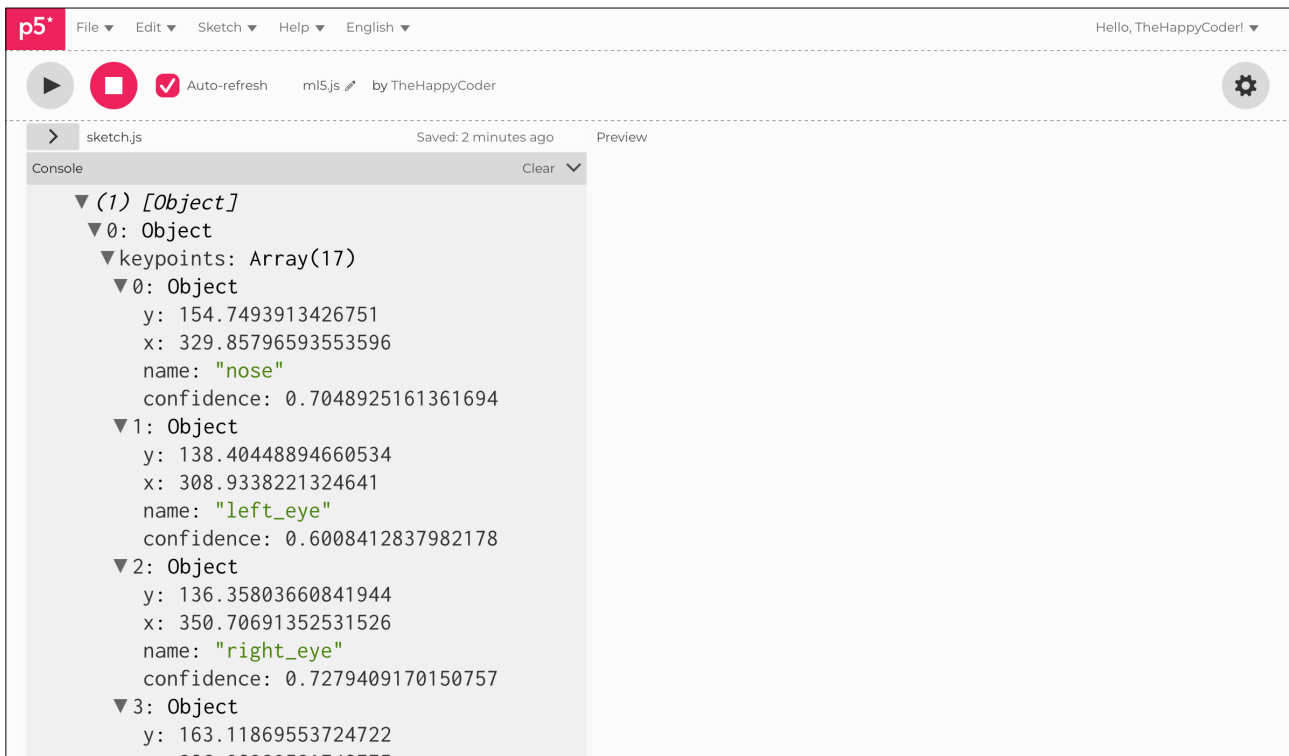
function draw()
{
  background(220)
  image(video, 0, 0, width, height)
}

function gotPoses(results)
{
  poses = results
  console.log(poses)
}
```

Notes

You can see the results in the console (just stop the code running and open up the arrays). They are called **keypoints**, which is important. They contain an (**x**, **y**) co-ordinates, a description (**name:**), and a **confidence** score. We can use these values to draw them on the canvas.

Figure B4.3





Sketch B4.4 keypoints

Let's draw these **keypoints** on the canvas. We have all the results in the **poses[]** array. We are going to loop through them and pull out the co-ordinates of all the **keypoints**.

! Remove the console log

```
let video
let bodyPose
let poses = []

async function setup()
{
  ml5.setBackend("webgl")
  bodyPose = await ml5.bodyPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
  bodyPose.detectStart(video, gotPoses)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  for (let i = 0; i < poses.length; i++)
  {
    let pose = poses[i]
    for (let j = 0; j < pose.keypoints.length; j++)
    {
      let keypoint = pose.keypoints[j]
      fill(255, 0, 0)
      circle(keypoint.x, keypoint.y, 10)
    }
  }
}

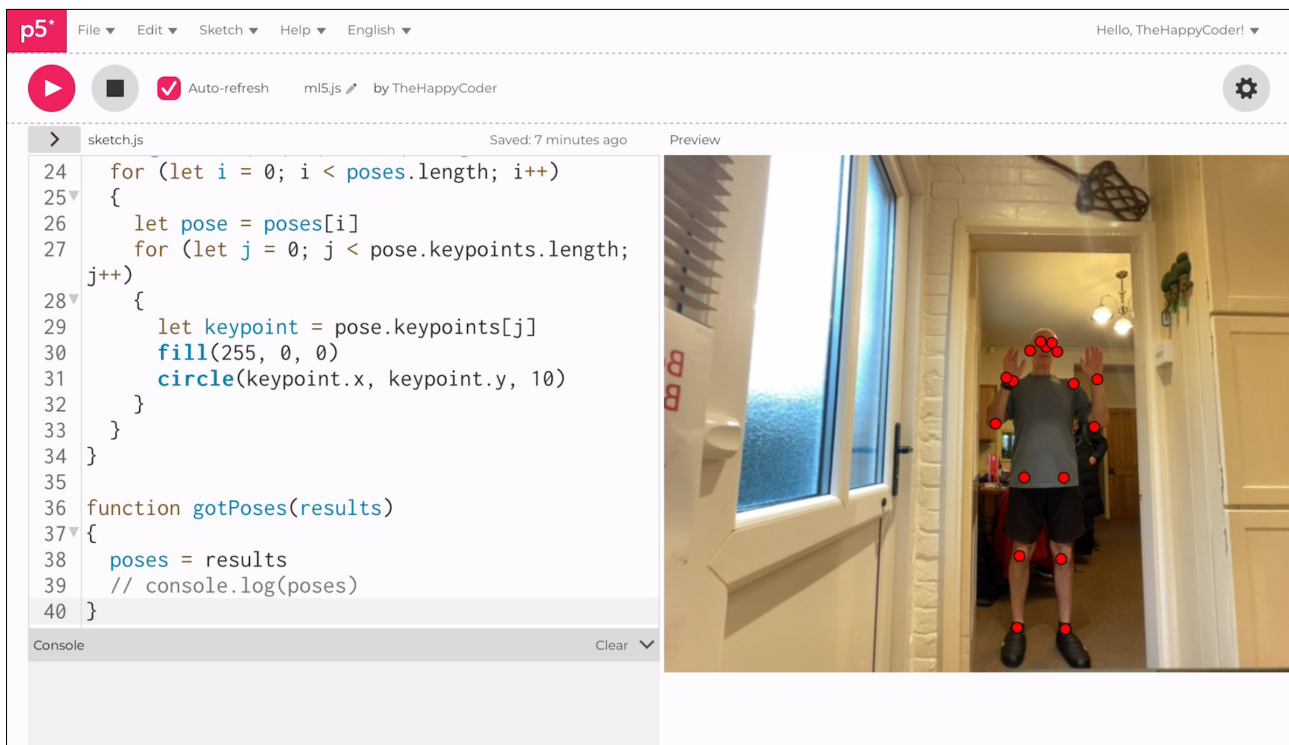
function gotPoses(results)
```

```
{  
  poses = results  
  // console.log(poses)  
}
```

Notes

In the console log, you can see the number of **keypoints**.

Figure B4.4





Sketch B4.5 confidence

We draw all the points, whether you can see the parts of the body or not. We can add a conditional statement to only draw the point if the confidence score is above **0.1**.

```
let video
let bodyPose
let poses = []

async function setup()
{
  ml5.setBackend("webgl")
  bodyPose = await ml5.bodyPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
  bodyPose.detectStart(video, gotPoses)
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  for (let i = 0; i < poses.length; i++)
  {
    let pose = poses[i]
    for (let j = 0; j < pose.keypoints.length; j++)
    {
      let keypoint = pose.keypoints[j]
      if (keypoint.confidence > 0.1)
      {
        fill(255, 0, 0)
        circle(keypoint.x, keypoint.y, 10)
      }
    }
  }
}
```



```
function gotPoses(results)
{
  poses = results
  // console.log(poses)
}
```

Notes

Now you should only get ones that are relevant to what you can see.

Challenge

Change the confidence score.

Code Explanation

```
if (keypoint.confidence > 0.1)
```

Another attribute found in the array, here we check to see if the key point confidence is above 0.1 for each keypoint



Sketch B4.6 skeleton

Let's draw the skeleton by drawing a line between points. Notice that we aren't just joining all the points, only the relevant ones, using the `getSkeleton()` function as part of the `bodyPose` model. There is a lot of code to get your head around and would take a lot of explaining. Just understand that every point has a corresponding point, and the inner nested loop (containing the `j` variable) goes through all of them. I suggest console logging to see what is happening.

```
let video
let bodyPose
let poses = []
let connections

async function setup()
{
  ml5.setBackend("webgl")
  bodyPose = await ml5.bodyPose({flipped: true})
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
  bodyPose.detectStart(video, gotPoses)
  connections = bodyPose.getSkeleton()
}

function draw()
{
  background(220)
  image(video, 0, 0, width, height)
  for (let i = 0; i < poses.length; i++)
  {
    let pose = poses[i]
    for (let j = 0; j < pose.keypoints.length; j++)
    {
      let keypoint = pose.keypoints[j]
      // if (keypoint.confidence > 0.1)
      // {
      //   fill(255, 0, 0)
      //   circle(keypoint.x, keypoint.y, 10)
```

```

    // }
  }
}

for (let i = 0; i < poses.length; i++)
{
  let pose = poses[i]
  for (let j = 0; j < connections.length; j++)
  {
    let pointAIndex = connections[j][0]
    let pointBIndex = connections[j][1]
    let pointA = pose.keypoints[pointAIndex]
    let pointB = pose.keypoints[pointBIndex]
    if (pointA.confidence > 0.1 && pointB.confidence > 0.1)
    {
      stroke(255)
      strokeWeight(3)
      line(pointA.x, pointA.y, pointB.x, pointB.y)
    }
  }
}

}

function gotPoses(results)
{
  poses = results
}

```

Notes

You should have the white lines joining most of the red circles. There are a lot of longer-named variables, which makes it difficult to read, but take your time to try to understand; otherwise, just know how to use it.

Challenge

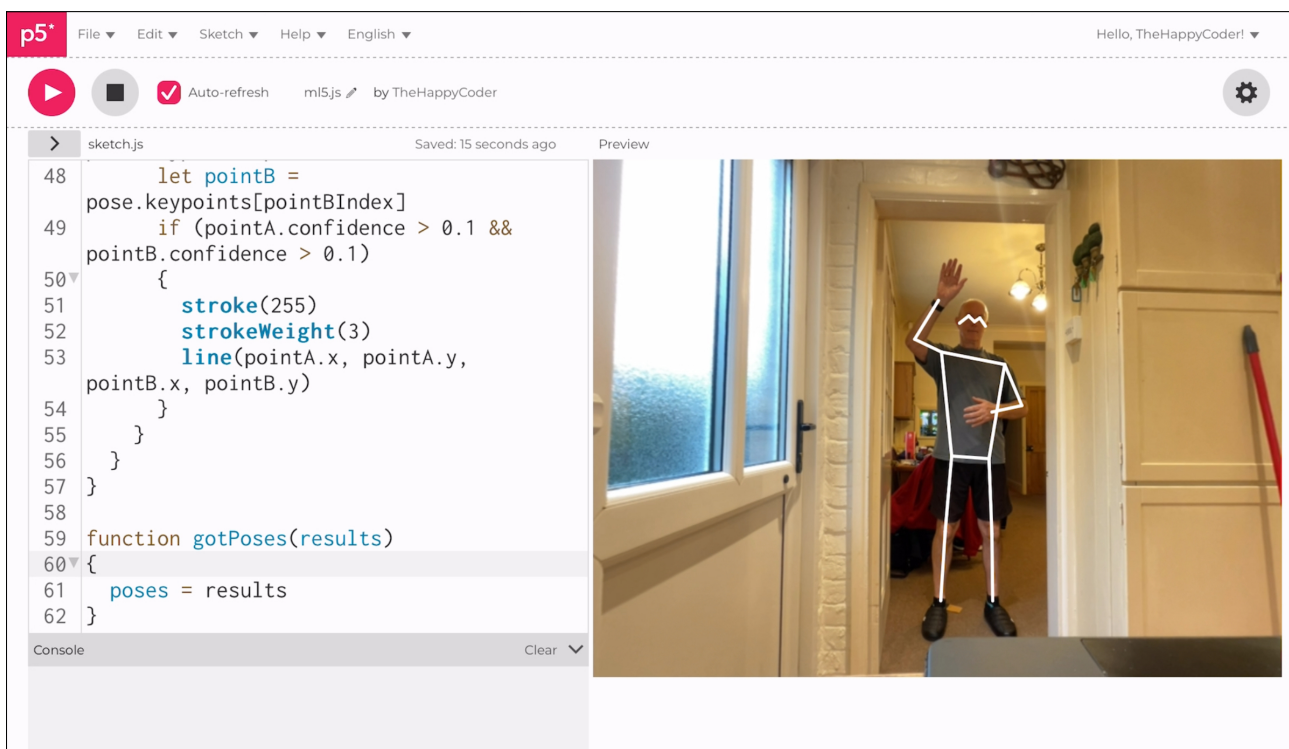
As I said earlier, to really start to understand the logic behind the nested loop, I recommend using the console log to explore.

Code Explanation

```
connections = bodyPose.getSkeleton()
```

The connections hold the skeleton connection points, this is so that they are joined together in a logical way rather than all of them joining each other.

Figure B4.6



Algorithmic Intelligence Module B Unit #5 pretrained speech



Module B Unit #5: soundClassifier with ml5.js

In this unit, we will demonstrate pre-trained speech classification using the **SpeechCommands18w** pretrained model. We will look at the basic model and see how good it is, and then use some of those commands to move an object around. There are **18** words, and they are as follows:

Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine, Up, Down, Left, Right, Go, Stop, Yes, No

You will need to allow access to the **microphone** on your computer (assuming there is one). The results can be a bit sketchy, but it is a good demonstration of what can be achieved. One real-world application is a wake word similar to a smart speaker.



The index.html file

Adding the code line for ml5.js.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch B5.1 the preload starting sketch

We will preload the pretrained model. We want it to have a confidence score of **0.7** or higher. We load **SpeechCommands18w** from ml5.js.

```
let classifier

async function setup()
{
  ml5.setBackend("webgl")
  let options = {probabilityThreshold: 0.7}
  classifier = await ml5.soundClassifier("SpeechCommands18w")
  createCanvas(400, 400)
}

function draw()
{
  background(220)
}
```

Notes

Nothing is meant to happen yet; all we have done is download the model.

Code Explanation

```
let options = {probabilityThreshold: 0.7}
```

The probabilityThreshold is a built in function

```
classifier = await  
ml5.soundClassifier("SpeechCommands18w")
```

This is a speech classifier that uses the
SpeechCommands18W pretrained words



Sketch B5.2 classifying

We want to start the pretrained model and classify the sounds/words using the `classifyStart()` function with a callback.

```
let classifier

async function setup()
{
  ml5.setBackend("webgl")
  let options = {probabilityThreshold: 0.7}
  classifier = await ml5.soundClassifier("SpeechCommands18w")
  createCanvas(400, 400)
  classifier.classifyStart(gotResults)
}

function draw()
{
  background(220)
}
```

Notes

! You will get an error message if you run it now. We will want a `gotResults()` callback function.

Code Explanation

```
classifier.classifyStart(gotResults)
```

Starts the classification of the sounds



Sketch B5.3 the got the results

Once we have the results, we need the callback function `gotResults()` and predict the label which will appear as the top, or highest, confidence score at `index[0]`. We will add the console log temporarily to see the results.

```
let classifier

let predictedWord = ""

async function setup()
{
  ml5.setBackend("webgl")
  let options = {probabilityThreshold: 0.7}
  classifier = await ml5.soundClassifier("SpeechCommands18w")
  createCanvas(400, 400)
  classifier.classifyStart(gotResults)
}

function draw()
{
  background(220)
}

function gotResults(results)
{
  predictedWord = results[0].label
  console.log(predictedWord)
}
```

Notes

It will pick up every sound and try to classify it. The output also depends on what data it is trained on, whose voices, and what accents (American, British, or a mixture). Another factor may be the sensitivity of your built-in microphone.

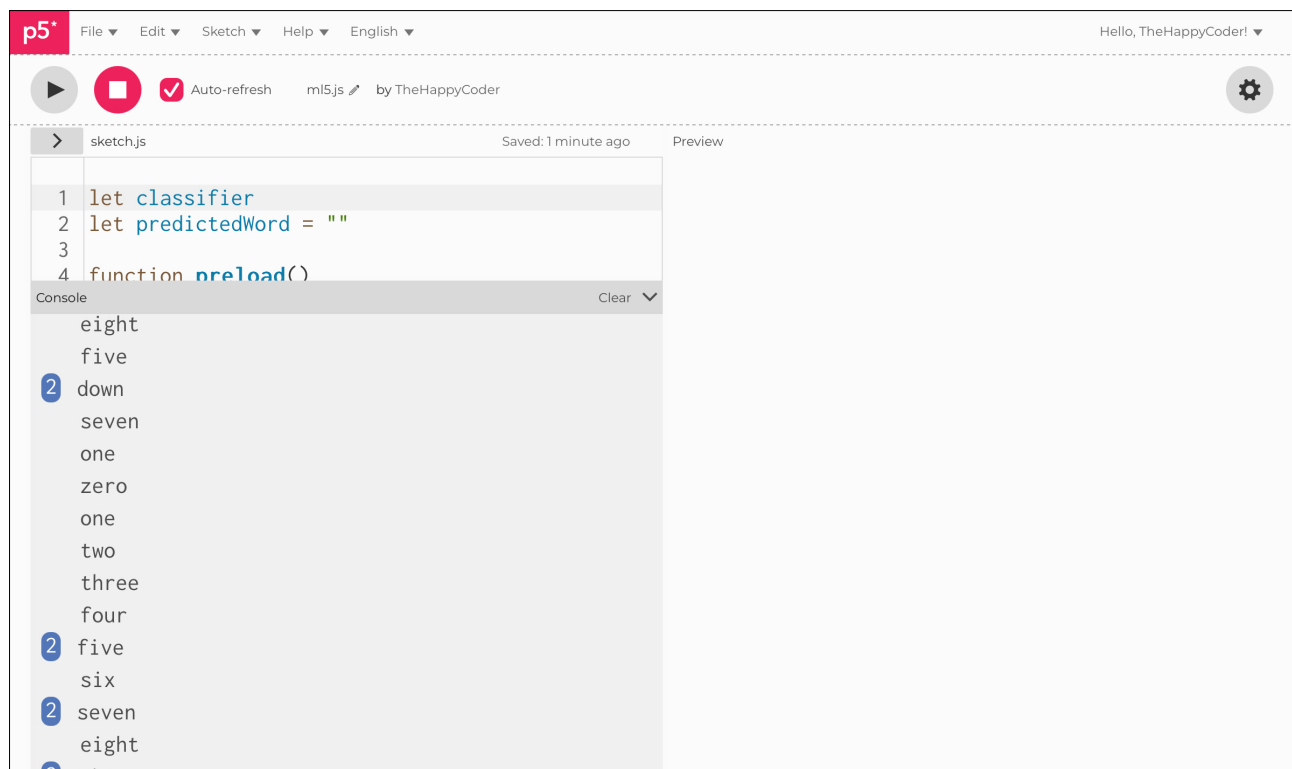
Challenges

1. Try different accents or shouting (quietly).
2. Turn off probability threshold

Code Explanation

<code>let predictedWord = ""</code>	Declare and initialise an empty string variable
<code>predictedWord = results[0].label</code>	Finds the first label with the highest probability

Figure B5.3





Sketch B5.5 a moving ball

Removing the text results in `draw()`. We will move the ball around the canvas with our voice. Add the `edges()` function so that if it goes off the edge, you don't lose it entirely. We will use the commands **UP**, **DOWN**, **LEFT**, **RIGHT**, and **STOP**.

```
let classifier
let predictedWord = ""

let x = 200
let y = 200

async function setup()
{
  ml5.setBackend("webgl")
  let options = {probabilityThreshold: 0.7}
  classifier = await ml5.soundClassifier("SpeechCommands18w")
  createCanvas(400, 400)
  classifier.classifyStart(gotResults)
}

function draw()
{
  background(220)

  if (predictedWord == "stop")
  {
    circle(x, y, 50)
    x = x
    y = y
  }
  else if (predictedWord == "up")
  {
    circle(x, y, 50)
    y -= 5
  }
  else if (predictedWord == "down")
  {
    circle(x, y, 50)
    y += 5
  }
}
```

```
else if (predictedWord == "left")
{
    circle(x, y, 50)
    x -= 5
}
else if (predictedWord == "right")
{
    circle(x, y, 50)
    x += 5
}
else
{
    circle(x, y, 50)
}
edges()
}
```

```
function gotResults(results)
{
    predictedWord = results[0].label
    console.log(predictedWord)
}
```

```
function edges()
{
    if (x > width)
    {
        x = 0
    }
    if (x < 0)
    {
        x = width
    }
    if (y > height)
    {
        y = 0
    }
    if (y < 0)
```

```
{  
  y = height  
}  
}
```

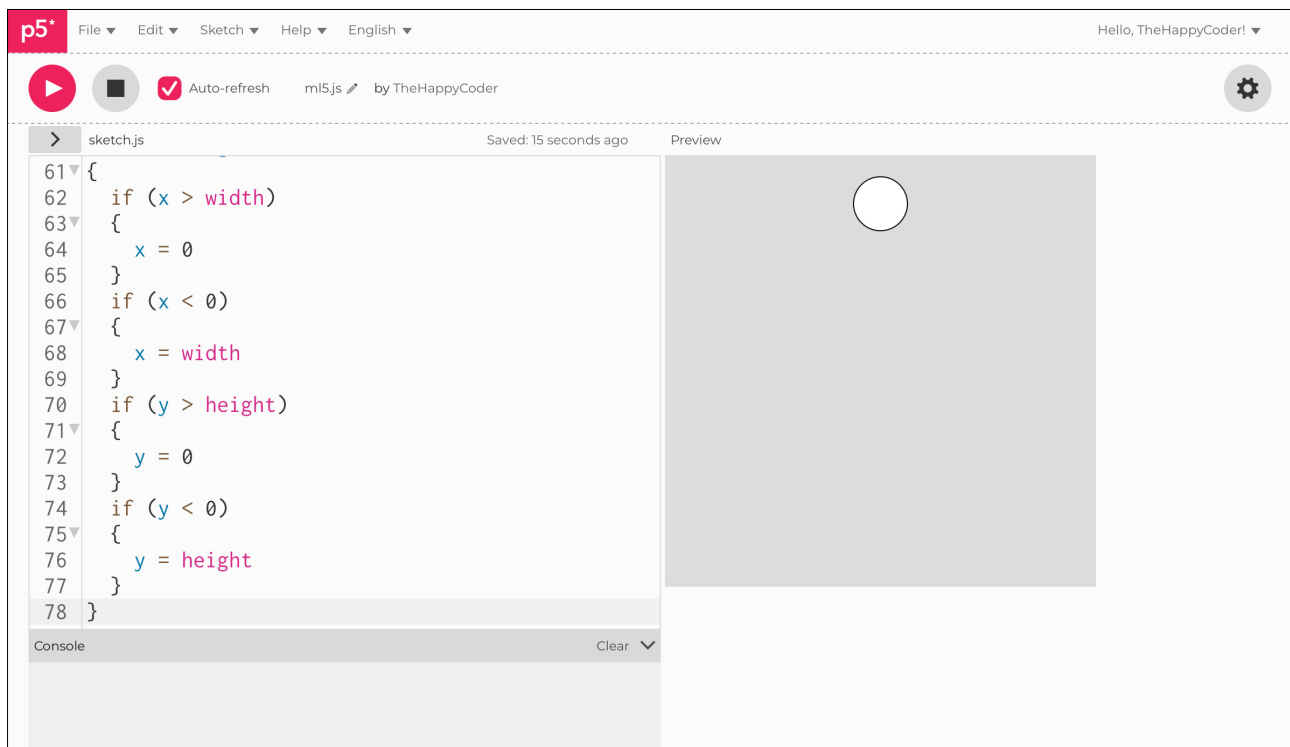
Notes

You should be able to move the circle around with your voice. There are quite a few lines of code, but it does not require much typing. The `edge()` function is standard in most games so that the object stays somewhere on the canvas. It is pretty self-evident.

Challenge

You could change the `edges()` function so that it stops at the edge rather than reappearing on the opposite edge.

Figure B5.5





Sketch B5.6 a splash of colour and text

Adding a splash of colour and the text returns on the canvas.

```
let classifier
let predictedWord = ""
let x = 200
let y = 200

async function setup()
{
  ml5.setBackend("webgl")
  let options = {probabilityThreshold: 0.7}
  classifier = await ml5.soundClassifier("SpeechCommands18w")
  createCanvas(400, 400)
  classifier.classifyStart(gotResults)
}

function draw()
{
  background(220, 0, 200)
  if (predictedWord !== "")
  {
    fill(0)
    textAlign(CENTER, CENTER)
    textSize(64)
    text(predictedWord, width/2, height/2)
  }
  fill(200, 200, 10)
  noStroke()
  if (predictedWord == "stop")
  {
    circle(x, y, 50)
    x = x
    y = y
  }
  else if (predictedWord == "up")
  {
    circle(x, y, 50)
```

```

    y -= 5
  }
  else if (predictedWord == "down")
  {
    circle(x, y, 50)
    y += 5
  }
  else if (predictedWord == "left")
  {
    circle(x, y, 50)
    x -= 5
  }
  else if (predictedWord == "right")
  {
    circle(x, y, 50)
    x += 5
  }
  else
  {
    circle(x, y, 50)
  }
  edges()
}

function gotResults(results)
{
  predictedWord = results[0].label
  console.log(predictedWord)
}

function edges()
{
  if (x > width)
  {
    x = 0
  }
  if (x < 0)
  {

```



```
    x = width
}
if (y > height)
{
    y = 0
}
if (y < 0)
{
    y = height
}
}
```

Notes

The accuracy will be the same, which is not great in my case; you may have better results. The accuracy will be affected by background noise, the quality of your microphone, and possibly your accent, amongst other things. Have a bit of fun with it.

Challenge

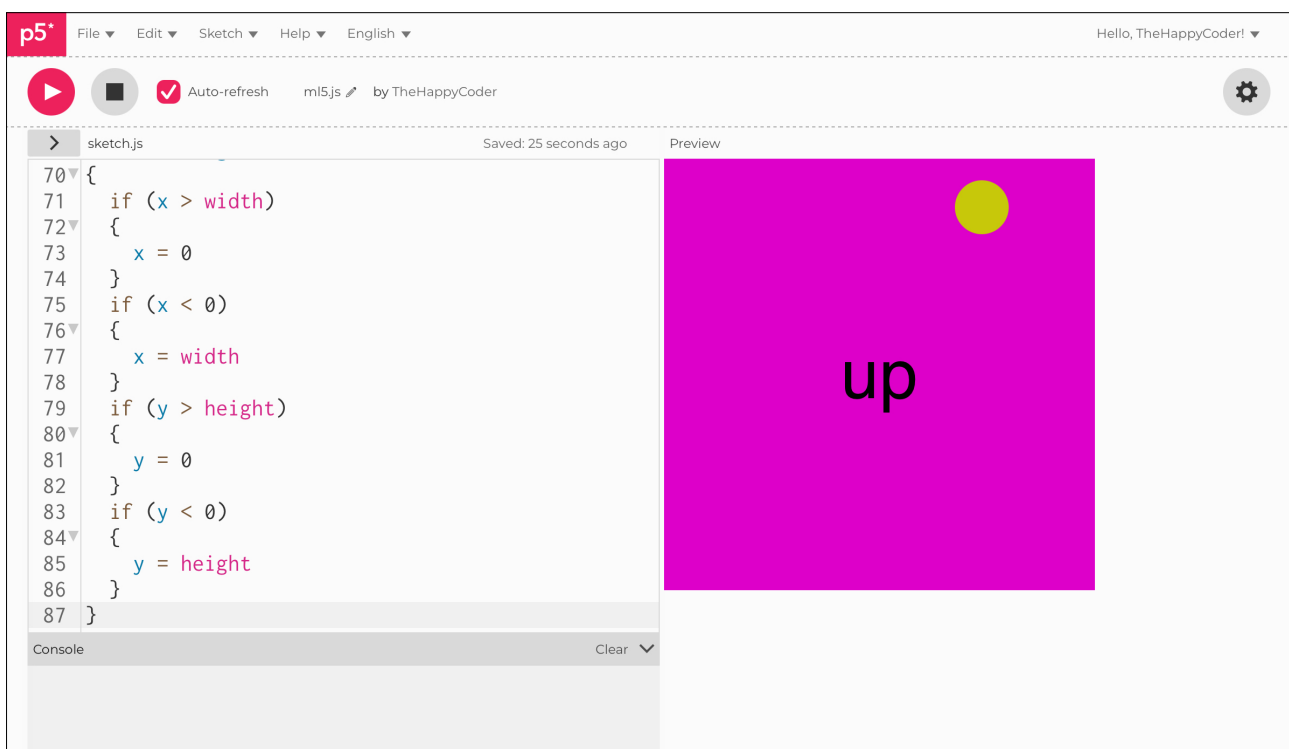
1. Try changing the `probabilityThreshold` value.
2. What could you do with the numbers?
3. Could you filter out the words you don't want?

Code Explanation

```
if (predictedWord !== "")
```

If there is NOT an empty string, in other words there IS a word it recognises

Figure B5.6



Algorithmic Intelligence Module B Unit #6 body segments



Module B Unit #6: body segmentation with ml5.js

The ml5.js BodySegmentation provides two models, **SelfieSegmentation** and **BodyPix**. The **SelfieSegmentation** model focuses on segmenting the human subject from the background. The **BodyPix** model is primarily used for detailed body part segmentation (e.g., distinguishing between different limbs) in images and videos. Although **BodyPix** can also perform person/background segmentation, it is more computationally intensive.

The ml5.js **BodySegmentation** is built on top of the **TensorFlow.js BodyPix model** and the **MediaPipe Selfie Segmentation model**.

It provides the following functionalities:

A) Real-time person/background segmentation:

The **SelfieSegmentation** model can segment people from the background in real time, and is designed to be lightweight. The **BodyPix** model can also be used for this purpose, but is more computationally intensive.

B) Real-time body part segmentation:

The **BodyPix** model can segment 24 body parts in real time.

Here is the reminder to have the latest **ml5.js** and **p5.js** scripts.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
    </main>
  <script src="sketch.js"></script>
</body></html>
```

The parts list with reference number:

LEFT_FACE: 0
RIGHT_FACE: 1
LEFT_UPPER_ARM_FRONT: 2
LEFT_UPPER_ARM_BACK: 3
RIGHT_UPPER_ARM_FRONT: 4
RIGHT_UPPER_ARM_BACK: 5
LEFT_LOWER_ARM_FRONT: 6
LEFT_LOWER_ARM_BACK: 7
RIGHT_LOWER_ARM_FRONT: 8
RIGHT_LOWER_ARM_BACK: 9
LEFT_HAND: 10
RIGHT_HAND: 11
TORSO_FRONT: 12
TORSO_BACK: 13
LEFT_UPPER_LEG_FRONT: 14
LEFT_UPPER_LEG_BACK: 15
RIGHT_UPPER_LEG_FRONT: 16
RIGHT_UPPER_LEG_BACK: 17
LEFT_LOWER_LEG_FRONT: 18
LEFT_LOWER_LEG_BACK: 19
RIGHT_LOWER_LEG_FRONT: 20
RIGHT_LOWER_LEG_BACK: 21
LEFT_FOOT: 22
RIGHT_FOOT: 23

We will be using BodyPix for this unit.



Sketch B6.1 our basic sketch

Here we have created our video and flipped it as per usual.

```
let video

function setup()
{
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

You should have a video of you in the canvas area.



Sketch B6.2 preload

Loading the pre-trained **bodySegmentation** model in ml5.js.

```
let video
let body

async function setup()
{
  body = await ml5.bodySegmentation('BodyPix')
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

Only the video of you.

Code Explanation

```
body = ml5.bodySegmentation('BodyPix')
```

Loading the bodySegmentation pretrained model, BodyPix version



Sketch B6.3 masking the body parts

We want the options to mask the body parts.

```
let video
let body

let options = { maskType: 'parts'}

async function setup()
{
  body = await ml5.bodySegmentation('BodyPix', options)
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

Still just the video of you.

Code Explanation

```
let options = { maskType: 'parts'}
```

We want it to identify and mask over the main body parts



Sketch B6.4 start detecting your body parts

Using the function `detectStart()` and having the callback `gotResults()`.

```
let video
let body
let options = { maskType: 'parts'}

async function setup()
{
  body = await ml5.bodySegmentation('BodyPix', options)
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
  body.detectStart(video, gotResults)
}

function draw()
{
  background(220)
  image(video, 0, 0)
}
```

Notes

! As you might guess, we get an error message because we haven't created the `gotResults()` function yet.



Sketch B6.5 gotResults()

Now we add the `gotResults()` function. This includes a variable called `segmentation` (made-up name) to hold the results. We need this when we draw the segmented version of you.

```
let video
let body
let options = { maskType: 'parts' }
let segmentation

async function setup()
{
  body = await ml5.bodySegmentation('BodyPix', options)
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
  body.detectStart(video, gotResults)
}

function draw()
{
  background(220)
  image(video, 0, 0)
}

function gotResults(results)
{
  segmentation = results
}
```

Notes

Next, we want to draw the segmented version of you.



Sketch B6.6 detecting your segments

If we have detected your segments, then it will draw them; otherwise, nothing will happen. Also, include webgl backend and flip the segmented image.

```
let video
let body
let options = { maskType: 'parts', flipped: true}
let segmentation

async function setup()
{
  ml5.setBackend('webgl')
  body = await ml5.bodySegmentation('BodyPix', options)
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
  body.detectStart(video, gotResults)
}

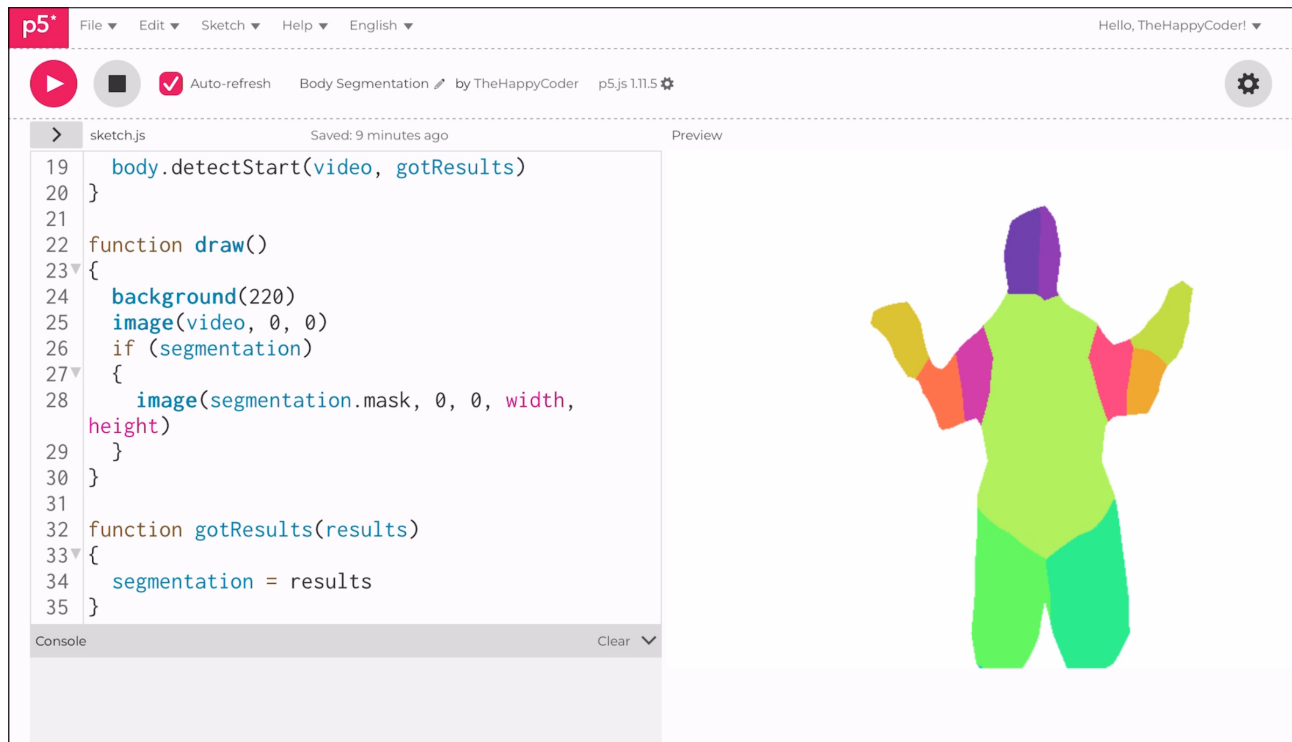
function draw()
{
  background(220)
  image(video, 0, 0)
  if (segmentation)
  {
    image(segmentation.mask, 0, 0, width, height)
  }
}

function gotResults(results)
{
  segmentation = results
}
```

Notes

You should get something like the figure below. It was far from perfect, but maybe a better camera or processor (use CPU or GPU for the backend) might yield better results. also on my Mac it always centred around the head, I clicked on the video camera icon and deselected that option.

Figure B6.6





Sketch B6.7 selecting a body part

We can highlight and play with our body parts; this is just an illustration or suggestion for how you might use it.

```
let video
let body
let options = { maskType: 'parts', flipped: true}
let segmentation

async function setup()
{
  ml5.setBackend('webgl')
  body = await ml5.bodySegmentation('BodyPix', options)
  createCanvas(640, 480)
  video = createCapture(VIDEO, {flipped: true})
  video.size(640, 480)
  video.hide()
  body.detectStart(video, gotResults)
}

function draw()
{
  background(220)
  image(video, 0, 0)
  if (segmentation)
  {
    let parts = body.getPartsId()
    let gridSize = 10
    for (let x = 0; x < video.width; x += gridSize)
    {
      for (let y = 0; y < video.height; y += gridSize)
      {
        if (segmentation.data[y * video.width + x] == parts.TORSO_FRONT)
        {
          fill(255, 0, 0)
          noStroke()
          circle(x, y, gridSize)
        }
      }
    }
  }
}
```

```

    }
  }
}

function gotResults(results)
{
  segmentation = results
}

```

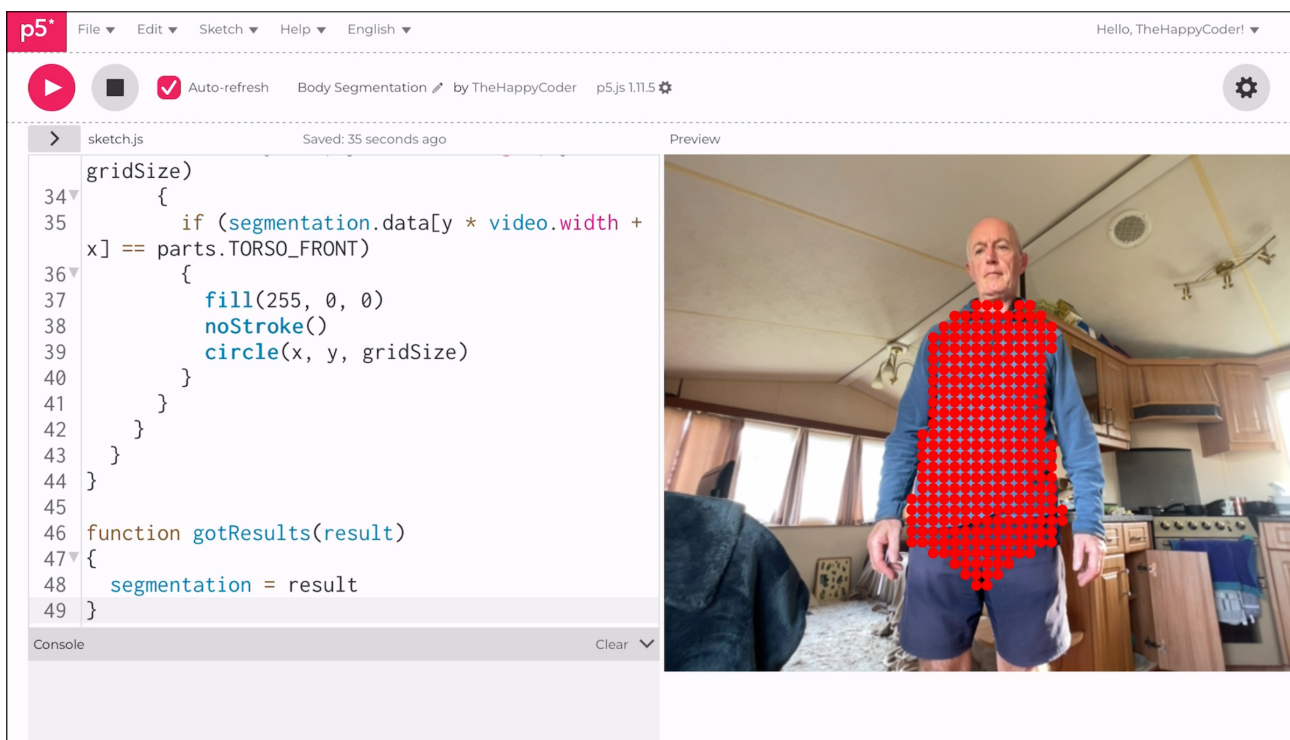
Notes

You will cover your torso with lots of red dots.

Challenge

Try another part of your body.

Figure B6.7



Algorithmic Intelligence Module B Unit #7 mobileNet Image Classification



Module B Unit #7: image classification

The pre-trained model we are going to use is called **mobileNet**. It is a very handy model that you can use online. It uses a dataset from **ImageNet**, which has nearly 15 million images. So it is quite comprehensive but not complete, as you may well find out when you try to look at objects.

This unit, we just upload images to classify.



Sketch B7.1 index.html

Make sure you have the `ml5.js` line of code in your `index.html` file.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch B7.2 callback

We have now access to another pre-trained model with `ml5.js` called `mobileNet`. We are going to check that everything is working by having a callback. The callback tells us when it is loaded in the console.

```
let mobilenet

function setup()
{
  createCanvas(400, 400)
  mobilenet = ml5.imageClassifier('MobileNet', modelReady)
}

function modelReady()
{
  console.log('model is ready')
}
```

Notes

We get the callback console log: `model is ready`. It may take a few seconds.



A banana

You can use any object you like, but choose something that doesn't have any other object in it. Just make sure it works OK from the start. I have selected a banana image from [Unsplash](#).

Figure 1: Source: 'Mockup Graphics' on Unsplash

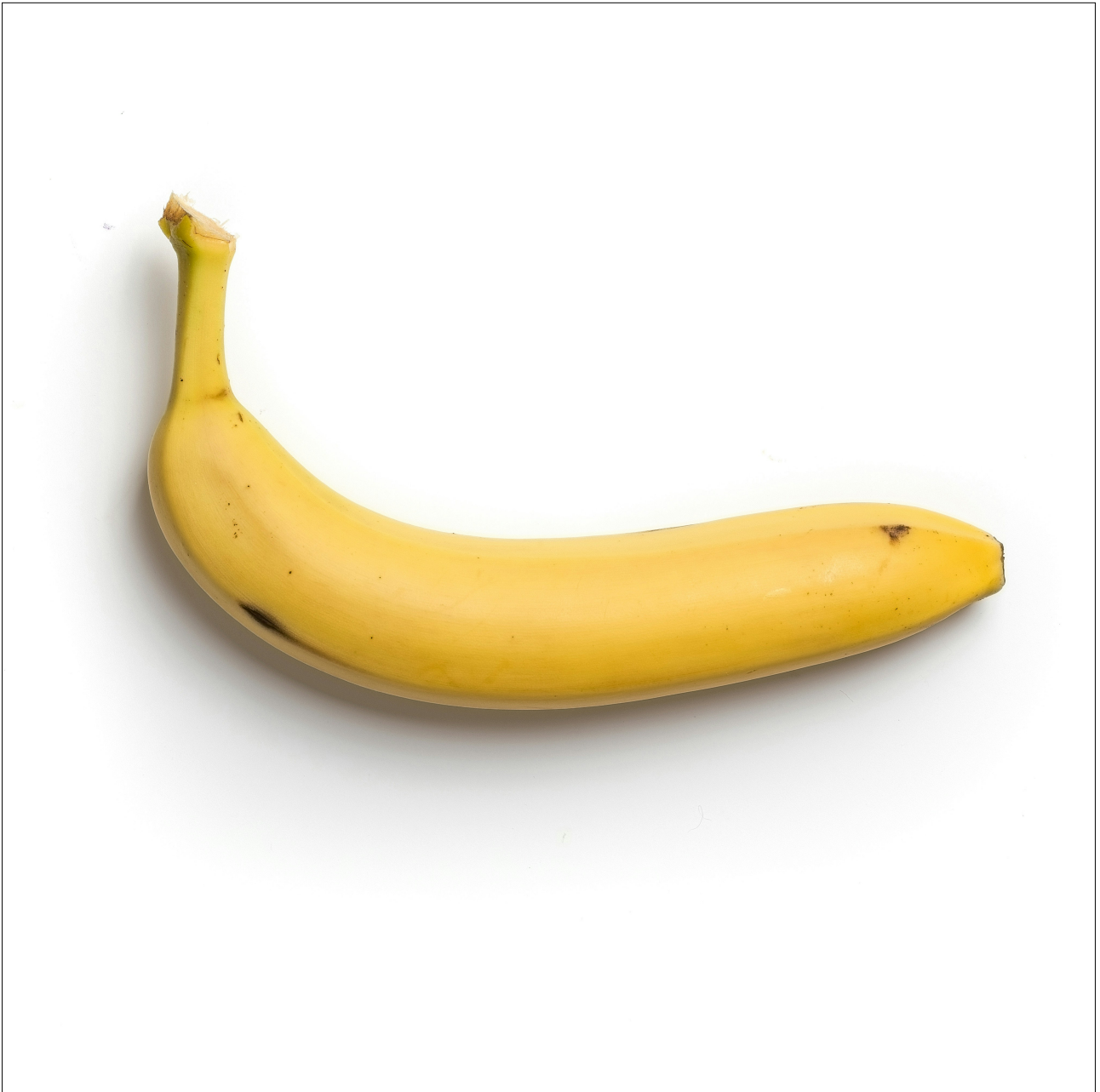


Figure 2: Upload it to the sketch

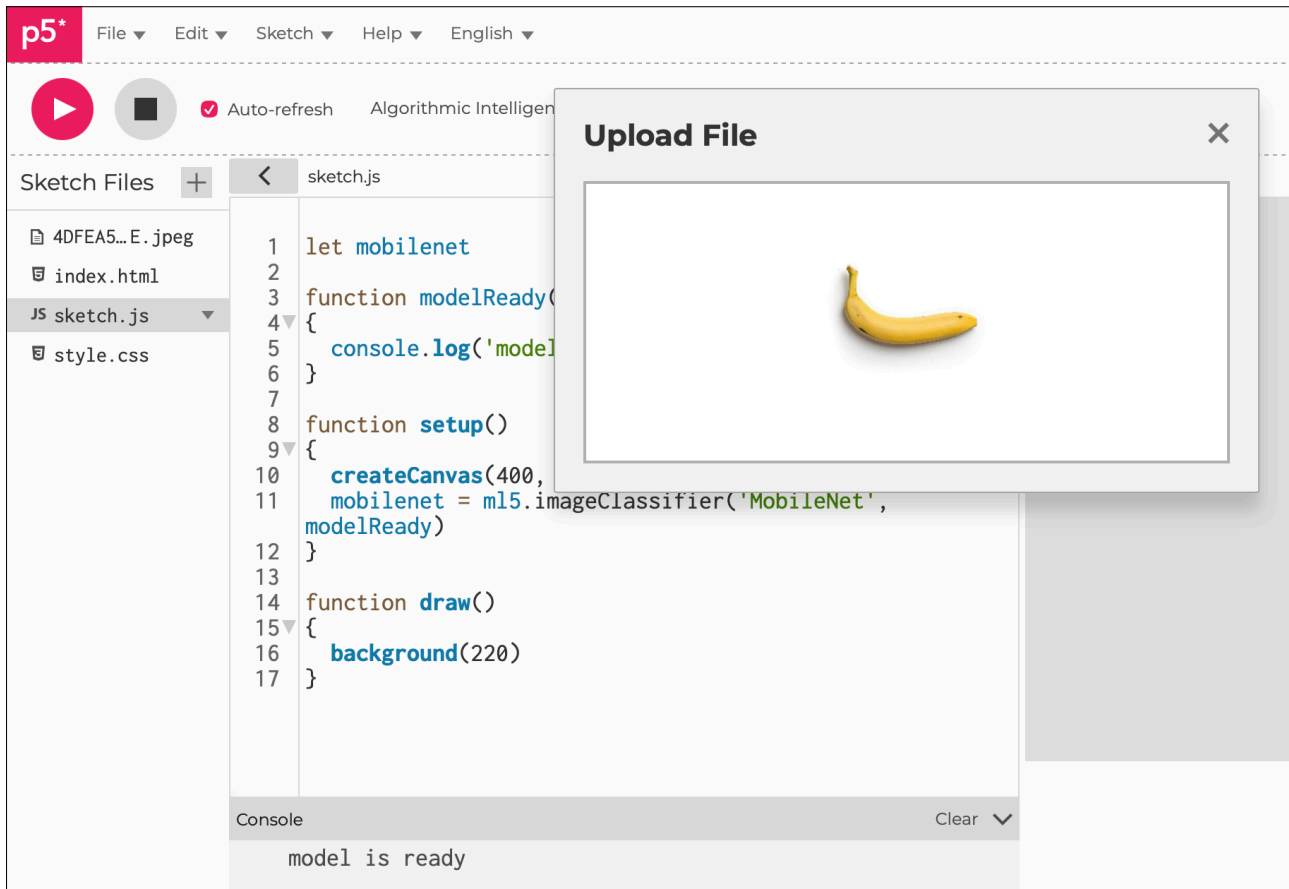
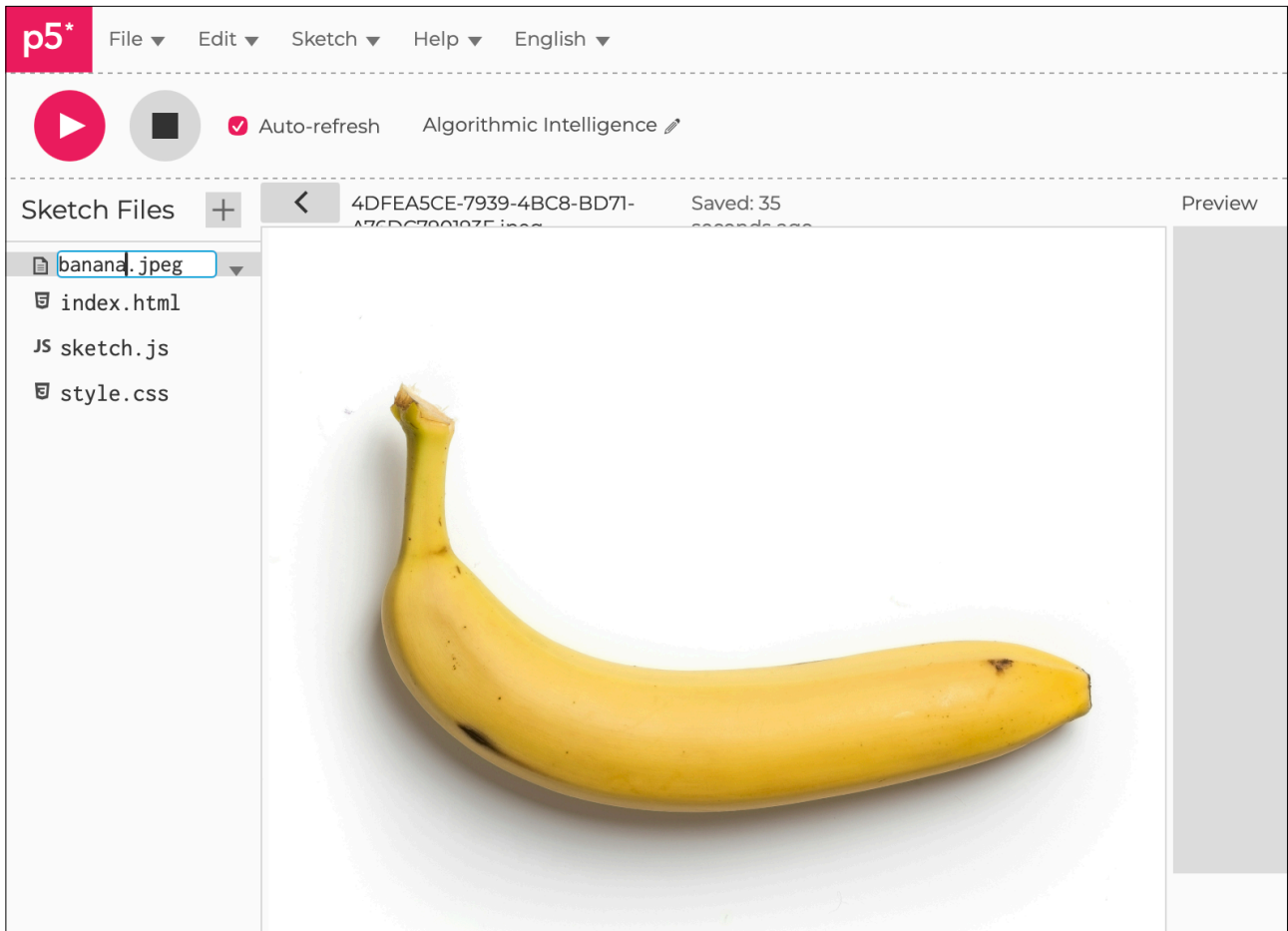


Figure 3: rename 'banana'





Sketch B7.3 the appearance of the banana

Now that you have successfully uploaded an image and renamed it, then we can display it on the canvas.

```
let mobilenet
let img

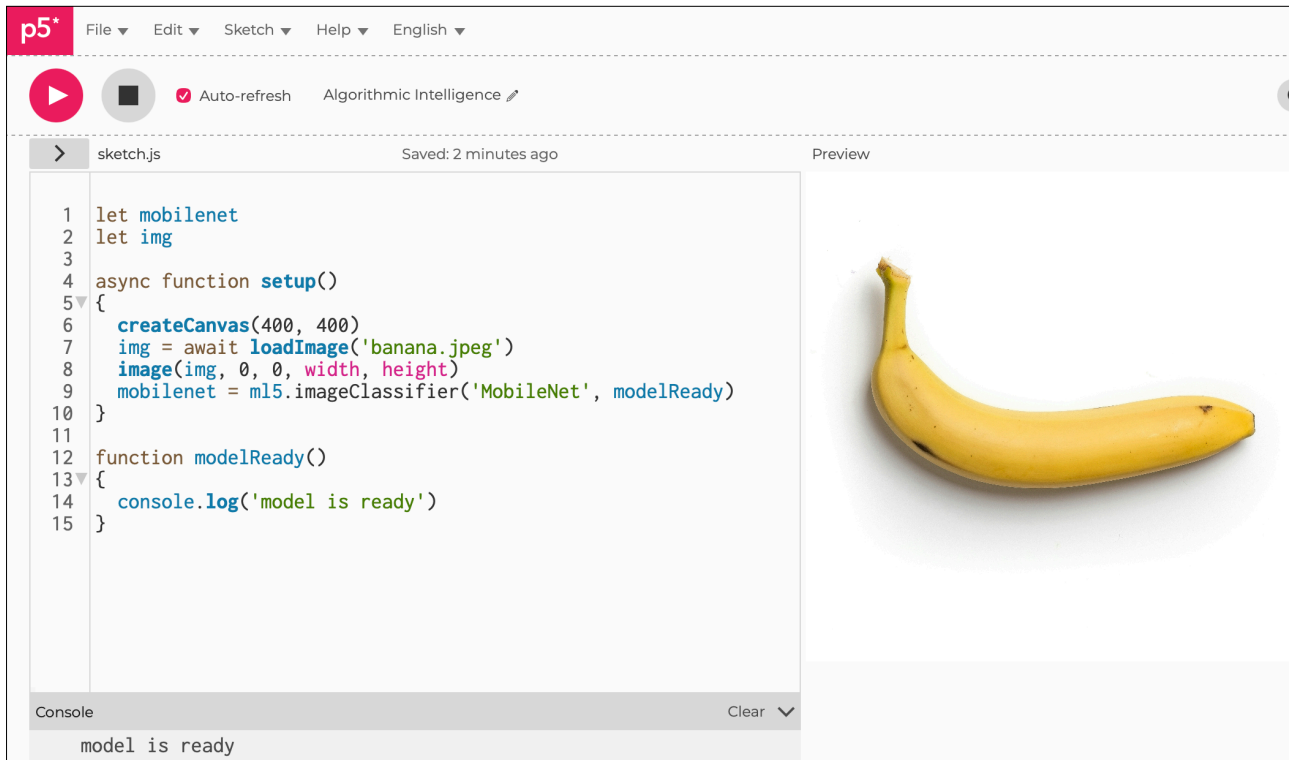
async function setup()
{
  createCanvas(400, 400)
  img = await loadImage('banana.jpeg')
  image(img, 0, 0, width, height)
  mobilenet = ml5.imageClassifier('MobileNet', modelReady)
}

function modelReady()
{
  console.log('model is ready')
}
```

Notes

We use **async** and **await** because we want the image to be downloaded first, before the model is loaded. The banana should magically appear on the canvas. I deliberately chose a square image so there's no squashing or stretching.

Figure B7.3





Sketch B7.4 the classification

We can make a few changes to the sketch to get the model to **classify** (predict) what the object is in the image. The first change is that we need the model to load before we start to classify. We get rid of the callback and change its name (to **gotResults**) so we can repurpose it for collecting the results of the classification.

```
let mobilenet
let img

async function setup()
{
  createCanvas(400, 400)
  img = await loadImage('banana.jpeg')
  image(img, 0, 0, width, height)
  mobilenet = await ml5.imageClassifier('MobileNet')
  mobilenet.classify(img, gotResults)
}

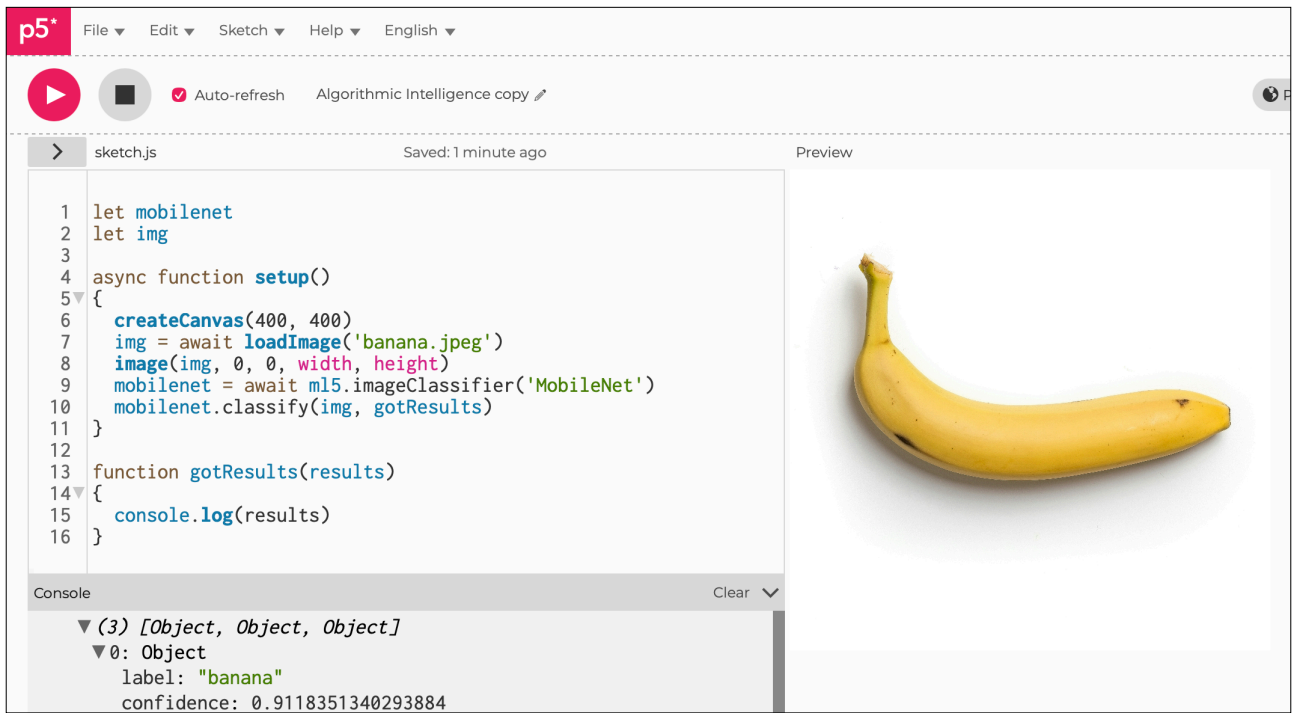
function gotResults(results)
{
  console.log(results)
}
```

Notes

You will notice that we get the top three guesses (predictions), and they are scored as a probability. Clearly, this is a banana with almost **100% (0.91)** score; the other two are interesting. A slug, I could just about get!

! If you get an error message, just check you have removed the redundant code.

Figure B7.4





Sketch B7.5 get the results

We want the top result, which is the **label** at **index [0]**, the first one. This is also true for probability (confidence). We create two variables to hold these values as strings, as we will want to display them on the canvas as a fixed value or name.

! Remove the console log.

```
let mobilenet
let img
let label = ''
let confidence = ''

async function setup()
{
  createCanvas(400, 400)
  img = await loadImage('banana.jpeg')
  image(img, 0, 0, width, height)
  mobilenet = await ml5.imageClassifier('MobileNet')
  mobilenet.classify(img, gotResults)
}

function gotResults(results)
{
  label = results[0].label
  confidence = results[0].confidence
}
```

Notes

Nothing new to see.



Sketch B7.6 display

! Move the image into the `draw()` function from `setup()`.

We multiply the confidence by `100` (to get the percentage) and take the floor value. We then print the text onto the canvas. We use the `draw()` function for this.

```
let mobilenet
let img
let label = ''
let confidence = ''

async function setup()
{
  createCanvas(400, 400)
  img = await loadImage('banana.jpeg')
  mobilenet = await ml5.imageClassifier('MobileNet')
  mobilenet.classify(img, gotResults)
}

function draw()
{
  image(img, 0, 0, width, height)
  textSize(20)
  text('label: ' + label, 100, 350)
  text('confidence: ' + floor(confidence * 100) + '%', 100, 380)
}

function gotResults(results)
{
  label = results[0].label
  confidence = results[0].confidence
}
```

Notes

A very confident banana

Algorithmic Intelligence Module B Unit #8 mobileNet Video Classification



Module B Unit #8: video classification

Using `mobileNet` as before we are going to live stream and try to classify what it can see, should be fun! There are a lot of similarities to classifying an image. Do remember that it has a limited knowledge of objects and will throw up some interesting alternatives (if any).



Sketch B8.1 index.html

Make sure you have the `ml5.js` line of code in your `index.html` file.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch B8.2 starting sketch

Getting the video to work, you will get a prompt asking permission to allow access to the webcam.

```
let mobilenet
let video

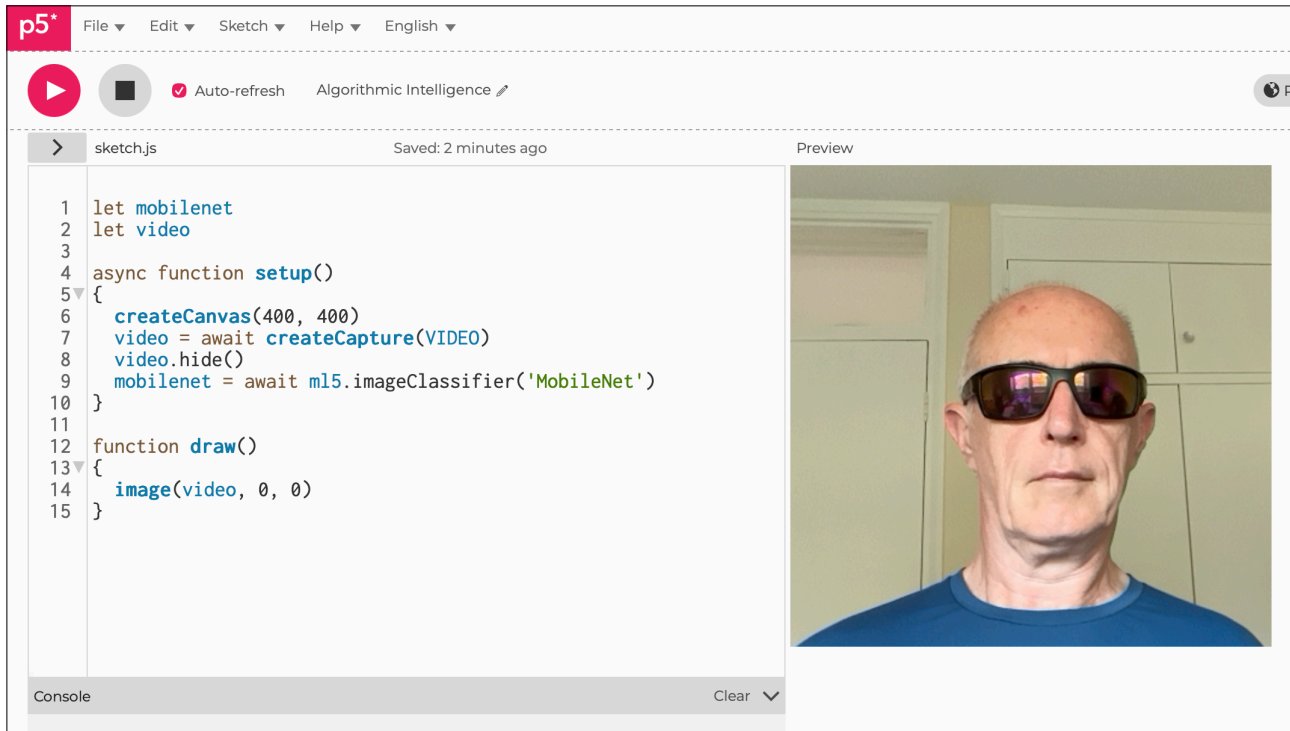
async function setup()
{
  createCanvas(400, 400)
  video = await createCapture(VIDEO)
  video.hide()
  mobilenet = await ml5.imageClassifier('MobileNet')
}

function draw()
{
  image(video, 0, 0)
}
```

Notes

You should get an image of yourself like below, but maybe without the sunglasses. The reason I am wearing sunglasses will be explained shortly.

Figure B8.2





Sketch B8.3 classifying

To classify a streaming video, we need to use a different function called `classifyStart()` rather than just `classify()`. We also add our old friend, the `gotResults()` function.

```
let mobilenet
let video

async function setup()
{
  createCanvas(400, 400)
  video = await createCapture(VIDE0)
  video.hide()
  mobilenet = await ml5.imageClassifier('MobileNet')
  mobilenet.classifyStart(video, gotResults)
}

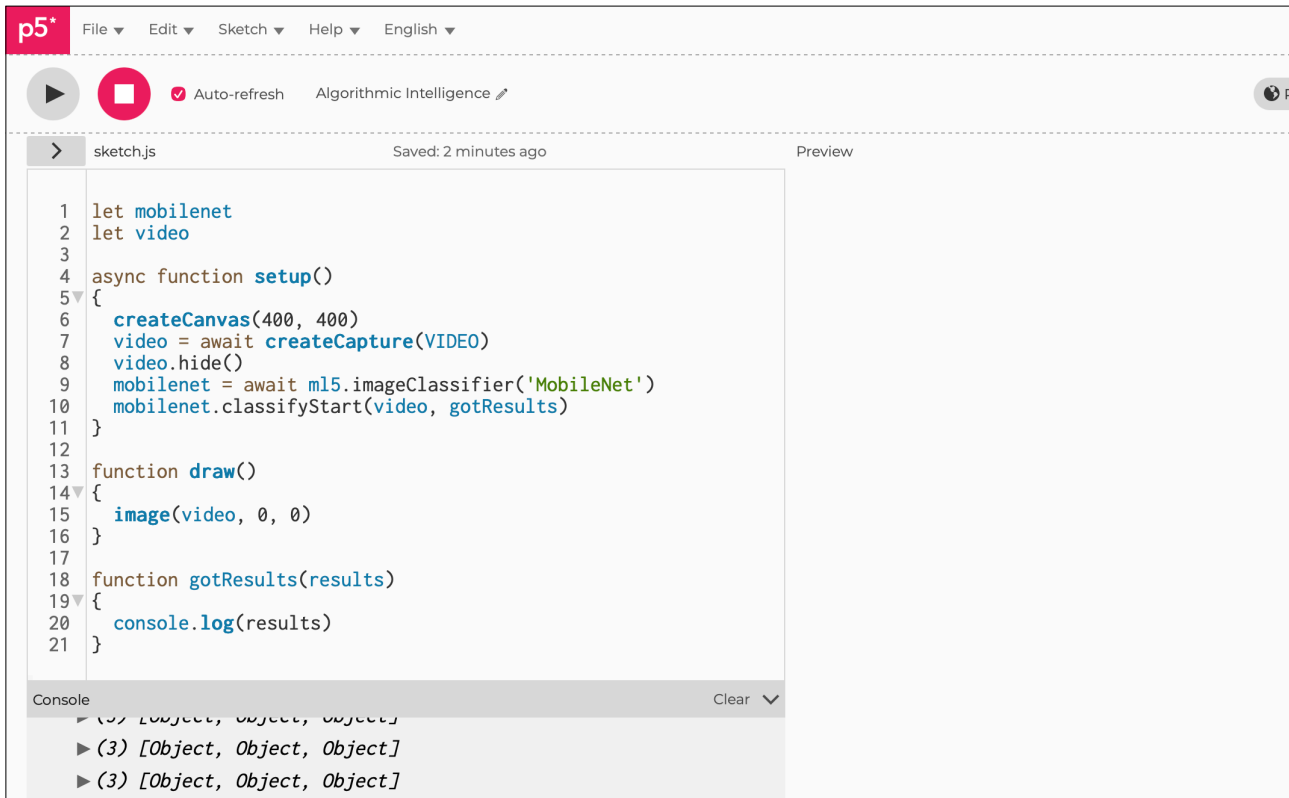
function draw()
{
  image(video, 0, 0)
}

function gotResults(results)
{
  console.log(results)
}
```

Notes

We get a continuous stream of array data, so press the stop/pause button and take a look inside one of these arrays. You will see three predictions with the usual label and confidence score. You might be very surprised by what it thinks it is looking at!

Figure B8.3





Sketch B8.4 display the results

We can follow a similar trend as we did with the image, but to make sure we can see the results, I will put the text over a separate background.

```
let mobilenet
let video
let label = ''
let confidence = ''

async function setup()
{
  createCanvas(400, 400)
  video = await createCapture(VIDEO)
  video.hide()
  mobilenet = await ml5.imageClassifier('MobileNet')
  mobilenet.classifyStart(video, gotResults)
}

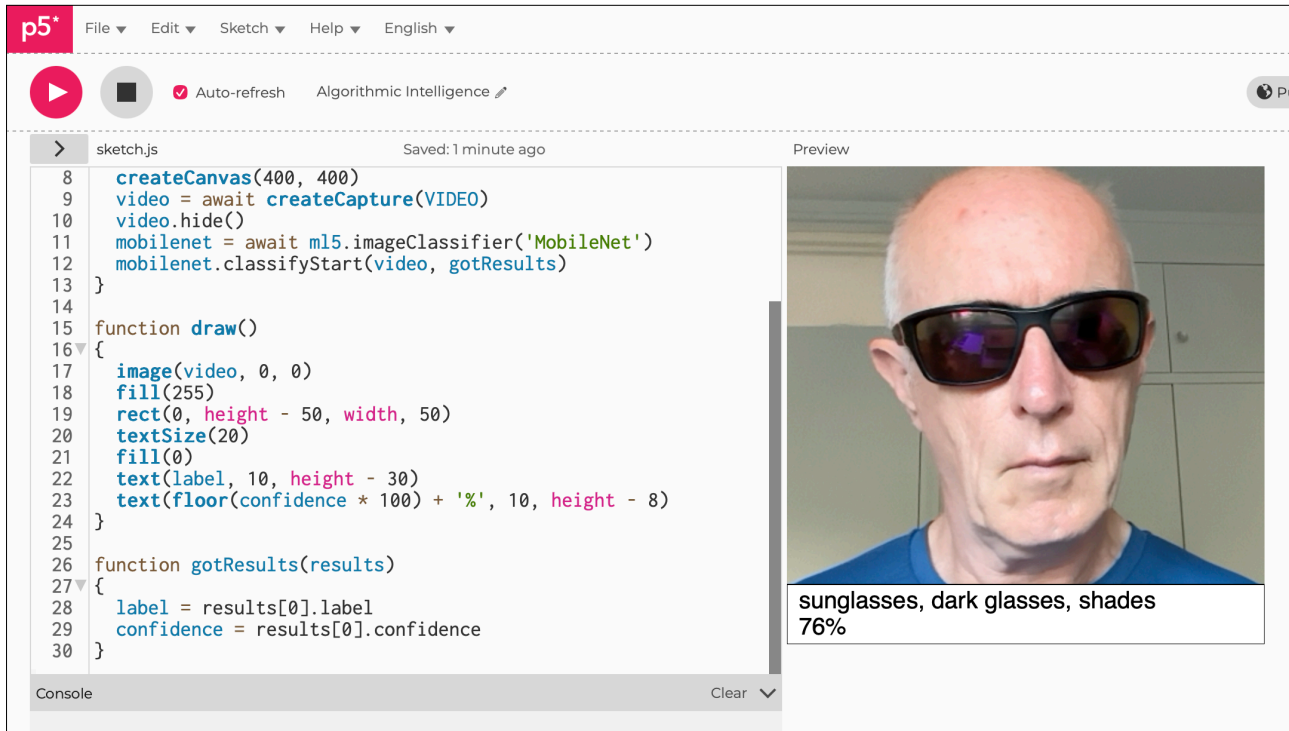
function draw()
{
  image(video, 0, 0)
  fill(255)
  rect(0, height - 50, width, 50)
  textSize(20)
  fill(0)
  text(label, 10, height - 30)
  text(floor(confidence * 100) + '%', 10, height - 8)
}

function gotResults(results)
{
  label = results[0].label
  confidence = results[0].confidence
}
```

Notes

This is why I wore the sunglasses; it gave a very correct answer, even if the confidence value varied from 30% to 80% constantly. Some of the other predictions were a lot less flattering, to say the least.

Figure B8.4





Sketch B8.5 a bit of a mug

By adding a button and changing the predicting function from `classifyStart()` to `classify()` and then putting it in another function called `snapshot()`, we can now position something and classify it just once.

```
let mobilenet
let video
let label = ''
let confidence = ''

let button

async function setup()
{
  createCanvas(400, 400)
  video = await createCapture(VIDEO)
  video.hide()
  mobilenet = await ml5.imageClassifier('MobileNet')
  // mobilenet.classifyStart(video, gotResults)
  button = createButton('take a snapshot')
  button.mousePressed(snapshot)
}

function snapshot()
{
  mobilenet.classify(video, gotResults)
}

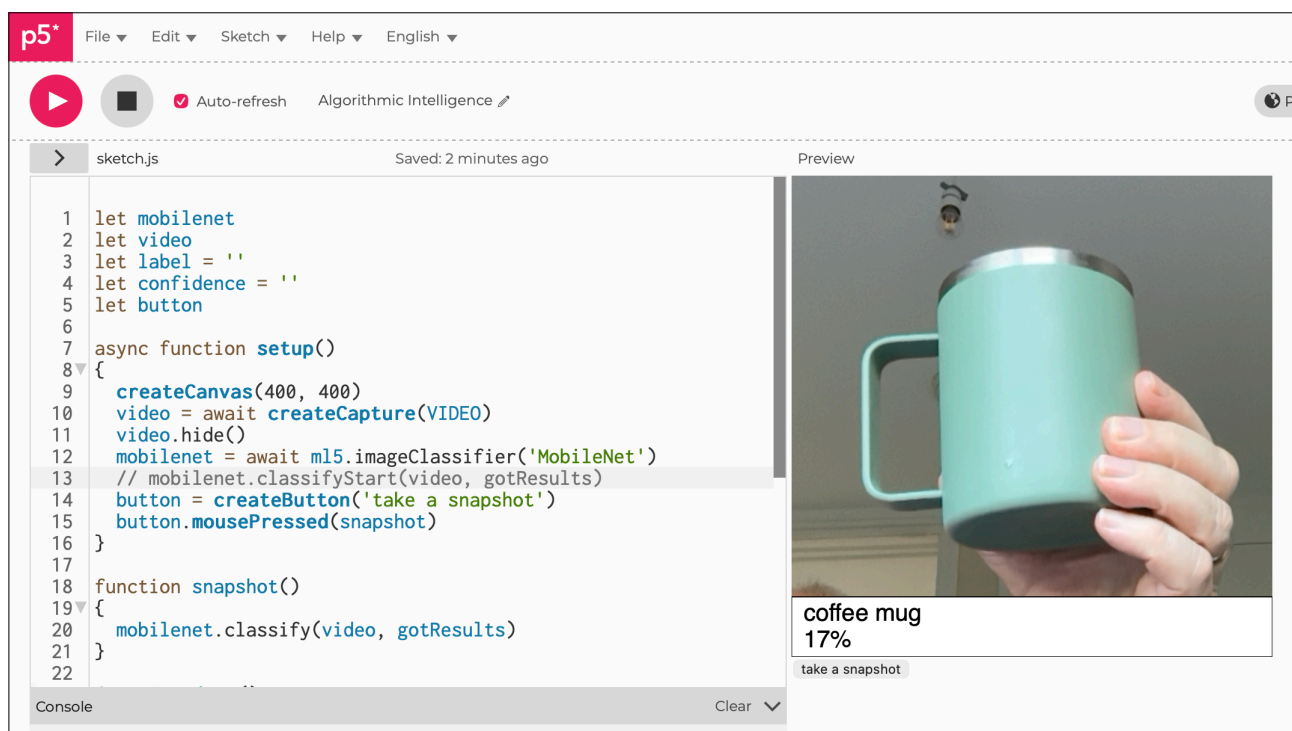
function draw()
{
  image(video, 0, 0)
  fill(255)
  rect(0, height - 50, width, 50)
  textSize(20)
  fill(0)
  text(label, 10, height - 30)
  text(floor(confidence * 100) + '%', 10, height - 8)
}
```

```
function gotResults(results)
{
  label = results[0].label
  confidence = results[0].confidence
}
```

Notes

This is repeatable, so get another object and click on the button.

Figure B8.5



Algorithmic Intelligence Module B Unit #9 CoCo SSD Bounding Box Image Classification



Module B Unit #9: bounding box image classification

We need an image, preferably one that has a few objects in it, not too many and not cluttered. For instance, the one below, I will include it on my website. Upload it the usual way and rename it photo (fruit or whatever). The actual image is quite large, but I have deliberately chosen a square image for simplicity. You can use your own image if you want.

Figure 1: a photo





Sketch B9.1 index.html

Make sure you have the `ml5.js` line of code in your `index.html` file.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch B9.2 the cocossd model

Our starting model, assuming that you have `ml5.js`.

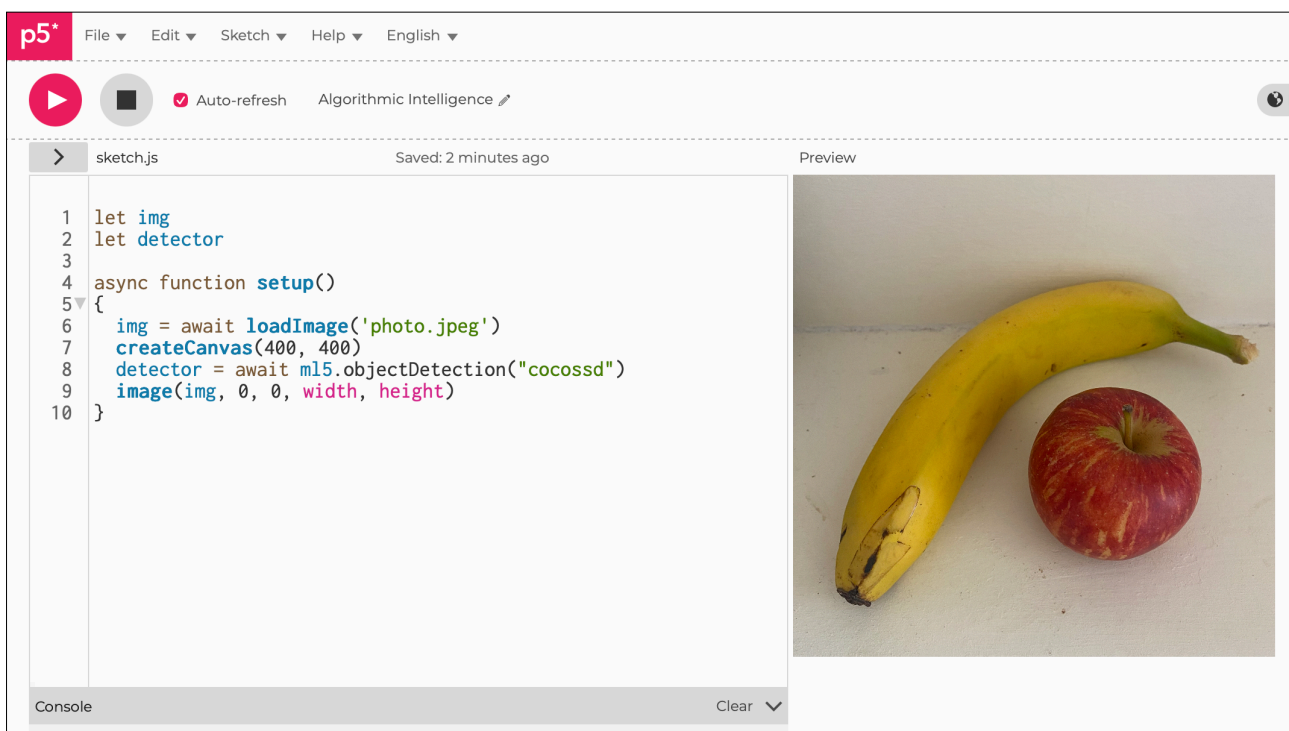
```
let img
let detector

async function setup()
{
  img = await loadImage('photo.jpeg')
  createCanvas(400, 400)
  detector = await ml5.objectDetection("cocossd")
  image(img, 0, 0, width, height)
}
```

Notes

Just the basics.

Figure B9.2





Sketch B9.3 let's go detecting

We need an array to hold the results from the detection.

```
let img
let detector
let detections = []

async function setup()
{
  img = await loadImage('photo.jpeg')
  createCanvas(400, 400)
  detector = await ml5.objectDetection("cocossd")
  detector.detect(img, gotDetections)
  image(img, 0, 0, width, height)
}

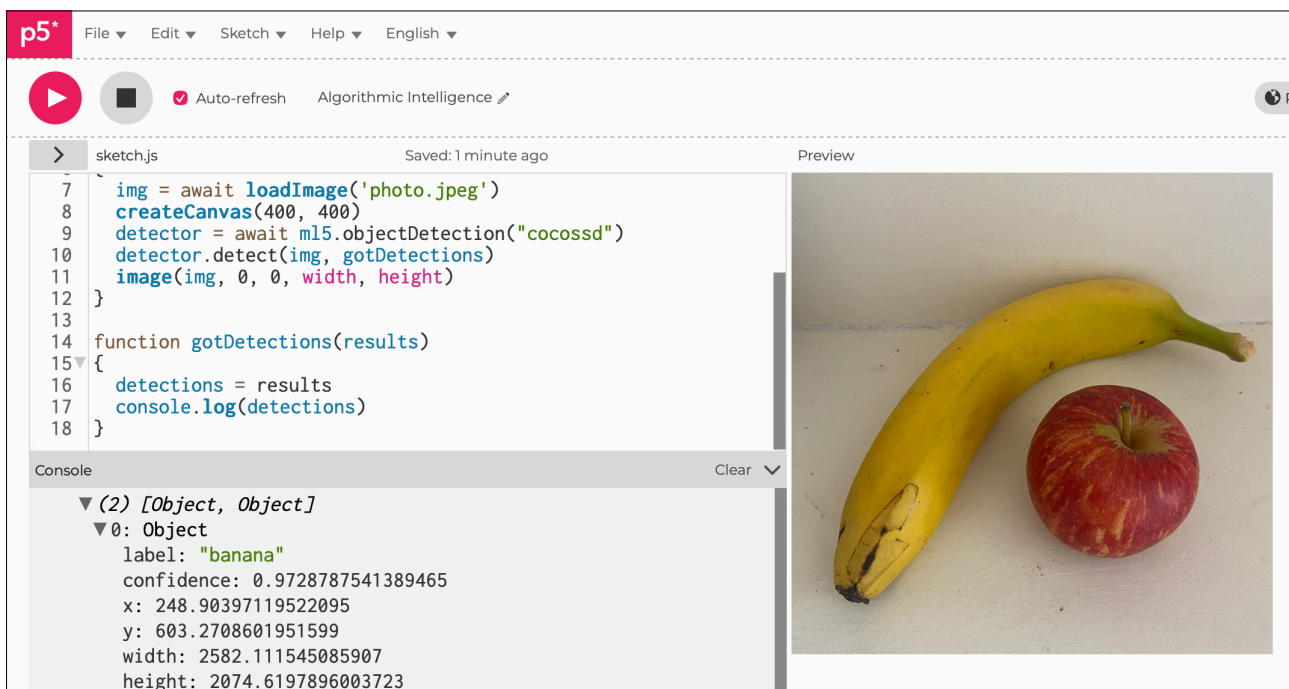
function gotDetections(results)
{
  detections = results
  console.log(detections)
}
```

Notes

If you look in the console, you will see that it returns two objects, and if you open them up, you will see that they are correctly labelled banana and apple with very high confidence scores.

You will also note that the **x**, **y**, **width**, and **height** are extremely large compared to our canvas. That is because the original image is much larger, and we just scaled it to fit conveniently on our canvas.

Figure B9.3





Sketch B9.4 resizing

We will have a problem when we come to draw the bounding box. The dimensions in the array for the banana and the apple will most likely be somewhere off to the right, off-screen. We can mitigate this by using a ratio of image to canvas; also, you want to use an image that is not a square, which this will also help to scale in both dimensions.

```
let img
let detector
let detections = []
let dim

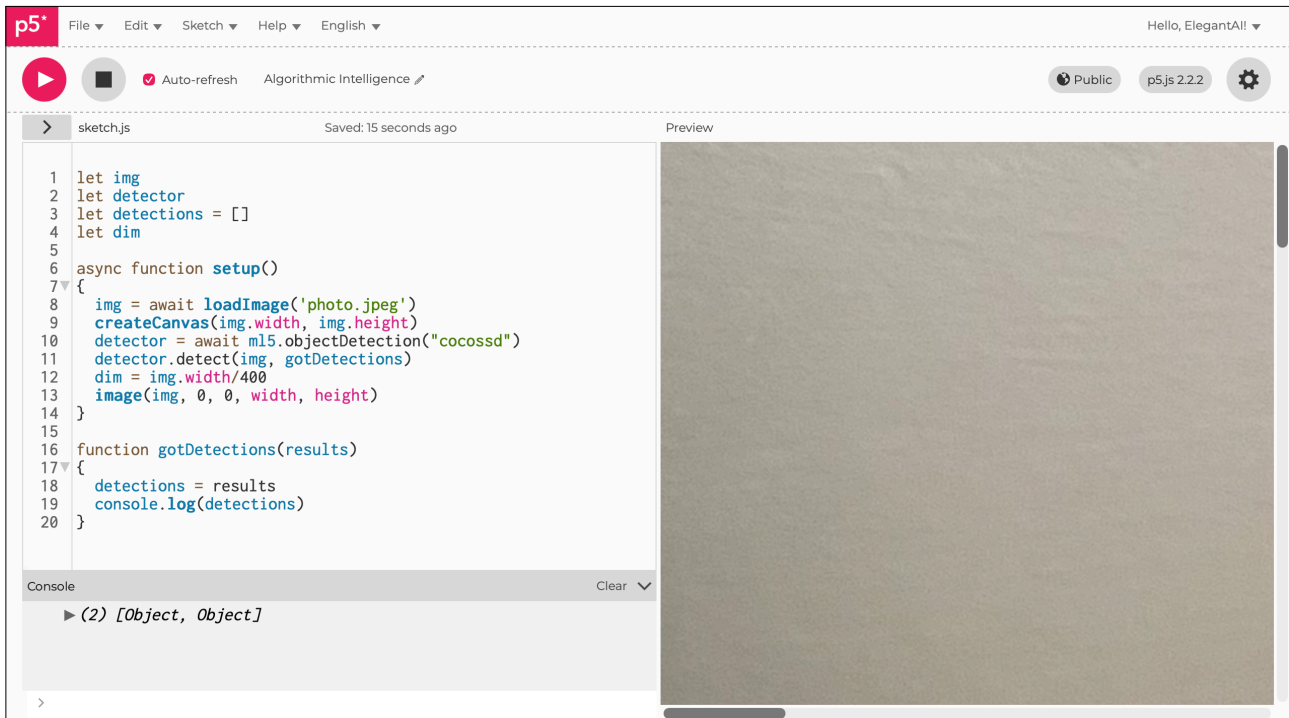
async function setup()
{
  img = await loadImage('photo.jpeg')
  createCanvas(img.width, img.height)
  detector = await ml5.objectDetection("cocossd")
  detector.detect(img, gotDetections)
  dim = img.width/400
  image(img, 0, 0, width, height)
}

function gotDetections(results)
{
  detections = results
  console.log(detections)
}
```

Notes

As you can see, the image is quite large, and we can only see a small part of it. So, we need to resize the image to fit the canvas.

Figure B9.4





Sketch B9.5 rearrange the image

We now have the image of a decent size. The image has been moved into the `draw()` function and removed from the `setup()` function.

```
let img
let detector
let detections = []
let dim

async function setup()
{
  img = await loadImage('photo.jpeg')
  createCanvas(img.width, img.height)
  detector = await ml5.objectDetection("cocossd")
  detector.detect(img, gotDetections)
  dim = img.width/400
  // image(img, 0, 0, width, height)
}

function draw()
{
  image(img, 0, 0, width/dim, height/dim)
}

function gotDetections(results)
{
  detections = results
  console.log(detections)
}
```

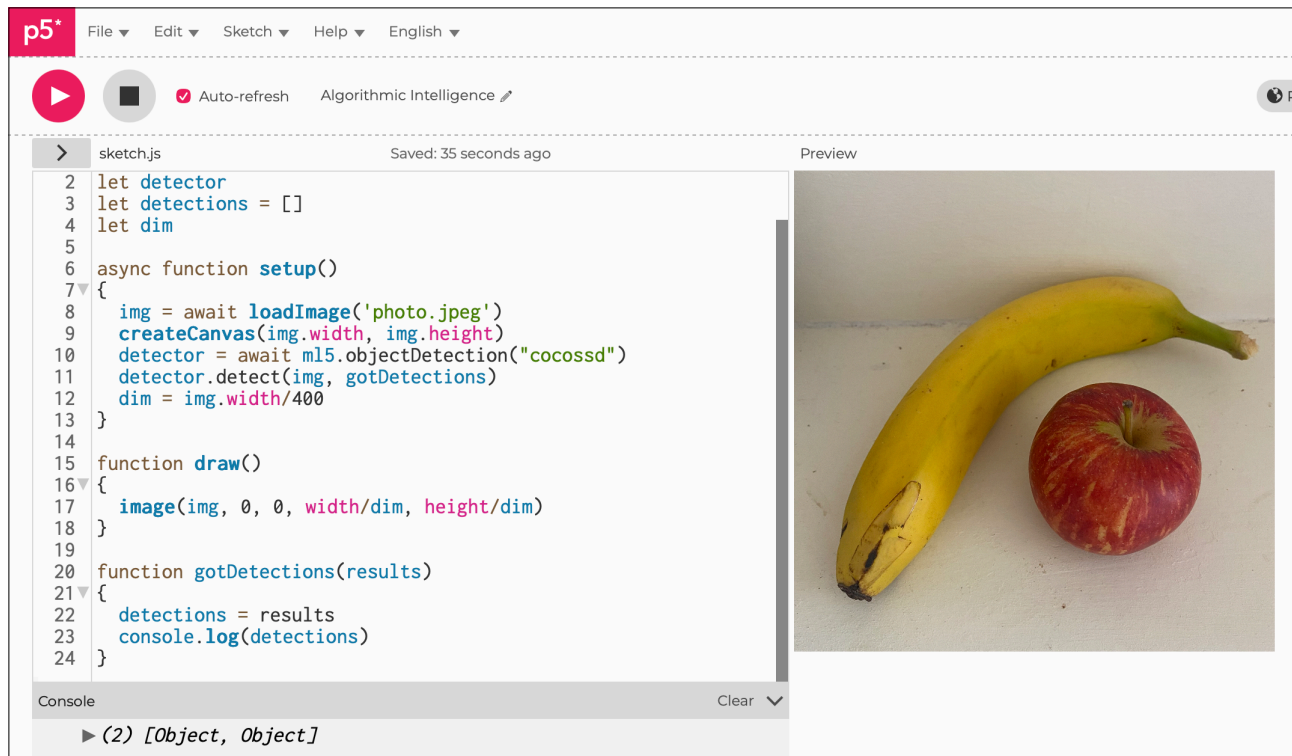
Notes

We have divided the width by the dim value to give us the better canvas image.

Challenges

1. Try other values other than **400**.
2. What happens if the image is not a square?

Figure B9.5





Sketch B9.6 bounding box

Now we have detected the objects in the image and a means of scaling things, we can use the data to draw a box around said objects. Because we have more than one object, we need to use a loop to work through all the objects (you might have many more than two).

```
let img
let detector
let detections = []
let dim

async function setup()
{
  img = await loadImage('photo.jpeg')
  createCanvas(img.width, img.height)
  detector = await ml5.objectDetection("cocossd")
  detector.detect(img, gotDetections)
  dim = img.width/400
}

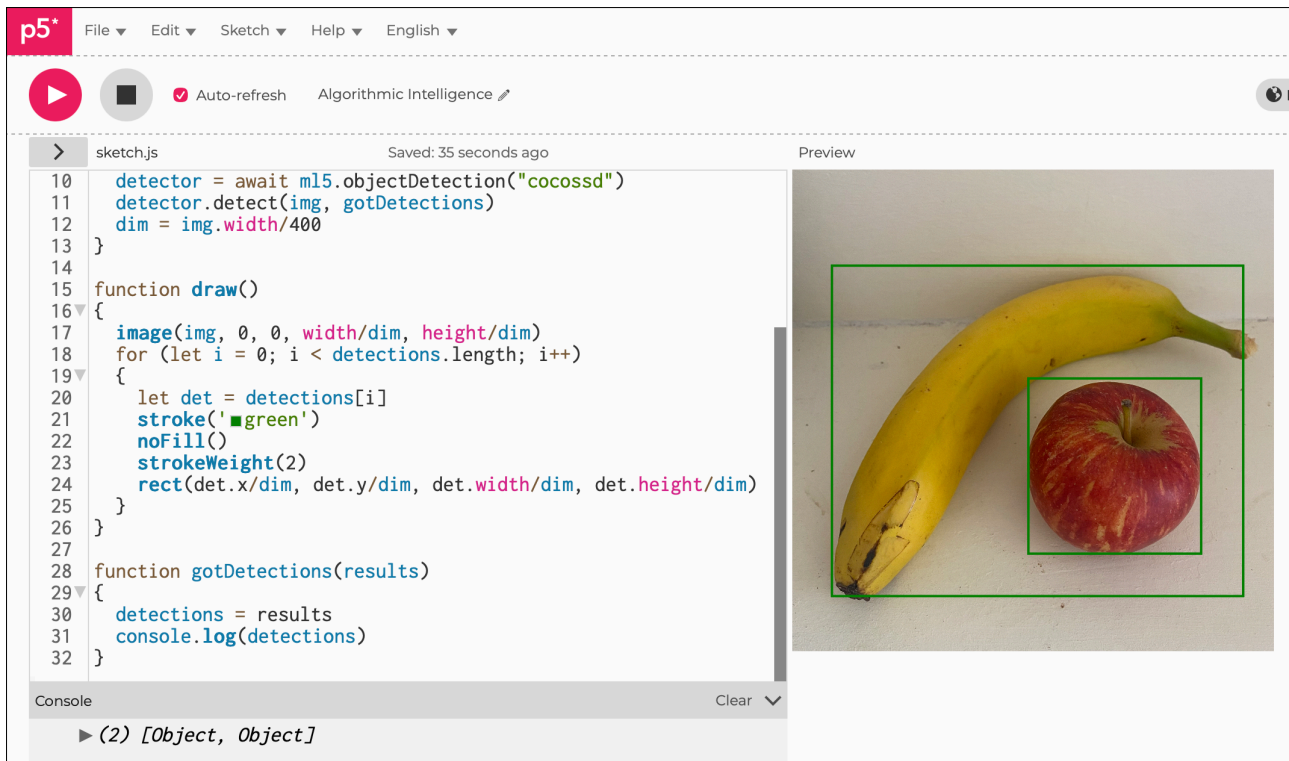
function draw()
{
  image(img, 0, 0, width/dim, height/dim)
  for (let i = 0; i < detections.length; i++)
  {
    let det = detections[i]
    stroke('green')
    noFill()
    strokeWeight(2)
    rect(det.x/dim, det.y/dim, det.width/dim, det.height/dim)
  }
}

function gotDetections(results)
{
  detections = results
  console.log(detections)
}
```

Notes

We get a nice green box for each object.

Figure B9.6





Sketch B9.7 labelling the objects

We can also insert the names of each object inside their respective bounding boxes.

```
let img
let detector
let detections = []
let dim

async function setup()
{
  img = await loadImage('photo.jpeg')
  createCanvas(img.width, img.height)
  detector = await ml5.objectDetection("cocossd")
  detector.detect(img, gotDetections)
  dim = img.width/400
}

function draw()
{
  image(img, 0, 0, width/dim, height/dim)
  for (let i = 0; i < detections.length; i++)
  {
    let det = detections[i]
    stroke('green')
    noFill()
    strokeWeight(2)
    rect(det.x/dim, det.y/dim, det.width/dim, det.height/dim)

    noStroke()
    fill('white')
    textSize(15)
    text(det.label, det.x/dim + 5, det.y/dim + 15)
  }
}

function gotDetections(results)
{
  detections = results
}
```

```
console.log(detections)
}
```

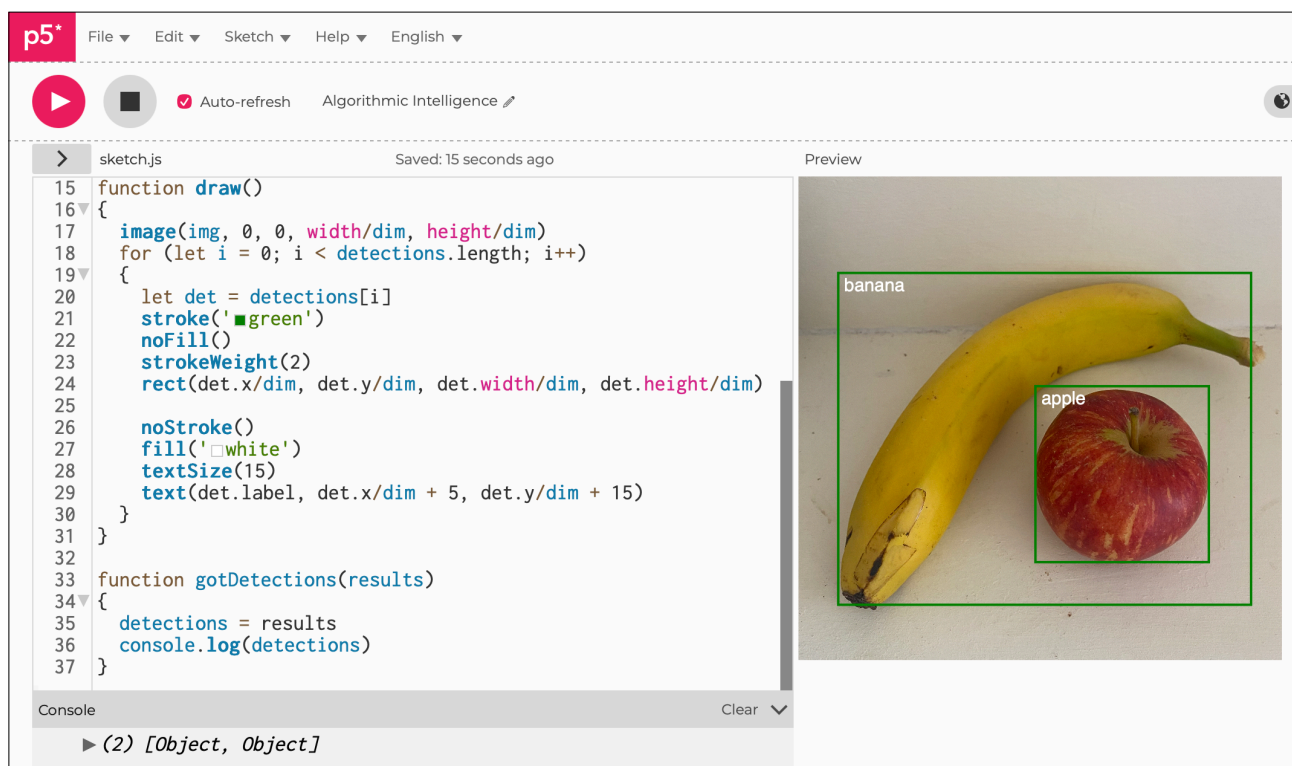
Notes

Clearly labelled and bounded. Suggest removing the `console.log()` if not needed.

Challenge

Try other colours, maybe add a background to the text so that it can be seen against a mixed colour background.

Figure B9.7



Algorithmic Intelligence Module B Unit #10 CoCo SSD Bounding Box Video Classification



Module B Unit #10: bounding box video classification

In the last unit, we looked at a single image. Now we are going to see how well it works with a live stream video from your webcam or other device.



Sketch B10.1 index.html

Make sure you have the `ml5.js` line of code in your `index.html` file.

```
<!DOCTYPE html>
<html lang="en"><head>
  <script src="https://cdn.jsdelivr.net/npm/p5@2.2.2/lib/p5.js"></script>
  <script src="https://unpkg.com/ml5@1/dist/ml5.min.js"></script>
  <link rel="stylesheet" type="text/css" href="style.css">
  <meta charset="utf-8">
</head>
<body>
  <main>
  </main>
  <script src="sketch.js"></script>
</body></html>
```



Sketch B10.2 getting the video stream

! Newish sketch.

First, set up a video stream from the webcam. No point in highlighting the changes, too many, also fairly intuitive.

```
let video
let detector

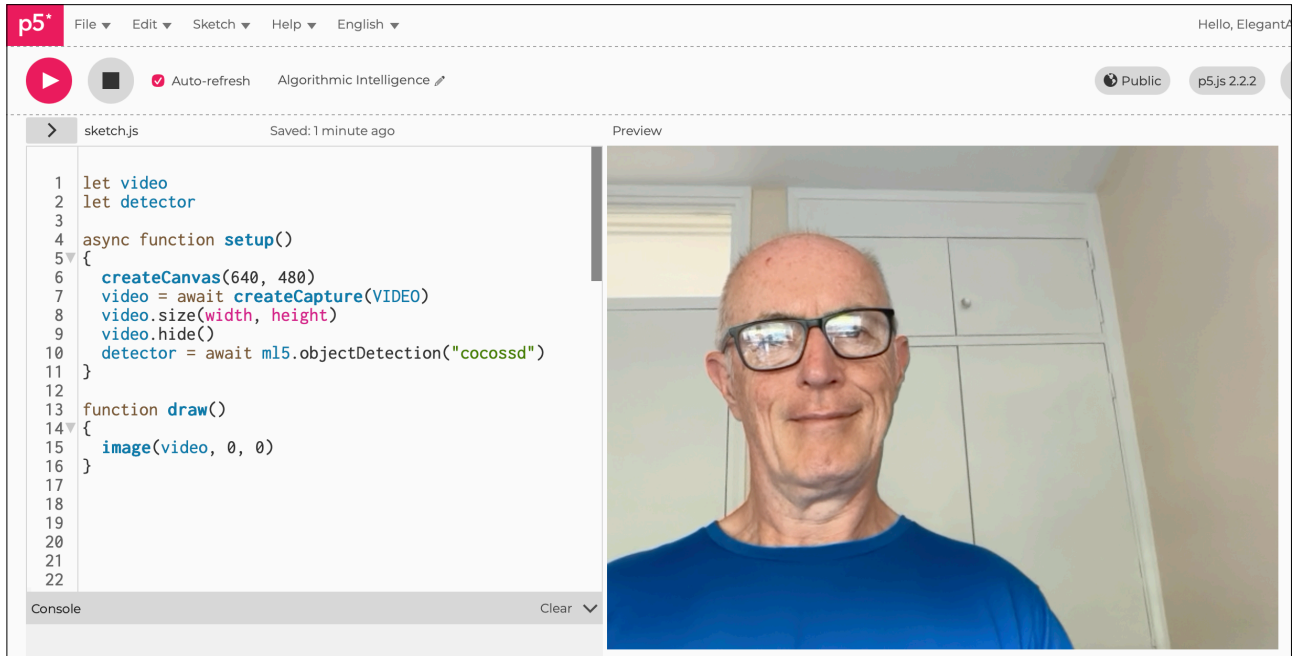
async function setup()
{
  createCanvas(640, 480)
  video = await createCapture(VIDEO)
  video.size(width, height)
  video.hide()
  detector = await ml5.objectDetection("cocossd")
}

function draw()
{
  image(video, 0, 0)
}
```


Notes

You should have a video image; you may have to allow access.

Figure B10.2





Sketch B10.3 now to detect

Let's start detecting something or somebody. Initialised the variable **detections** to **0** to avoid error messages later.

```
let video
let detector
let detections = 0

async function setup()
{
  createCanvas(640, 480)
  video = await createCapture(VIDEO)
  video.size(width, height)
  video.hide()
  detector = await ml5.objectDetection("cocossd")
  detector.detectStart(video, gotDetections)
}

function draw()
{
  image(video, 0, 0)
}

function gotDetections(results)
{
  detections = results
  console.log(detections)
}
```

Notes

You may get more than one object detected depending on what is in the background. You can pause the sketch and take a look in the console.



Sketch B10.4 drawing the bounding box

The bounding box is just as before, although this time it moves with the object or comes and goes.

```
let video
let detector
let det
let detections = 0

async function setup()
{
  createCanvas(640, 480)
  video = await createCapture(VIDEO)
  video.size(width, height)
  video.hide()
  detector = await ml5.objectDetection("cocossd")
  detector.detectStart(video, gotDetections)
}

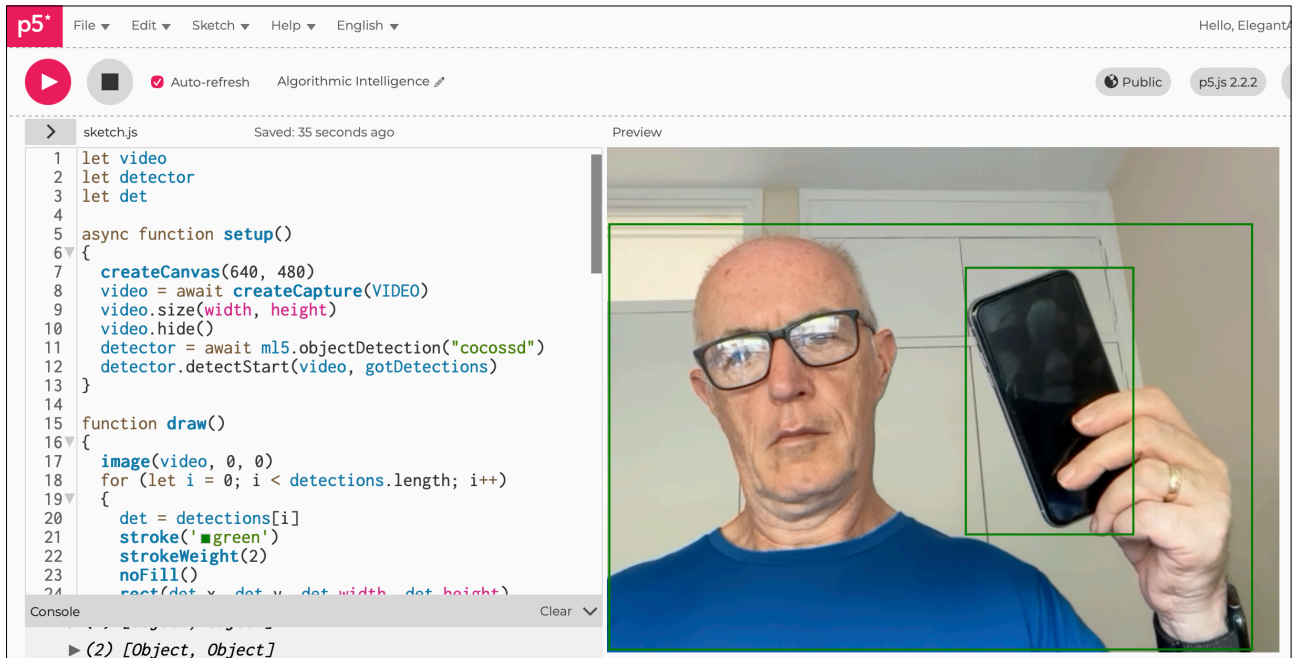
function draw()
{
  image(video, 0, 0)
  for (let i = 0; i < detections.length; i++)
  {
    det = detections[i]
    stroke('green')
    strokeWeight(2)
    noFill()
    rect(det.x, det.y, det.width, det.height)
  }
}

function gotDetections(results)
{
  detections = results
  console.log(detections)
}
```

Notes

You should have a bounding box, and if you hold something up (like a mobile phone), it should clock that as well.

Figure B10.4





Sketch B10.5 labelling

We can put the labels in, even though you could stop/pause the programme and look inside the array.

```
let video
let detector
let det
let detections = 0

async function setup()
{
  createCanvas(640, 480)
  video = await createCapture(VIDEO)
  video.size(width, height)
  video.hide()
  detector = await ml5.objectDetection("cocossd")
  detector.detectStart(video, gotDetections)
}

function draw()
{
  image(video, 0, 0)
  for (let i = 0; i < detections.length; i++)
  {
    det = detections[i]
    stroke('green')
    strokeWeight(2)
    noFill()
    rect(det.x, det.y, det.width, det.height)
    noStroke()
    fill('white')
    textSize(24)
    text(det.label, det.x + 10, det.y + 24)
  }
}

function gotDetections(results)
{

```

```
detections = results
console.log(detections)
}
```

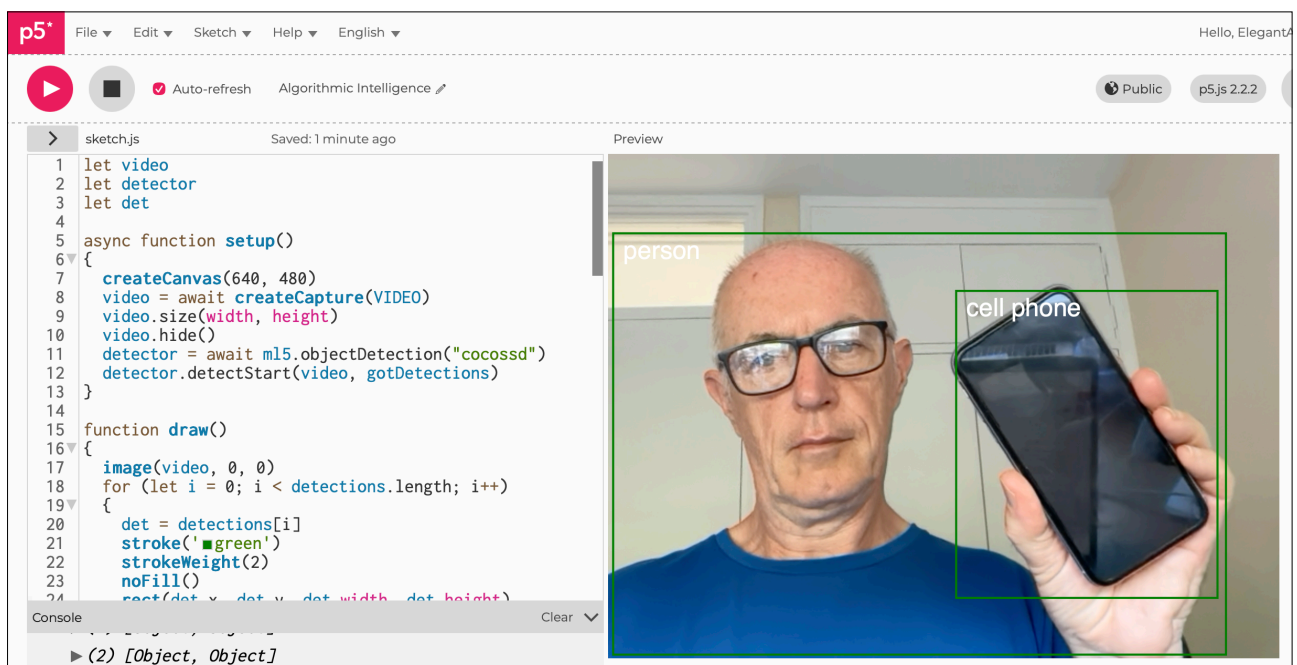
Notes

There is a limit to the number of objects it has been trained on, so you may well get some interesting labelling.

Challenge

Add the confidence score after label.

Figure B10.5



Algorithmic Intelligence Module C Unit #1 Introducing Transformers



Module C Unit #1: introduction to transformers

Transformers took off when a paper produced by some researchers at Google called **Attention is All You Need**. It literally transformed (no pun intended) the world of large language models, such as ChatGPT. A copy of this paper can be found at this link:

<https://arxiv.org/pdf/1706.03762>

Figure 1: The published paper

arXiv:1706.03762v7 [cs.CL] 2 Aug 2023

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaizer@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

I suspect you will already have used generative AI such as ChatGPT, Gemini, or Claude. These Large Language Models (LLMs) are extremely large, and the machines (servers) that do the work on your behalf are housed in huge warehouses called data centres and cost billions to build, run, and maintain.

This unit is not about using or incorporating these chatbots in p5.js but to introduce you to an alternative which is still extremely powerful, and they are called Small Language Models (SLMs). The beauty of these is that they are open-sourced, they are free, and operate in your browser rather than a server somewhere in the States or China.



Small Language Models

You may be wondering where you can get these SMLs and how you can use them with `p5.js`. The great thing is that Dan Shiffman (at [The Coding Train](#)) has already introduced these in his teaching material.

To integrate transformers into a `p5.js` sketch (in the browser), we need a transformer that works in the browser, and there is a library available called `transformer.js`. These can be found on the Hugging Face website.

The beauty of this approach is that no data leaves your computer. You are in control. The downside could be that it won't perform like a top-end LLM.



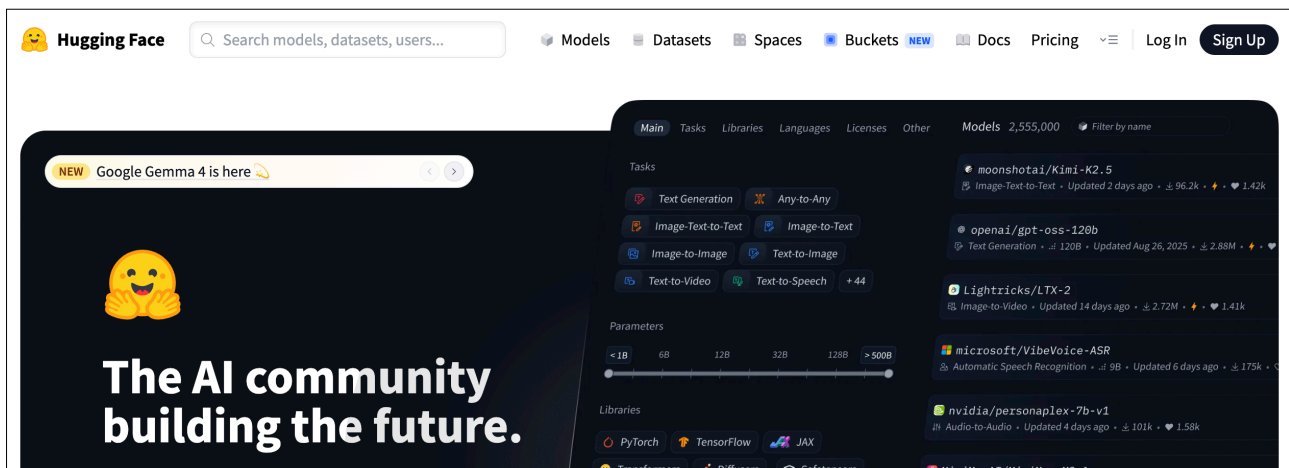
Hugging Face

The website address is:

<https://huggingface.co>

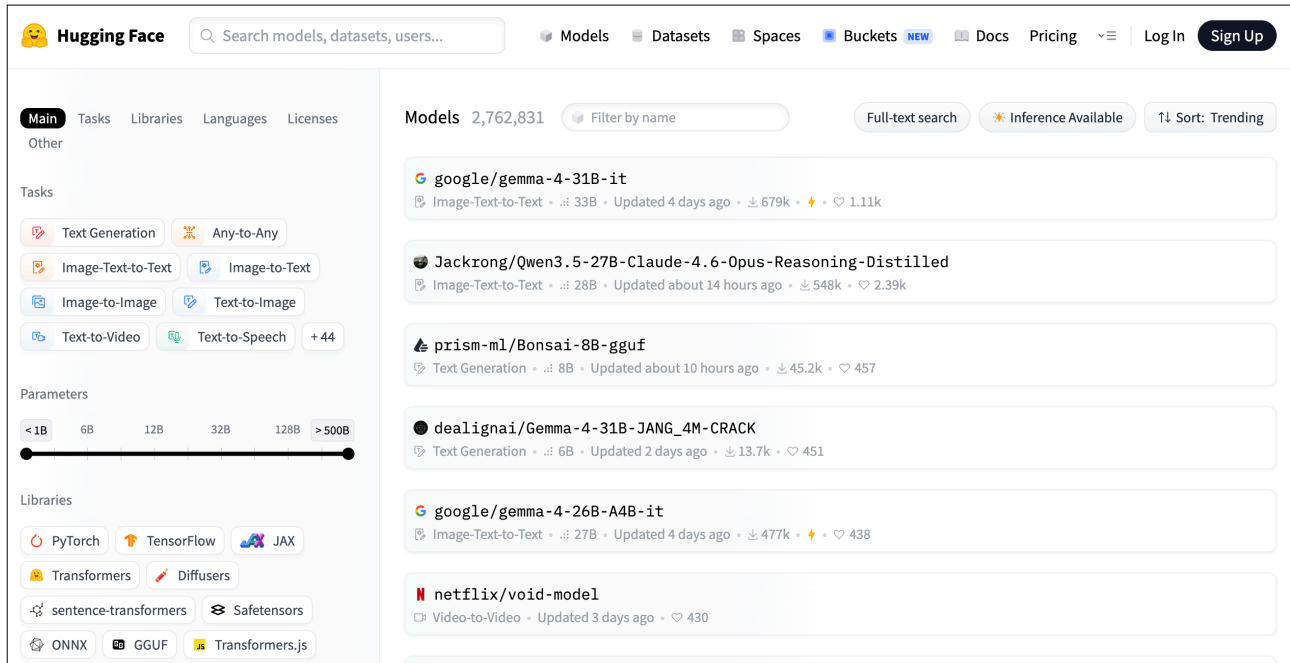
1) Go to that **website**:

Figure 2: hugging face website



2) Select **Models** (tab at the top):

Figure 3: models tab

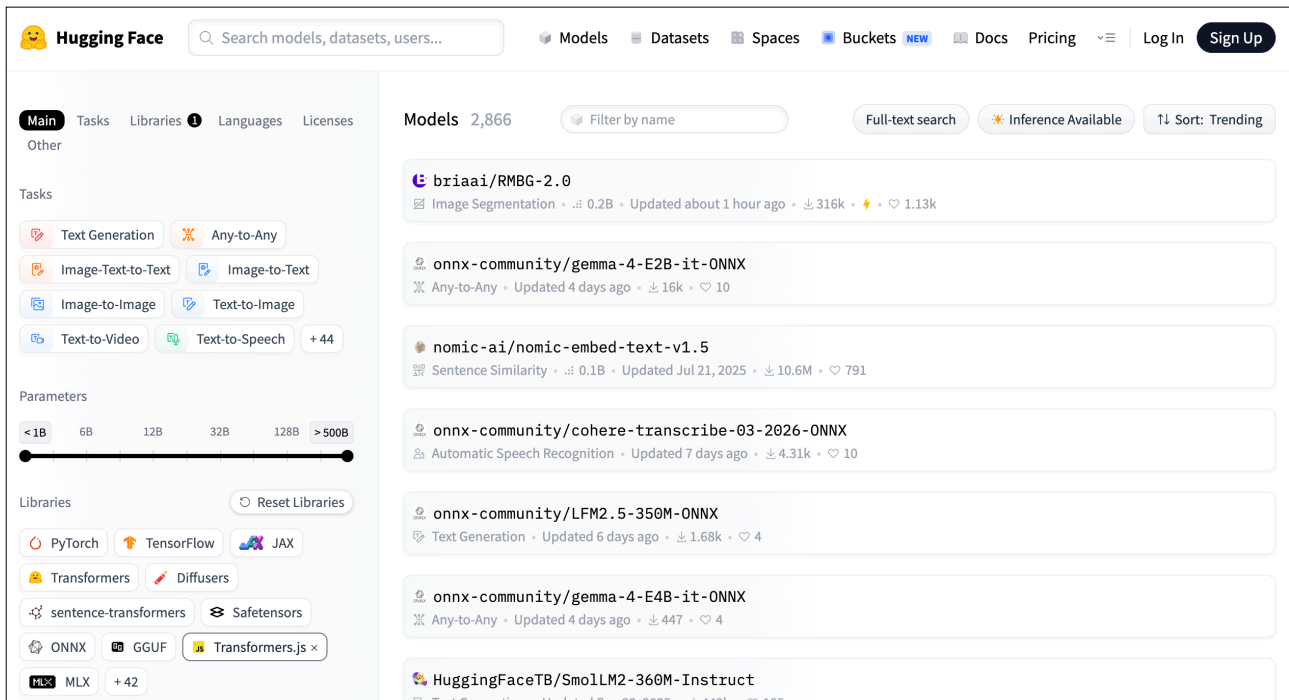




Transformer.js models

3) Next, click on the **transfers.js** button (near the bottom in Fig. 4):

Figure 4: transformers.js models



This gives you a list of the available **models** using the **transformer.js** library. There are a lot of them, 96 pages of them.



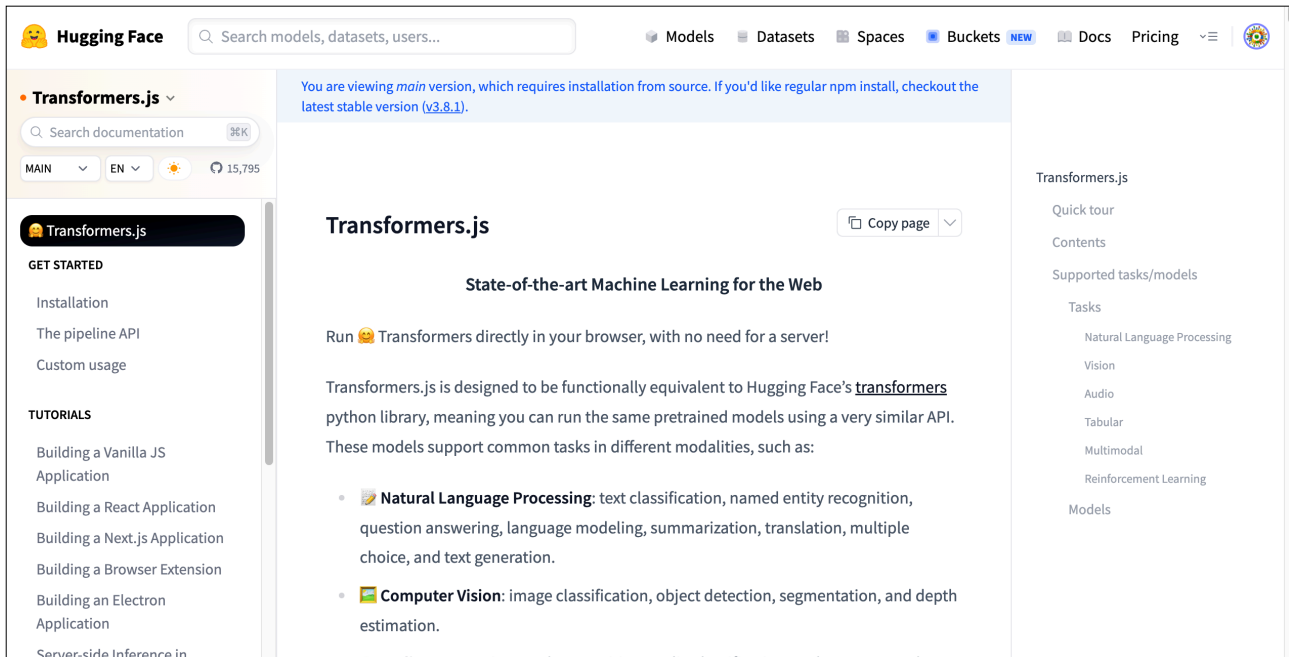
Pipeline API

To look at the documentation for transformers.js, go to:

<https://huggingface.co/docs/transformers.js/index>

You will see a page like this:

Figure 5: transformers.js documentation



API stands for: Application Programming Interface

We need to import the **API** pipeline using **JavaScript** (not **Python**). So scroll down the page (and scroll across if the text is large), see Fig. 6.

This is an example:

Figure 6: API pipeline example

The screenshot shows the Transformers.js documentation page. On the left is a sidebar with navigation links under 'GET STARTED' (Installation, The pipeline API, Custom usage) and 'TUTORIALS' (Building a Vanilla JS Application, Building a React Application, Building a Next.js Application, Building a Browser Extension, Building an Electron Application, Server-side Inference in Node.js). The main content area is titled 'Quick tour' and explains that pipelines group a pretrained model with preprocessing and postprocessing. It features a comparison table between Python (original) and JavaScript (ours) code. Below the table, it notes that a different model can be specified by providing a model ID or path as the second argument to the pipeline function.

Python (original)	JavaScript (ours)
<pre>import pipeline line for sentiment-analysis sentiment-analysis') a transformers!') ITIVE', 'score': 0.999806941}]</pre>	<pre>import { pipeline } from '@huggingface/transformers'; // Allocate a pipeline for sentiment-analysis const pipe = await pipeline('sentiment-analysis'); const out = await pipe('I love transformers!'); // [{label: 'POSITIVE', 'score': 0.999817686}]</pre>

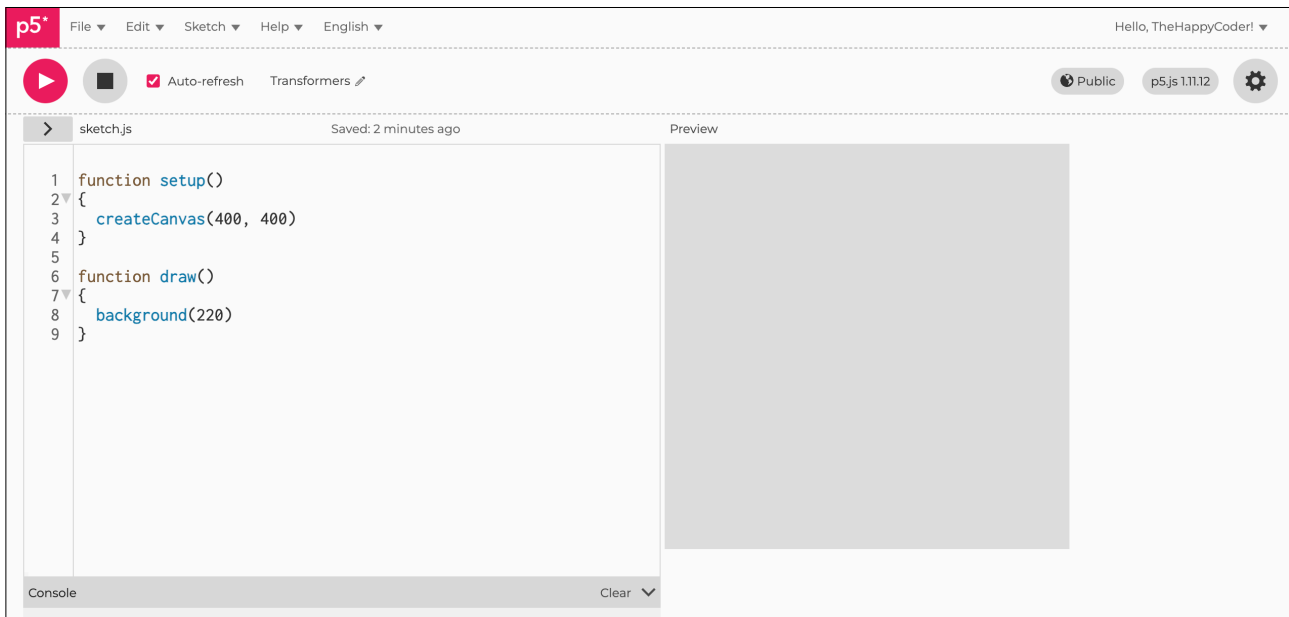
The pipeline in curly brackets **{ pipeline }** means it is an object with lots of properties.



The p5.js version

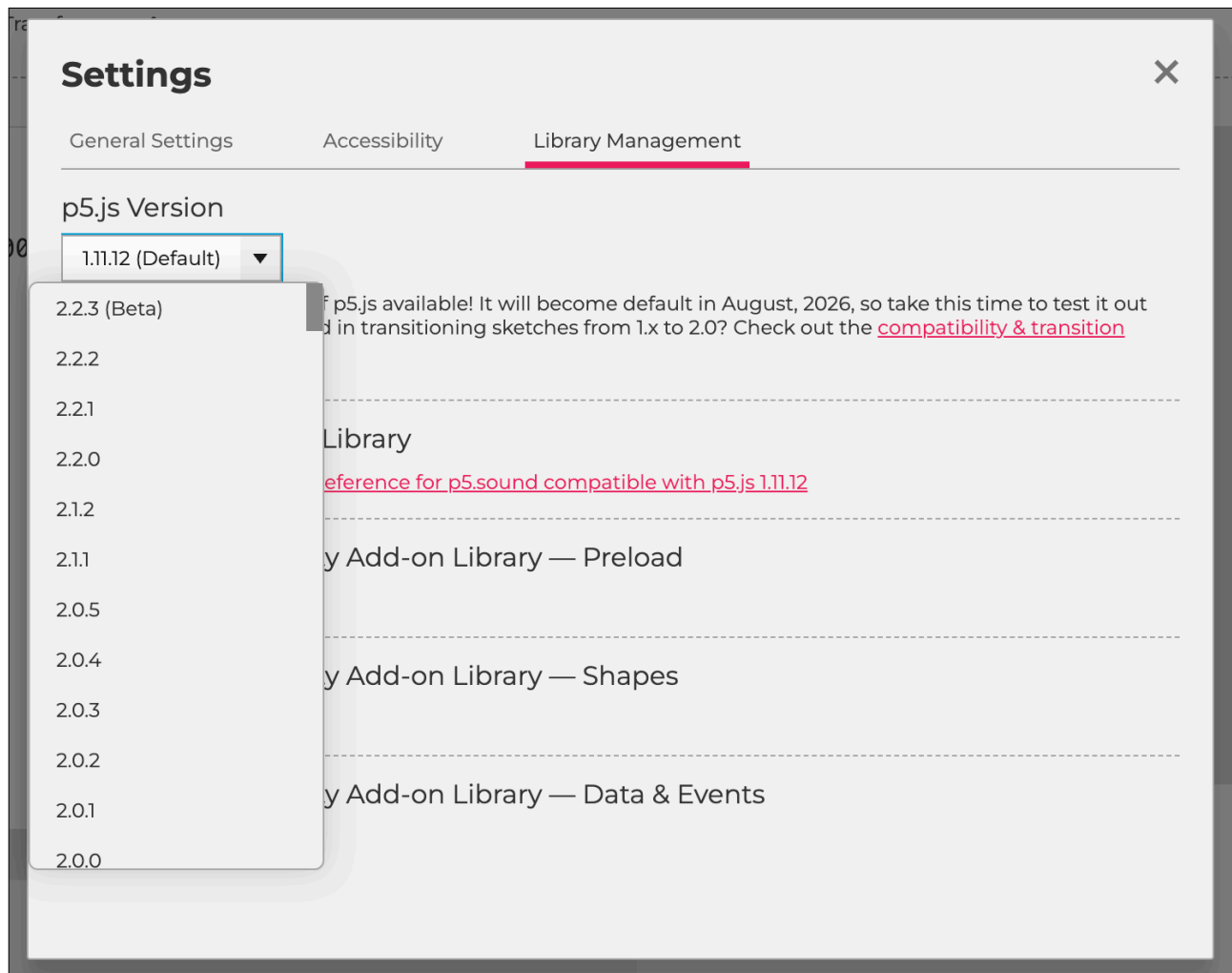
Before we run this example, you do need to be using the latest version of **p5.js**. If you are reading this before it switches automatically, you could well be running it as version **1.11.12**, as shown below in Fig. 7. The version is indicated in the top right-hand corner.

Figure 7: default (current) version



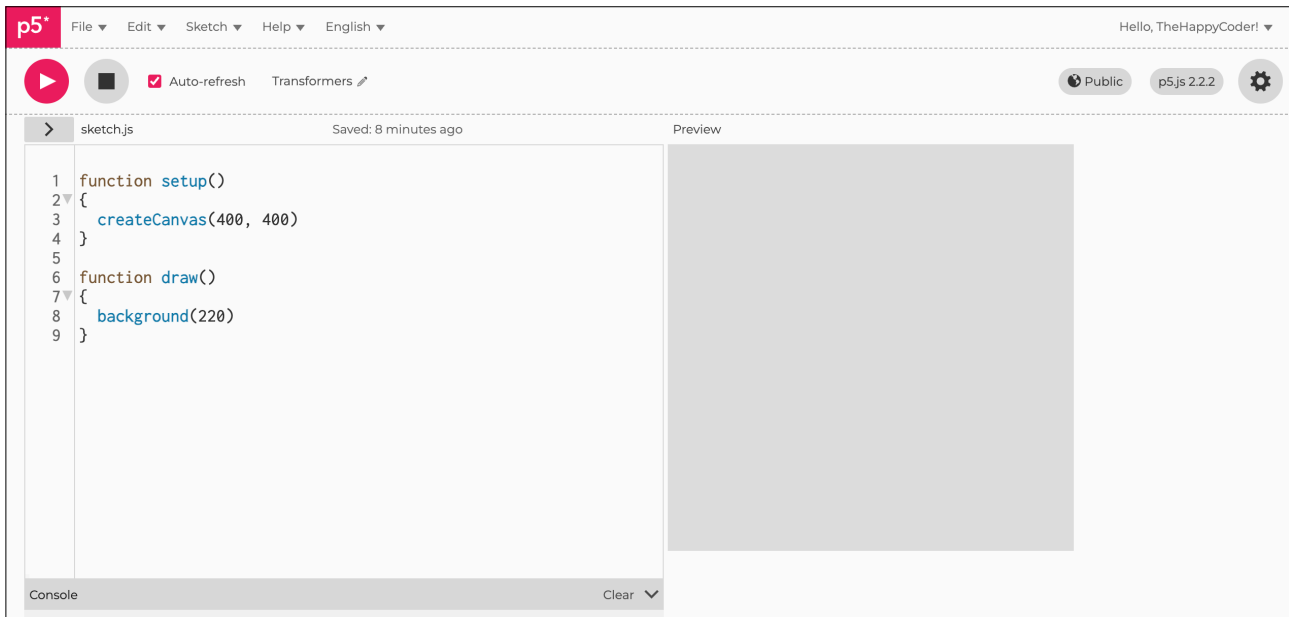
To change it, you simply click on that button and you will get a drop-down menu of all the other versions when you click on the arrow shown in Fig. 8. Select the latest (not beta) version and voilà.

Figure 8: version 2.2.2 or later



Your version should now show the latest version on your page, see Fig. 9.

Figure 9: latest version of p5.js (currently)





The import problem

The `p5.js` language doesn't use the `import` function as shown in the example that we found in the documentation. Our first step is to use `async` and `await`.

```
let { pipeline } = await import('@huggingface/transformers')
```

This still won't work, but we are getting closer. We need the proper URL for this library. To get it, you can simply use the search bot by typing in CDN for `transformers.js`. The result is:

```
let { pipeline } = await import("https://cdn.jsdelivr.net/npm/@huggingface/transformers@4.0.1")
```

If you don't put the version at the end, it will default to the latest version. This is called **object destructuring**. We are taking the `pipeline` property and putting it in its own variable.

We use `async` and `await` as used in version 2 to load the library asynchronously.



Sentiment analysis

We create the pipeline for sentiment analysis with:

```
let pipe = await pipeline('sentiment-analysis')
```

As shown in the example.

We can send to and get data from that pipeline with:

```
let results = await pipe('I love transformers!')
```

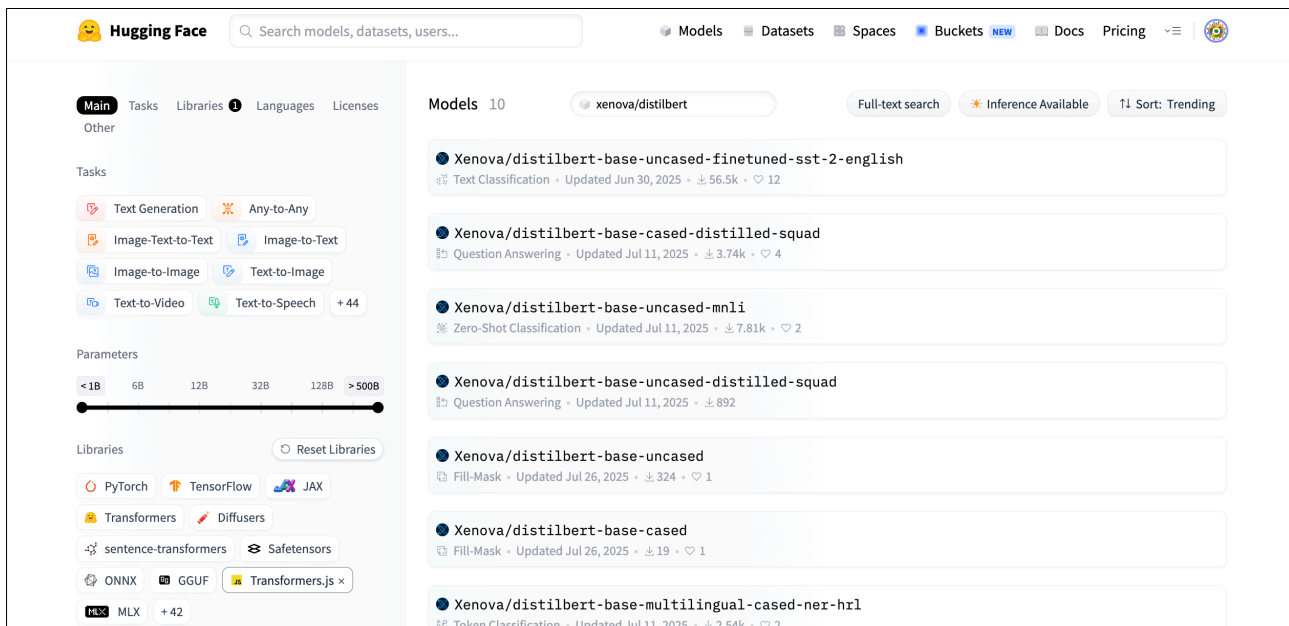
This will give back a sentiment score.



Specifying a model

In this case, we want to specify a specific model rather than just the example default. It will give us the same result as sentiment analysis. For this, we go back to the main page on the Hugging Face website, select the models tab, and select the `transformers.js` button.

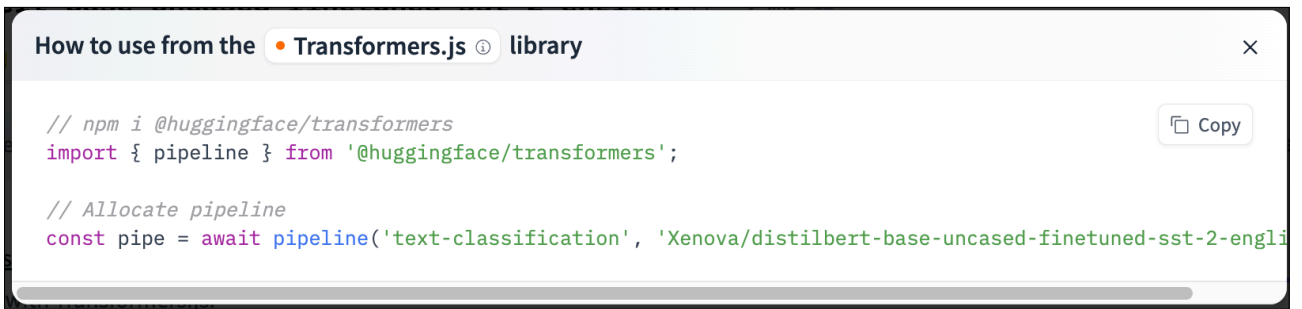
Figure 10: models and transformers.js



Type in the name of the model, by the person who created it. In this example, we are going to use a model by **Xenova** and a specific model called **distilbert-base-uncased-finetuned-sst-2-english**. You only need to start typing the first couple of words.

Select the one you want; it will take you to the page for that model. You can see an extract to help you get started. Here, you can copy the bit of the code that matters by clicking on the **use this model** button. And it will give you a pop-up window (Fig. 11) with the code.

Figure 11: the code for the model



```
How to use from the Transformers.js library ×

// npm i @huggingface/transformers
import { pipeline } from '@huggingface/transformers';

// Allocate pipeline
const pipe = await pipeline('text-classification', 'Xenova/distilbert-base-uncased-finetuned-sst-2-english');
```

You just need the bit in the brackets:

```
('text-classification', 'Xenova/distilbert-base-uncased-finetuned-sst-2-english')
```



Sketch C1.1 starting sketch

This is what we have in our sketch so far. I have tidied it up a bit to make it a bit more readable. Also, in the image below, I have reduced the text size so you can see the code in full without end-of-line wrap.

```
async function setup()
{
  createCanvas(400, 400)
  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')
  let classifier = await pipeline('text-classification', 'Xenova/distilbert-base-uncased-finetuned-sst-2-english')
  let results = await classifier('I love transformers!')
  console.log(results)
}

function draw()
{
  background(220)
}
```

Notes

You can see that we have a highly positive sentiment analysis.

Challenge

Change the text to: **I hate transformers.**

Figure C1.1

The image shows a web-based p5.js editor interface. At the top, there is a menu bar with 'File', 'Edit', 'Sketch', 'Help', and 'English'. Below the menu bar, there are controls for running the sketch (a play button) and a checkbox for 'Auto-refresh' which is checked. The main area displays a file named 'sketch.js' with the following code:

```
1 async function setup()
2 {
3   createCanvas(400, 400)
4   let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')
5   let classifier = await pipeline('text-classification', 'Xenova/distilbert-base-uncased-finetuned-sst-2-english')
6   let results = await classifier('I love transformers!')
7   console.log(results)
8 }
9
10 function draw()
11 {
12   background(220)
13 }
```

Below the code editor, there is a console window showing the output of the code. It displays a single log entry: (1) [Object]. This object contains a 'label' property with the value 'POSITIVE' and a 'score' property with the value 0.999788761138916. The console window also has a 'Clear' button.



Sketch C1.2 text box and button

! Remove the `console.log()` and the `draw()` function.

We create a text box and a submit button. We keep the original text input **I love transformers!** for now. We can add text later.

```
let textInput
let submitButton

async function setup()
{
  createCanvas(400, 400)
  background(220)
  textInput = createInput(' ')
  textInput.position(50, 50)
  submitButton = createButton('submit')
  submitButton.position(200, 50)

  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')

  let classifier = await pipeline('text-classification', 'Xenova/distilbert-base-uncased-finetuned-sst-2-english')

  let results = await classifier('I love transformers!')
}
```

Notes

This won't do anything just yet. We will be making more changes shortly but for now we build it slowly.



Sketch C1.3 sentiment

We need something to happen when we click the **submit** button (after we add the text we want analysing).

```
let textInput
let submitButton
let classifier

async function setup()
{
  createCanvas(400, 400)
  background(220)
  textInput = createInput(' ')
  textInput.position(50, 50)
  submitButton = createButton('submit')
  submitButton.position(200, 50)
  submitButton.mousePressed(sentiment)

  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')
  let classifier = await pipeline('text-classification', 'Xenova/distilbert-base-uncased-finetuned-sst-2-english')
  let results = await classifier('I love transformers!')
}

async function sentiment()
{
}
```

Notes

Nothing is going to happen yet as we haven't put anything in that function.



Sketch C1.4 text input

! Remove: `let results = await classifier('I love transformers!')`. It will now give a result on the canvas plus the score (as a percentage).

```
let textInput
let submitButton
let classifier

let txt
let results

async function setup()
{
  createCanvas(400, 400)
  background(220)
  textInput = createInput('I love transformers')
  textInput.position(50, 50)
  submitButton = createButton('submit')
  submitButton.position(200, 50)
  submitButton.mousePressed(sentiment)
  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')
  classifier = await pipeline('text-classification', 'Xenova/distilbert-base-uncased-finetuned-sst-2-english')
}

async function sentiment()
{
  background(220)
  txt = textInput.value()
  results = await classifier(txt)
  let { label, score } = results[0]
  textSize(20)
  text(label, 50, 100)
  text(floor(score * 100) + '%', 175, 100)
}
```

Notes

We can get the text (`txt`) input from the textbox and run `inference` on it to determine if it is negative or positive. We can get the results of the label and the score from the results array.

Challenges

1. Put in: `add text here if you care about anything`
2. Try: How much do you like transformers?
3. You could colour code the result and have lots of other things happen, such as emojis.

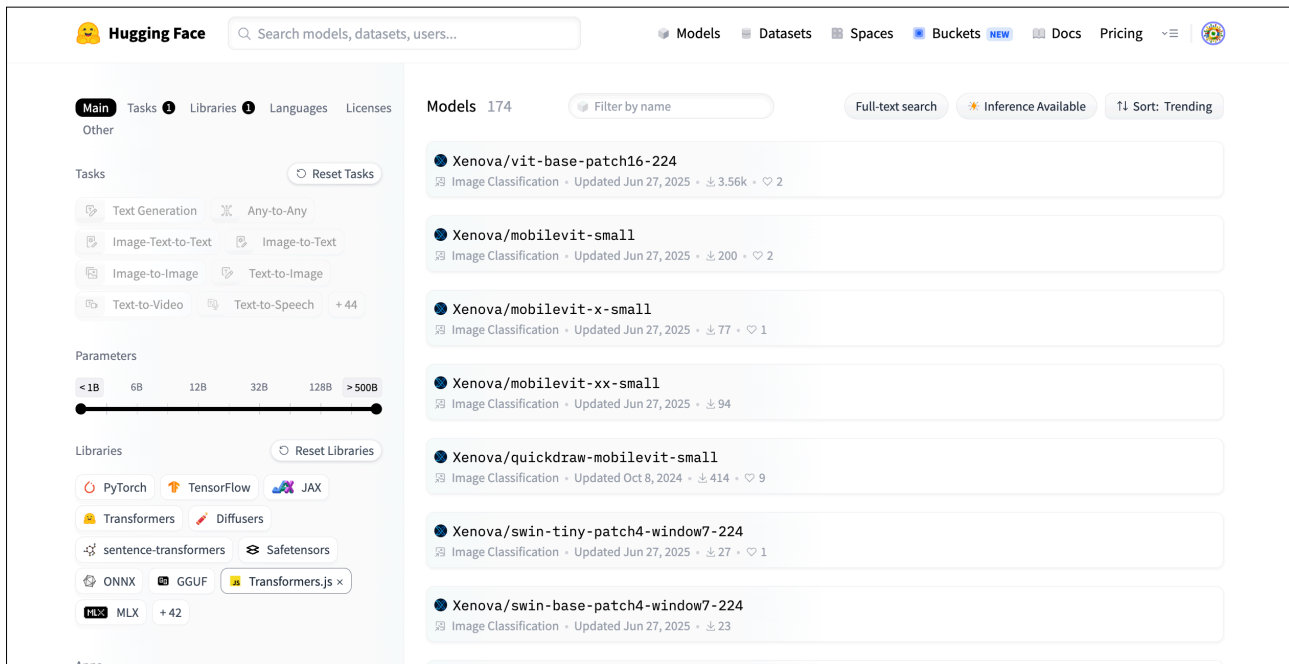
Algorithmic Intelligence Module C Unit #2 transformer image classifier



Module C Unit #2: image classification

In this example from **Hugging Face**, we make use of an image classifier. First, we select **transformers.js** from the main tab.

Figure 1: main tab - transformers.js

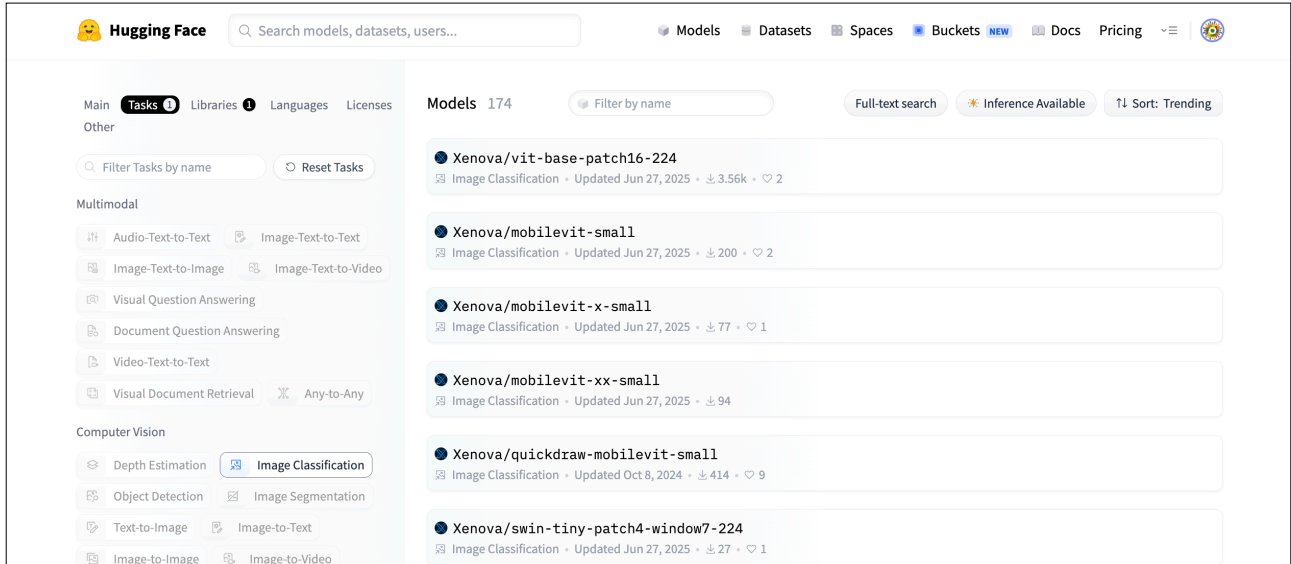




Choosing the task

From the **task** tab, we select **image classification**.

Figure 2: image classifier task

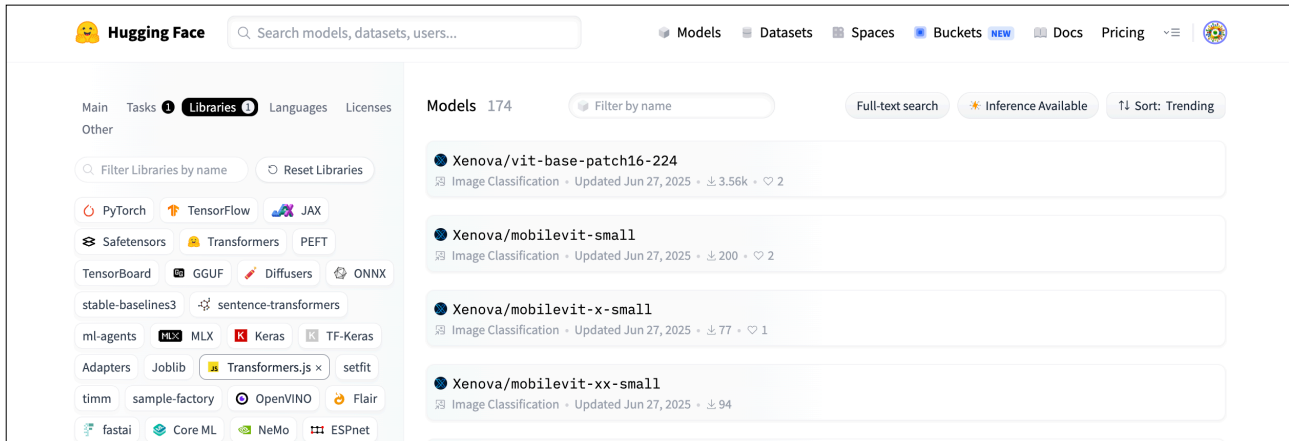




The task tab

In case you didn't select the library, we can do that from the library tab and select `transformers.js`.

Figure 3: task tab keck

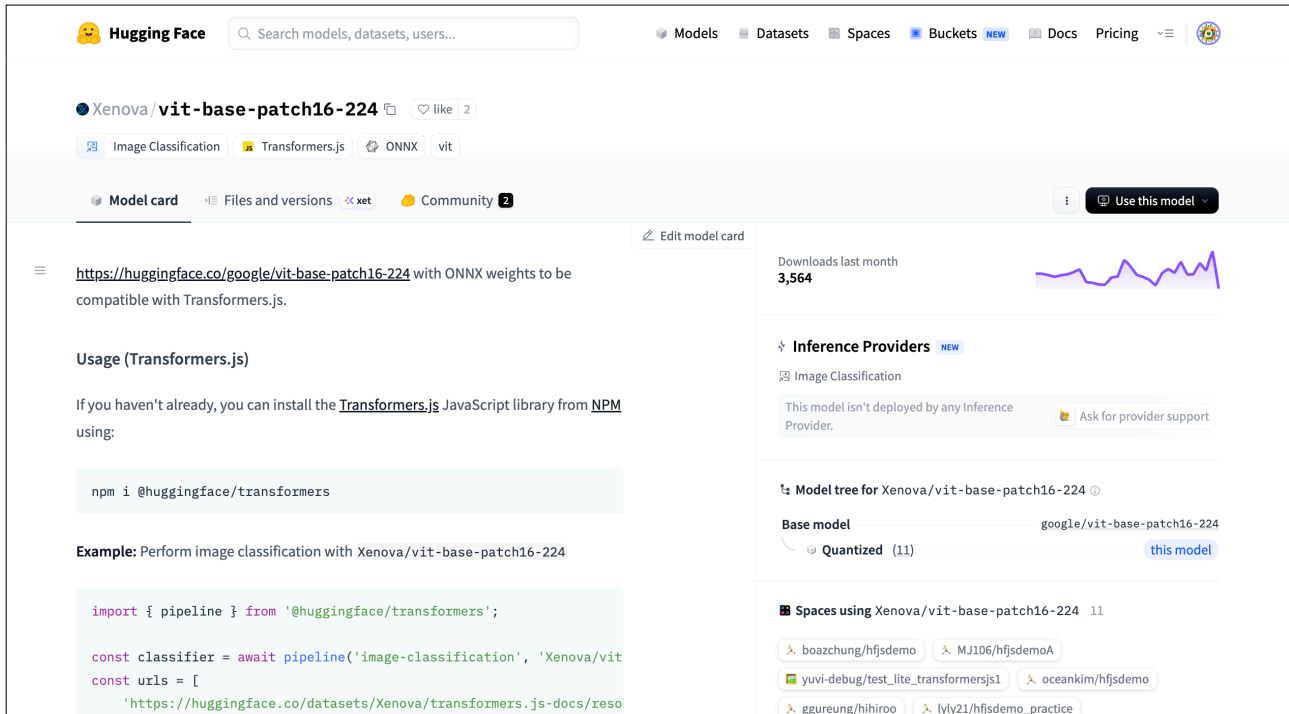




Select one

We select the top one, and we get the web page that corresponds to it.

Figure 4: corresponding web page



We want some help in getting started. First off, we click on the **use this model** tab, which will ask us to use the **transformers.js** library. Click on that, and you get the following information, which is what we need to insert into our code.

```
import { pipeline } from '@huggingface/transformers';
```

```
const pipe = await pipeline('image-classification', 'Xenova/vit-base-patch16-224');
```

The bit we want is highlighted in blue. We put it inside the speech marks inside the pipeline as before.



Their example

However, that is not all; we want to use an image. To help us, we can look at their example on the left-hand side of the page. We can copy it as such:

```
import { pipeline } from '@huggingface/transformers';

const classifier = await pipeline('image-classification', 'Xenova/
vit-base-patch16-224');
const urls = [
  'https://huggingface.co/datasets/Xenova/transformers.js-docs/
resolve/main/tiger.jpg',
  'https://huggingface.co/datasets/Xenova/transformers.js-docs/
resolve/main/cats.jpg',
];
const output = await classifier(urls);
// [
//   { label: 'tiger, Panthera tigris', score:
0.6074584722518921 },
//   { label: 'Egyptian cat', score: 0.8246098756790161 }
// ]
```

The bit that we are interested in is highlighted in blue. Instead of a URL, we can upload an image and call that image directly. It will load the image and classify it. We will also display it on the canvas.



Upload the dog

Upload the dog image (as we have done many times before). You can use your own image or click on the button for one provided. We just call it dog (remember to rename it in the file).

Figure 5: download dog image



Figure 6: our canine friend

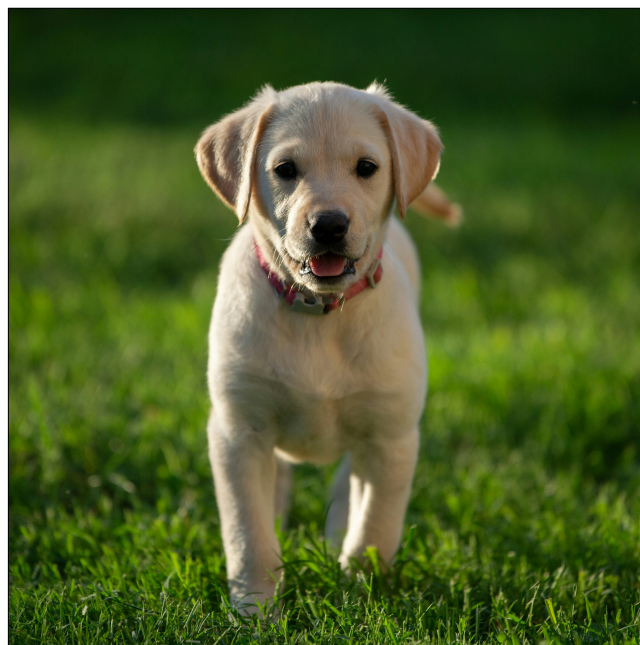
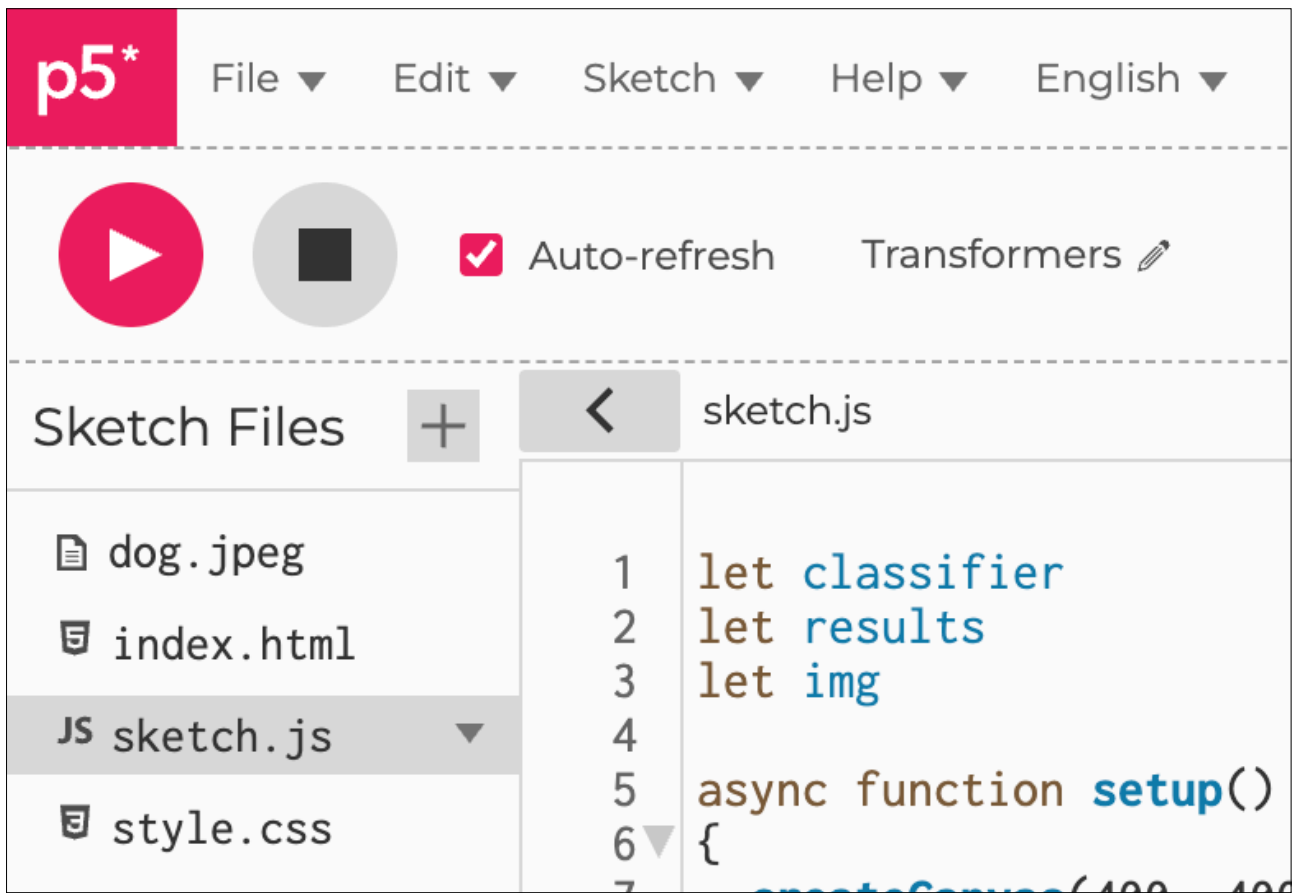


Figure 7: Sketch File (dog added)





Sketch C2.1 the code

This is virtually the same code.

! Remove reference to the **button** and the **text box** and **text**.

```
let classifier
let results
let img

async function setup()
{
  createCanvas(400, 400)
  background(220)

  img = await loadImage('dog.jpeg')
  image(img, 0, 0, 400, 400)

  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')

  classifier = await pipeline('image-classification', 'Xenova/vit-base-patch16-224')
  results = await classifier('dog.jpeg')
  fill('white')

  let { label, score } = results[0]
  textSize(20)
  text(label, 10, 350)
  text(floor(score * 100) + '%', 10, 380)
}
```

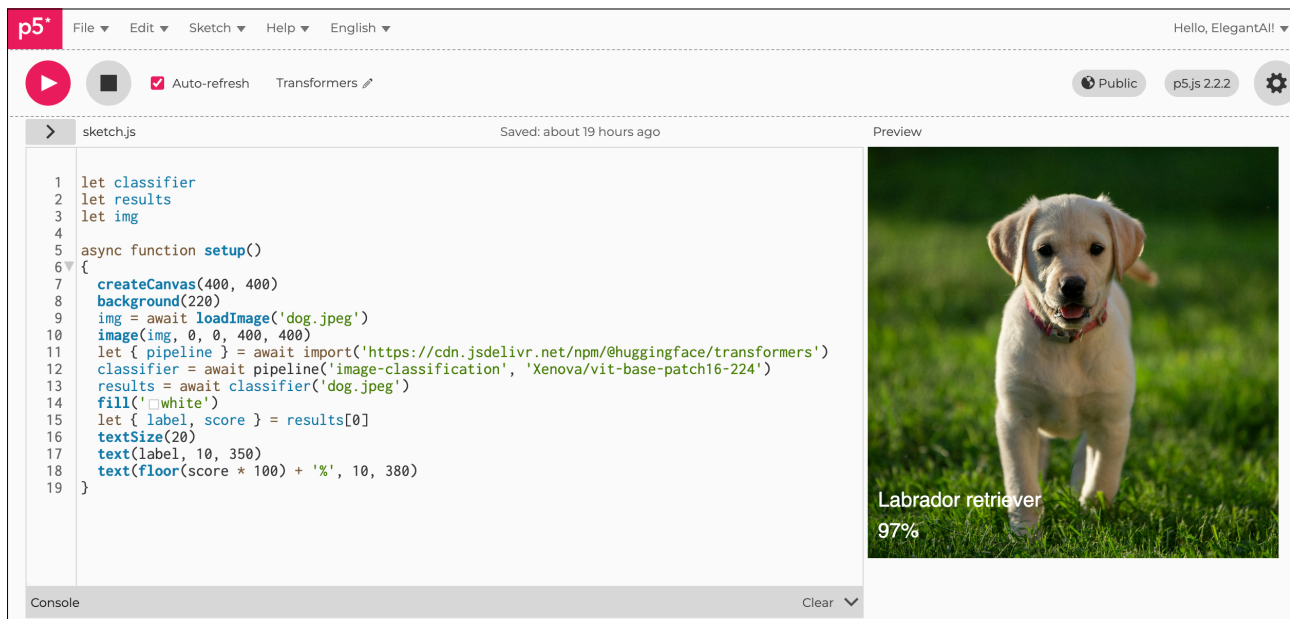
Notes

You can see from the results that we have a good match.

Challenge

Try other animals or objects.

Figure C2.1



Algorithmic Intelligence Module C Unit #3 transformer depth analysis



Module C Unit #3: depth analysis

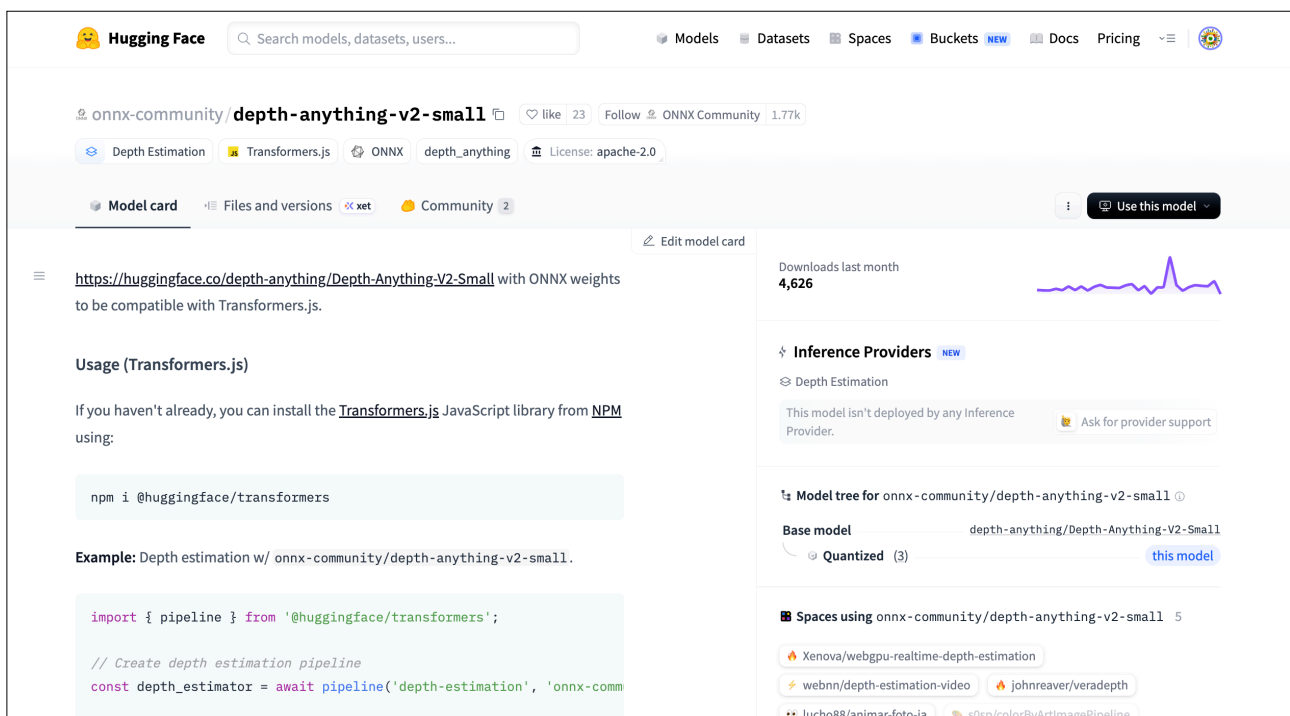
There are numerous tasks. Another one is depth analysis. We follow the same procedure as before. We will use the same photo as before, so keep most of the code from before. Here, it will try to estimate depth in the photo. The brighter the colour, the nearer it is.

I changed the task to depth analysis and scrolled down till I found one I liked the look of. In this case, I chose:

`onnx-community/depth-anything-v2-small`

I recommend that you look it up yourself and see how I got the code to work.

Figure 1: depth analysis



If you haven't still go it in your sketch from the last unit, download the dog image as we have done before, then upload it as a file. We just call it dog (remember to rename it in the file).



Sketch C3.1 the code

Our starting code is below. Very similar to the last one with a different model. We will save the resulting image as a PNG.

```
let depth_estimator
let img

async function setup()
{
  createCanvas(400, 400)
  background(220)
  img = await loadImage('dog.jpeg')
  image(img, 0, 0, 400, 400)
  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')
  depth_estimator = await pipeline('depth-estimation', 'onnx-community/depth-anything-v2-small')
  const { depth } = await depth_estimator('dog.jpeg')
  depth.save('depth.png')
}
```

Notes

The main addition is the `depth.save()` function; this should result in a PNG being uploaded. The image Fig.5 below:

Figure C3.1

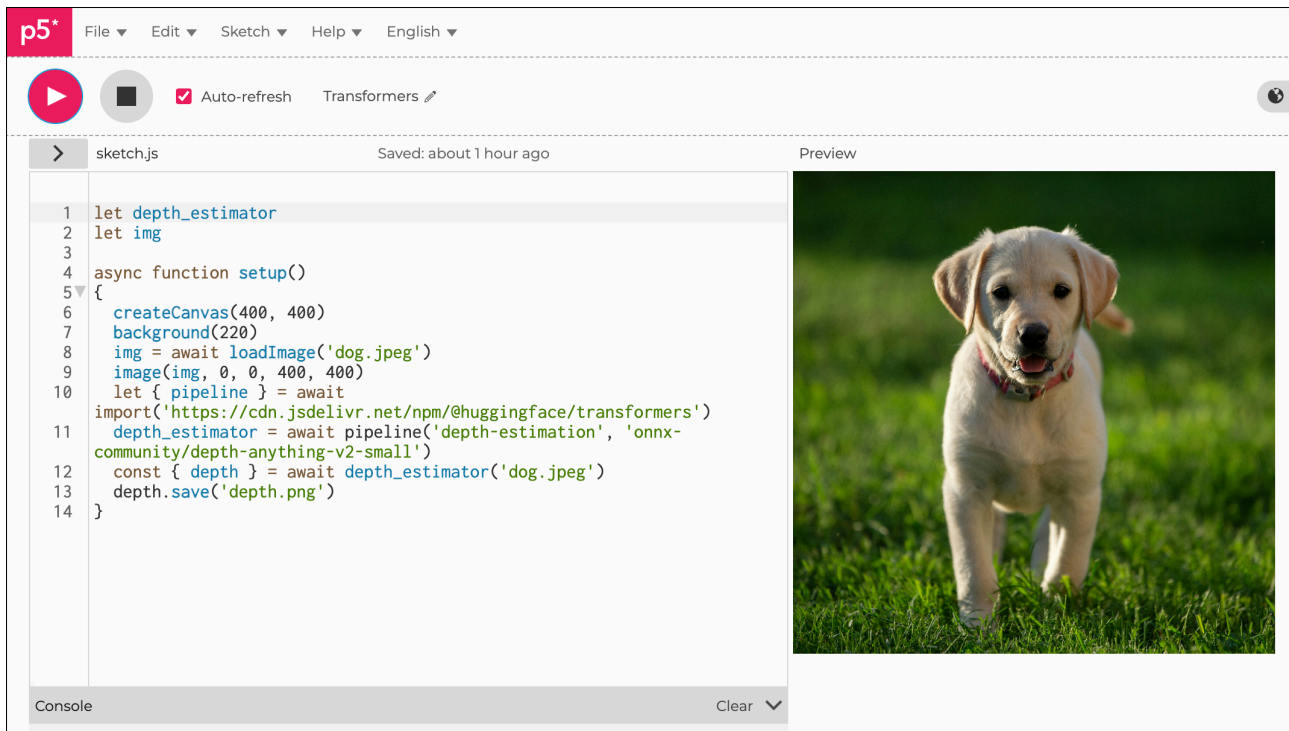


Figure 5: depth analysis



Algorithmic Intelligence Module C Unit #4 transformer image detection



Module C Unit #4: image detection

This looks for objects it can detect and puts a bounding box (plus label) around that object. This is useful if you want to train a model to detect specific objects.

Upload the dog image as we have done before, use your own or click on the button for one provided. We just call it dog (remember to rename it in the file).



Sketch C4.1 image detection

This puts a bounding box around an object it detects. In this case, it is the dog from the previous sketches.

```
let results
let img

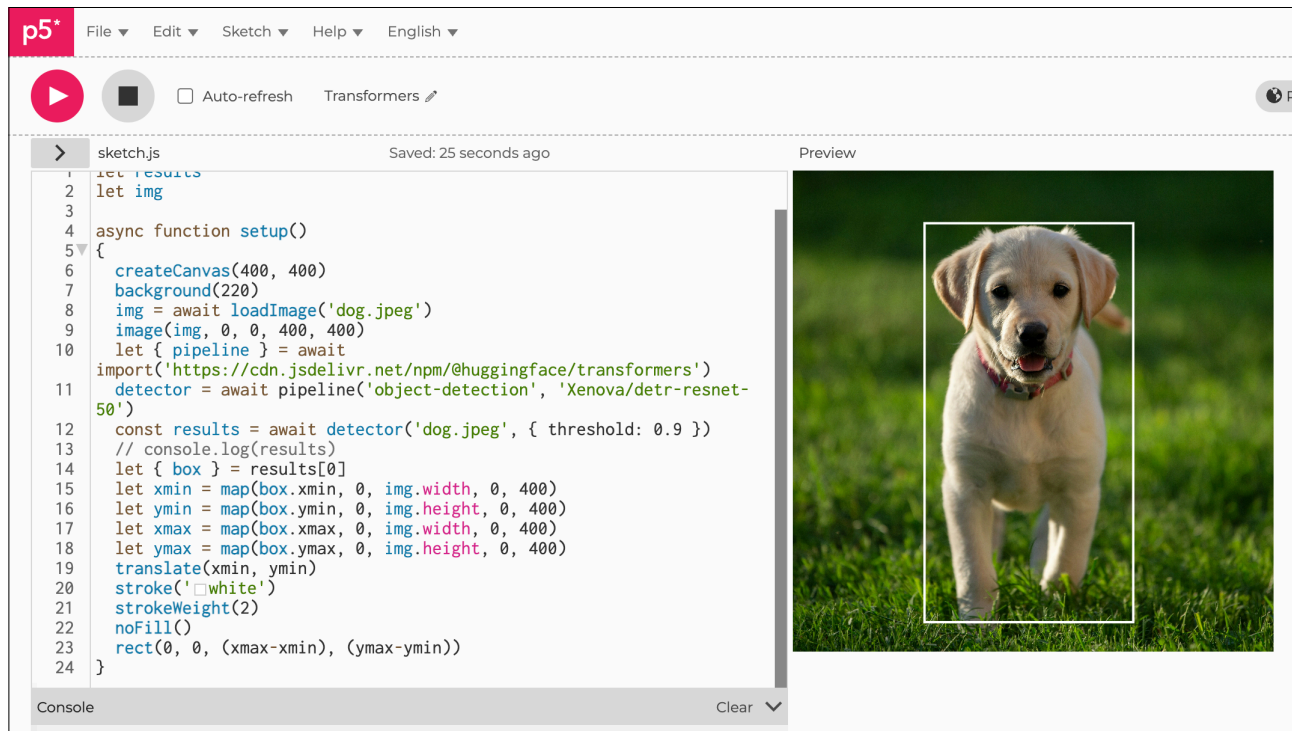
async function setup()
{
  createCanvas(400, 400)
  background(220)
  img = await loadImage('dog.jpeg')
  image(img, 0, 0, 400, 400)

  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')
  detector = await pipeline('object-detection', 'Xenova/detr-resnet-50')
  const results = await detector('dog.jpeg', { threshold: 0.9 })
  // console.log(results)
  let { box } = results[0]
  let xmin = map(box.xmin, 0, img.width, 0, 400)
  let ymin = map(box.ymin, 0, img.height, 0, 400)
  let xmax = map(box.xmax, 0, img.width, 0, 400)
  let ymax = map(box.ymax, 0, img.height, 0, 400)
  translate(xmin, ymin)
  stroke('white')
  strokeWeight(2)
  noFill()
  rect(0, 0, (xmax-xmin), (ymax-ymin))
}
```

Notes

We had to get the dimensions of the image on the canvas and map it to the real image size, which was considerably bigger. To see the raw results, uncomment the console.log.

Figure C4.1





Sketch C4.2 adding the label

We can also create a label for the image that it has bounded. After all, it found an image of something it recognised.

```
let results
let img

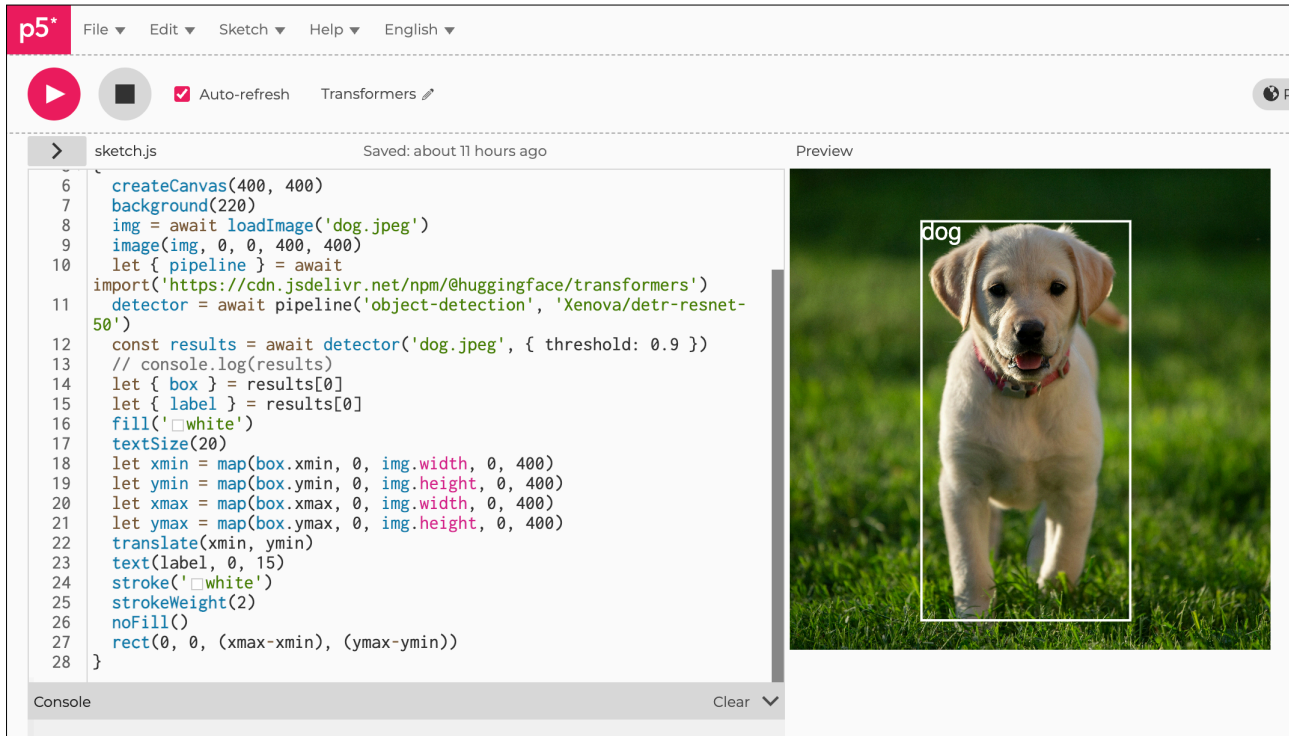
async function setup()
{
  createCanvas(400, 400)
  background(220)
  img = await loadImage('dog.jpeg')
  image(img, 0, 0, 400, 400)
  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')
  detector = await pipeline('object-detection', 'Xenova/detr-resnet-50')
  const results = await detector('dog.jpeg', { threshold: 0.9 })
  // console.log(results)
  let { box } = results[0]
  let { label } = results[0]
  fill('white')
  textSize(20)

  let xmin = map(box.xmin, 0, img.width, 0, 400)
  let ymin = map(box.ymin, 0, img.height, 0, 400)
  let xmax = map(box.xmax, 0, img.width, 0, 400)
  let ymax = map(box.ymax, 0, img.height, 0, 400)
  translate(xmin, ymin)
  text(label, 0, 15)
  stroke('white')
  strokeWeight(2)
  noFill()
  rect(0, 0, (xmax-xmin), (ymax-ymin))
}
```

Notes

The label is useful to identify the object and if it is being used for training another model.

Figure C4.2





Sketch C4.3 multiple objects

Uploaded a new image with many objects, ran a `for()` loop to extract the data, although you have to be careful not to have too many objects, otherwise it will be cluttered. This is an example that pulls the dimensions of the image and resizes it on the canvas.

```
let results
let img

async function setup()
{
  createCanvas(600, 600)
  // background(220)
  img = await loadImage('cup.jpeg')
  image(img, 0, 0, img.width/4, img.height/4)

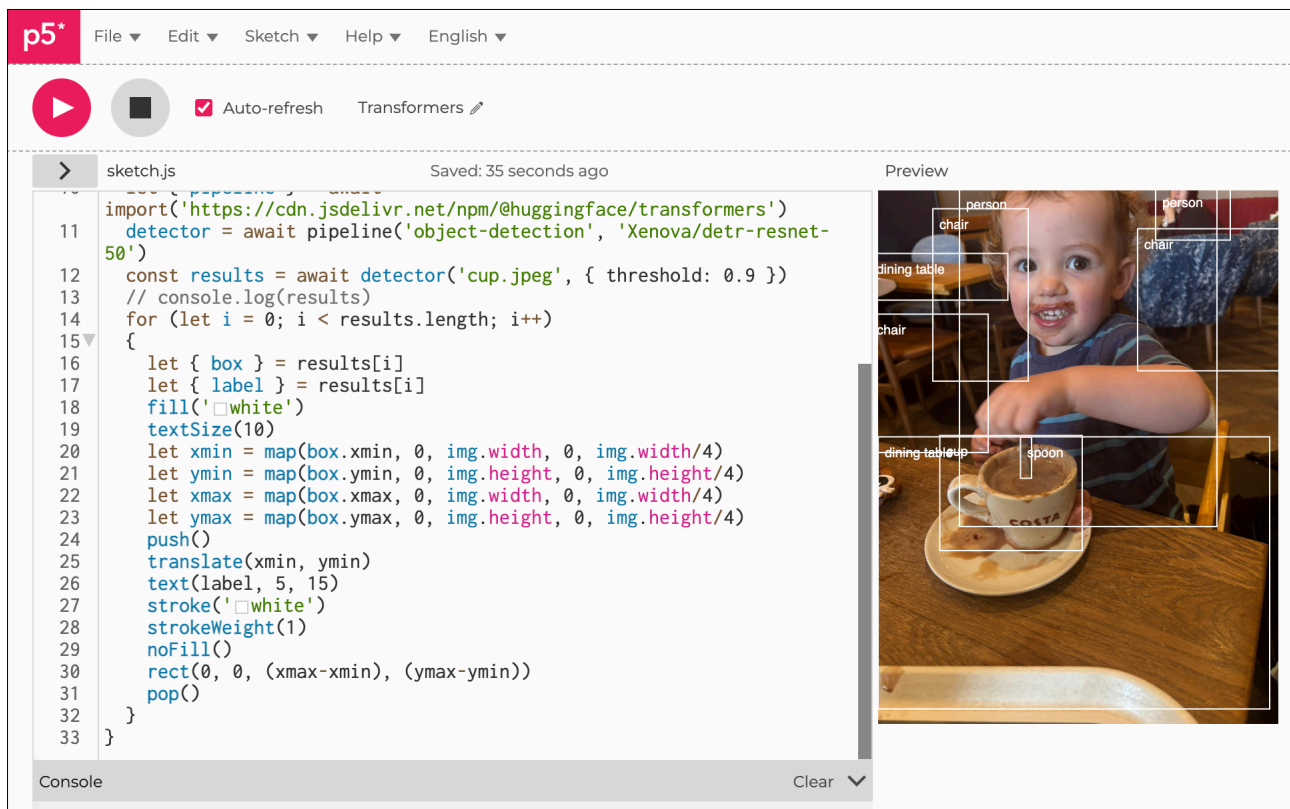
  let { pipeline } = await import('https://cdn.jsdelivr.net/npm/@huggingface/transformers')
  detector = await pipeline('object-detection', 'Xenova/detr-resnet-50')
  const results = await detector('cup.jpeg', { threshold: 0.9 })
  // console.log(results)
  for (let i = 0; i < results.length; i++)
  {
    let { box } = results[i]
    let { label } = results[i]
    fill('white')
    textSize(10)
    let xmin = map(box.xmin, 0, img.width, 0, img.width/4)
    let ymin = map(box.ymin, 0, img.height, 0, img.height/4)
    let xmax = map(box.xmax, 0, img.width, 0, img.width/4)
    let ymax = map(box.ymax, 0, img.height, 0, img.height/4)
    push()
    translate(xmin, ymin)
    text(label, 5, 15)
    stroke('white')
    strokeWeight(1)
    noFill()
    rect(0, 0, (xmax-xmin), (ymax-ymin))
    pop()
  }
}
```

```
}
```

Notes

Resizing the image to fit on the canvas is a bit manual. I suspect it doesn't really matter if all you want is what objects are in an image (or video).

Figure C4.3





Quantisation

The weights are the stored parameters that a trained model uses when running the model for real. They can be stored as floating points, 32-bit, or 16-bit, for example. The more floating-point bit size, the more accurate it could be, but at the cost of being more computationally expensive.

This model works in your browser, so it needs to be as slimmed down as possible for it to work quickly and efficiently. One way is to have the weights stored as integers, and that is called quantisation.

In some models, you can specify this with such options as fp32 or q4, for instance. This is what these values mean. By default, the model may store weights as 32-bit floating points. The weird thing is that by reducing the number of floating-point decimal places, you don't lose much accuracy and you gain a faster result.