More Data, More Sheets API

How the Google Sheets API uses magic (and code) to fill your Sheet with visual data.

Made Lapuerta

Aug 12, 2019 · 6 min read ★

As a huge fan of Yo-Yo Ma, I've been playing around with the Google Vision API by passing through an image of a cello. After obtaining the

API's label detection results, I become overjoyed when words like "String Instrument" output on my screen. How neat that a computer can identify and enjoy an image of a cello as much as I do.

However, what exactly is the *point* of sending image through the Vision API? I can already see what is in it, so why do I need a computer to reiterate what I already know?

Like for most of life's questions, the answer is Google Sheets.

Spreadsheets are a great tool to organize and visualize data, and (buzzword!) machine learning is an excellent way to obtain such data. Machine learning — specifically the Vision API — becomes wonderfully useful when its returned data is collected, organized, and visually presented through Sheets.

"But using Sheets can be time consuming and confusing!"

You are right, it most certainly can be, which is why I've been using the Google Sheets API and Cloud Functions to do all of the dirty work for me.

Getting Data.

Making a call to the Google Vision API is as simple as a couple of lines of code.

```
const client = new
vision.ImageAnnotatorClient();
const [result] = await
client.labelDetection({image:
    {content: fileBytes}
  });
```

Specifically, I am performing label detection on an image, defined by the Vision API as a way to "detect and extract information about entities in an image, across a broad group of categories." Since my cello image is local, I am sending its image bytes through the API as a base64-encoded string.

You can take a closer look at the response returned from label detection, but here, I am only interested in the label's description and its score.

```
var labelsResult =
result.labelAnnotations;
var labels = labelsResult.map(label
=> label.description);
var labelScores =
labelsResult.map(label =>
label.score);
```

In these two variables, **labels** and **labelScores**, I hold every label returned by the Vision API as well as its corresponding score. Having this awesome data, however, is not enough. In fact, data is practically useless if you do not know what to do with it, or how to understand it.

If you're unsure how to proceed, here's a viable option: send it all to Sheets.

Preparing our Data.

The label detection specifications explain that our **label** and **labelScores** variables will simply be

JSON maps. What if, in my Sheet, I want every row to contain one label and its corresponding score? The best option here is to split this JSON map into a 2D array which mirrors how we want our data to appear in a Sheet.

First, we use JSON.stringify to turn our JSON data into one big string. Next, we remove any unwanted characters and, lastly, we turn this big string into an array by indicating a new index every time we reach a comma. (This is because that the labels in our returned JSON map are each separated by a comma).

```
var labelData =
JSON.stringify(labels, null, 2);
labelData = labelData.replace(/[\
[\]'\"]+/g,'');
labelData = labelData.split(',');
```

I can perform these same exact lines of code on my **labelScores** JSON response to obtain another array containing all of the scores. I choose to be

unoriginal and name this second array **scoreData**.

Writing to a Sheet.

In order to send my two arrays to a Sheet, I need to ensure they abide by the Sheets API's call parameters. As previously mentioned, the array of values I want to send to a Sheet needs to mirror the way I want my data formatted in the Sheet itself.

I know that my two arrays have the same length, since for each label there is a corresponding score, and vice versa.

```
var dataLength = labelData.length;
```

I also know that, for every row, I want one label in column A and its corresponding score in column B. So, that is exactly what my **values** array needs to look like.

```
var values = []
  for (var i = 0; i < dataLength;
i++) {
    var dataToPush = [labelData[i],
scoreData[i]];
    values.push(dataToPush);
  }
```

The method I need to call in order to push these **values** to my Sheet is…

```
sheet.spreadsheets.values.update(requ
est, {
```

… where my **request** variable is as follows:

```
var body = {
 values: values
}

var request = {
    spreadsheetId: 'YOUR_SHEET_ID',
    valueInputOption: 'USER_ENTERED',
    range: 'A1:B',
    resource: body,
    auth: // YOUR OAUTH CLIENT
};
```

Specifying `spreadsheetId` is crucial whenever making a Sheets API call, in order to ensure you are altering exactly which Sheet you're meant to. Here's how you can find yours.

Credentials and data aside, perhaps the most important value in this request variable is **valueInputOption**, where you get to specify *how* you want your values to be inputted into the sheet. `'USER_ENTERED'` is your best bet, making sure that the data will appear in your Sheet *exactly as if you had taken the time to input it yourself*.

Additionally, the **range** value (given in A1 notation) is essential to ensure your data is formatted into your Sheet as specified. A1 notation 'A1:B' means *start writing at cell A1*, *move over to column B*, then *continue moving down rows and switching from column A to B until all of your data is written*. So, even though you've set up your **values** array as *how* you'd like your data to be

displayed, you need to correctly set the **range** variable to *where* you'd like it displayed.

	A	B
1	String instrument	0.9929680228
2	Musical instrument	0.9884474277
3	String instrument	0.983848393

Fragment of what your inputted labels and label scores should look like in your Sheet. I hope you like cellos!

Generating Visual Data.

If you take a peek inside of your Spreadsheet now, you'll see all of the data that you've obtained from the Vision API's label detection. Again — what's data if you don't know how to understand it?

Your next step here is to remember that everybody loves graphs! Let's use GCF to make one.

The Sheets API call to create a column chart from existing data is a batchUpdate request, which closely resembles our previously-made values.update call. The key difference here is the request parameter we send in to our batchUpdate:

an uncomfortably long JSON body where domain, series, axis titles, and position are specified.

Let's take a look at some of the specifics of our JSON body, which we'll store in a variable named **chartRequest**:

```
    chartType: "COLUMN",
              legendPosition:
"BOTTOM_LEGEND",
              axis: [
                {
                  position:
"BOTTOM_AXIS",
                  title: "Vision
Labels"
                },
                {
                  position:
"LEFT_AXIS",
                  title: "Score"
                }
              ],
```

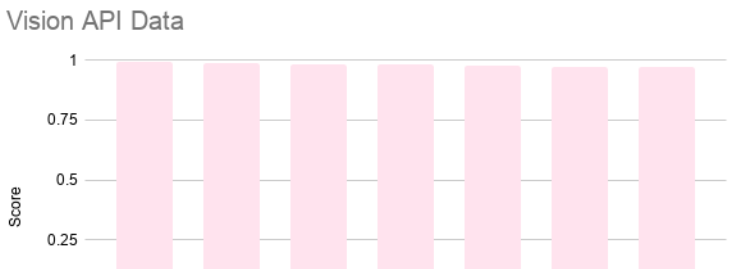
The above is pretty self explanatory with the data we have: we are going to create a column chart where every column is a label and its height is

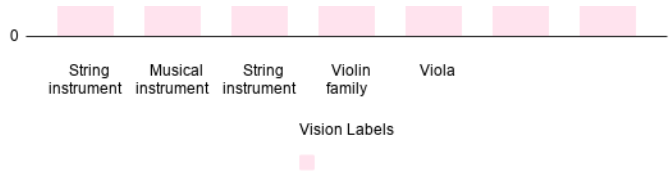
determined by its score. Again, this is just a small fragment of what our **chartRequest** needs to look like, but understanding these specifics of the request body will allow you to create a chart for whatever future data layouts you may have.

Long JSON body aside, the batchUpdate request itself is quite simple…

```
sheet.spreadsheets.batchUpdate({
      spreadsheetId,
      resource: chartRequest,
});
```

That's all you need: your spreadsheet's ID (where to find the data) and your chartRequest (how to create a chart from the data). See? The Sheets API is just like magic.





The column chart you just generated from the Vision API data. Personally, I like to set all my columns to #ffe3ee.

Resolving the Existential Crisis.

Okay, we get it, you can use the Sheets API to do some cool things with data & graphs — but what's the point?

Great question. The *point* is that we are living in the era of big data, meaning computers and software are allowing us to access more data than ever before. Hello better consumer analytics, sales reports, and market research! Yet, quick and constant improvements in technology are making it tough for companies to keep up, and data often ends up being lost, misunderstood, or un-analyzed.

With these simple Sheets API calls, data can be easily stored, analyzed, and visualized, without having to actually navigate through a Google

Sheet itself. By handling these implementations on the back-end, it is no longer confusing, tedious, or time-consuming to generate data visualizations. Plus, Google Sheets keeps your data easily accessible & simple to share.

Big data, small problems.

. . .

You can find my full GCF code on my GitHub.

A special thanks to my internship host, Anu.

Let me know how you're using GCF and the Sheets API to play around with data, I'd love to hear from you!

Thanks to Anu Srivastava.

JavaScript Google Sheets Machine Learning

Data Analytics Big Data