Machine Learning Web App: Student Guide

# Overview

There are normally two computers that run code when a web application is used: the client and the server. The ***client*** is the computer that the user owns, and the ***server***is the computer that the service provider owns.

When you run a web app, the client sends requests to the server, which responds to these requests. For example:





In our application, there will be *three* computers involved:

1. The **Client**, owned by the person accessing the web app
2. The **Web App Server**, owned by the person running the web app
3. The **Google Cloud server**, owned by Google

The client will make a request, and send it to the web app server. To determine the response to send back to the client, the web app server will make its own request to Google’s server to use Google’s Machine Learning APIs. For example:



There are two main programs that need to be written, the *front-end* and the *back-end*.

The ***front-end*** is the code that the client runs in order to interact with the user and send requests to the server. This usually includes HTML to create a web page, and JavaScript to configure its behaviour.

The ***back-end*** is the code that the web app server runs in order to respond to requests made by the client. In our application, this entails calling Google’s APIs and then interpreting the results it returns.

# Front-End

The front-end code is all kept in a folder named *front-end*. When the user first goes to the web app, each file in this folder is requested and sent to them.

The HTML file constructs a simple interface: an input field where users can type in an image URL, checkboxes where users can choose what type of image detection they’d like to perform, and a Go button users can click to submit their request to the server.

The JavaScript file adds the corresponding behavior to the interface so that it can compile requests, send them to the server, and update the interface appropriately when it gets the response.

The CSS file controls how the page is displayed by web browsers, both in terms of style and layout.

Remember, a good user interface should be:

* **Clear**; it should always be clear to the user what the current state of the application is, and it should always be clear what the effect will be of taking any action.
* **Efficient**; it should allow the user to complete their objective quickly and easily.
* **Flexible**; it should work whether the user is using a brand-new smartphone or an ancient desktop computer with a slow connection.
* **Error-tolerant**; if the user makes a mistake, it should provide useful feedback.

You have two options: you can improve the existing interface for the app as it is, or you can use the code as a basis for creating an interface for a more specific app that you have chosen.

**Improve**

If you have got only limited experience with web development, you might choose to take the given interface and improve it. The most important improvements are normally related to ease of use, rather than how good it looks. Below are 4 suggested ways of improving the interface. Some are easier than others; it’s your choice which to complete.

**1. The Go Button**

To make the interface look better, and to make it easier for users (especially mobile users) to select it, the size and prominence of the Go button could be improved.

1. Currently the button is the browser default - there is no style information associated with it. Open the CSS file in a text editor, which allows you to change the style of different elements.
2. We need to add a *selector*, which matches a certain set of elements to allow you to change their style. There are two possible selector which we could use: #go and input[type=”button”]. The first specifies the id of the Go button specifically, whereas the second matches any type of button. Decide whether you’d like to change just the Go button or the style of all buttons on the page (e.g., others that might be added later).
3. Add a section to the CSS file, starting with your chosen selector, and specifying different style properties:

|  |
| --- |
| YOUR\_SELECTOR\_HERE {  background-color: #4CAF50;  border: solid;  border-color: #FF0D0D;  color: #FFFFFF;  padding: 15px 32px;  font-size: 18px;  } |

1. Edit the properties to improve the design of the button. Experiment and see what works best.
   * **background-color** can be changed, either by picking a different colour from an online colour selector, or by using one of the “highlight colours” defined at the top of the file. For example, you could use background-color: var(--highlight-1);
   * **border** specifies whether the button has a border, and the type of that border (solid or dashed, for example).
   * **border-color** specifies the border colour.
   * **color** specifies the colour of the text.
   * **padding** specifies how much space there is between the button text and the button border.
   * **font-size** specifies how large the font is.
2. Make sure that the file is saved; then redeploy your app to see how it looks.

**2. No Feature Feedback**

1. Try inputting a URL to the text box, but selecting no features. What happens when the Go button is pressed?
2. The problem is that an error is returned if no features are requested. It would be better if instead of passing an empty request to the server, we instead provided feedback to the user. We can change this behaviour by adding a data-validation check to the JavaScript file.
3. Examine the code that runs when the Go button is clicked. The first thing that happens is that a list is compiled of the selected features. Then, a request is sent to the server of these features. We want this request to be sent only if more than 0 features were requested.

Add in a conditional if-else statement to the JavaScript program. It should check whether more than 0 features were requested, and make sure that the next (big) section of code is only run if they were. Otherwise, you could use the *result* section of the page to display some feedback to the user.

|  |
| --- |
| $(“#result”).html(“Error: you need to select at least one feature”); |

**3. No URL Feedback**

Likewise, if the URL box is empty, the request should not be sent, and useful feedback should be displayed to the user. Using the same method as above, introduce another validation check into the JavaScript code to make sure the URL box is not empty. (You might want to use the “else if” construct.)

**4. One-Click “Select All”**

If the user wants to select many features, they currently have to click to select each individually. It would save time if there was a button they could press to select all the features at once.

1. Add a new button to the interface. Give it a unique id, so that it can be specifically selected by CSS and JavaScript if necessary.
2. Add a function to JavaScript that should be run when the button is clicked.

|  |
| --- |
| $("#YOUR-BUTTON-ID").click(function() {  // Your code here  }); |

1. This function should use a selector to select each of the feature input elements, and should check each box (if they are checkboxes) or add the *chosen* class (if they are buttons because you completed Buttons Not Boxes above).

**Repurpose/Redesign**

You might choose to repurpose the given interface by removing any parts which are not needed for your application, and modifying any parts that are needed so that they best suit how the user will interact with the application. If you are more experienced in web development, you might even choose to entirely redesign the interface, using the example as a guide but writing your code from scratch.

Below is a rough guide showing how you might approach this, with the example of an application in which the user guesses the labels for an image.

|  |  |  |
| --- | --- | --- |
| **Question** | **What To Do** | **Example** |
| Is the set of features that needs to be requested fixed? | If the application always needs a fixed set of features to be requested, then changes need to be made:   1. The checkboxes allowing the user to select which features to request should be removed 2. Rather than the JavaScript file compiling a list of features in the variable selected\_feats, which is then included in the request, this list should be hardcoded to always request the same features. | In the example application, the user needs to guess labels for an image. Therefore, only the label detection feature needs to be requested. (Or you might also choose to request landmark or logo detection, with the results to be accepted as “labels” too).  Therefore, selected\_feats should always be set to ["LOGO\_DETECTION"]. |
| What does the user need to input? | If the application requires the user to input anything other than the image URI, then the interface needs to provide a way of doing this:   1. The HTML needs to be added to include suitable input elements. [Online references](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) are available, but these might include text boxes, radio buttons, drop-down menus, and so on. 2. The JavaScript needs to be edited to make sure that this extra information is included in the request, if it is needed by the back-end for processing.    1. Use a jQuery selector to gather the values of the inputs when the Go button is pressed.    2. Include this information in the request data object, if the back-end needs it. | In the example application, the user needs to be able to input their label guesses. It makes sense to do this using a text box, so this would have to be added to the interface.  Then, the decision needs to be made as to whether the guess is verified on the back-end by the Python program running on the server, or verified on the front-end by the JavaScript program running on the client.  If the Python program will process the guess, then this information must be included in the request data object. This could be done in one line by adding the following property to that object:  guess: $("#user\_guess).val() |
| What processing does the client need to do? | Any processing could be done on the front-end or the back-end. If any processing is to be done on the front-end, JavaScript code needs to be added to achieve this. Typically, the results from the server response would be manipulated to get information in the form that is ready to be shown to the user. | If the guess verification was done using the front-end rather than the back-end, a comparison could be made between the user guess and the returned labels:  guess = $("#user\_guess).val()  if (results.includes(guess)){  // Correct guess  } else {  // Incorrect guess  } |
| How will the results be displayed? | In the given example, the results are split into different boxes, which are displayed in a grid. You should think carefully about the best way to display the results to your user. | The result can be shown by displaying one of two different icons next to the user’s guess, depending on whether it was correct or not. |

# Back-End

The template back-end code (the code that the web app server runs) is contained in *back-end.py*. When the client sends a request, this code processes the request and sends a response back to the client. In our application, the back-end logic involves making a request to a Google API and parsing (processing) the response.

There are three main parts to the program:

1. A function that can be used to make a request to the Google Vision API.
2. A function that takes the response from the Google API and turns it into some HTML content that can be displayed to the user.
3. The app initialisation, including instructions to use these two functions to process requests.

You have two options: you can improve the existing features provided by the back-end, or you can use the code as a basis for implementing new features that may be required for your app.

**Improve**

If you have got only limited experience with web development, you might choose to take the given code and try to improve it. Below are 6 suggested ways of improving the program. Some are easier than others, so it is your choice which to complete.

**1. Only One Logo**

An image of a logo contains only one logo: the logo pictured! However, the logo detection results can sometimes return a list—for example, if the machine learning model isn’t 100% sure about which logo it is. Rather than displaying all the results in this list, it might make more sense just to suggest the top (most probable) logo result.

1. Find the function which parses the logo annotations and produces an HTML string to display to the user.
2. annotations is a list of Python dictionaries, each representing one possible logo result. We only want the first of these dictionaries, since it is the most probable, so let’s write a line of code storing this one result in a variable.

|  |
| --- |
| top\_result = annotations[0] |

1. Then, the string associated with "description" in this dictionary contains the name of the suggested logo. Write a line of code to retrieve this value.

|  |
| --- |
| top\_logo\_name = top\_result["description"] |

1. Then, rather than returning the list of logo results, return just this one result. You might want to include a prefix such as “Suggested logo result: ”.

**2. Only One Landmark**

In the same way, an image of a landmark usually only contains one landmark! So again, it might be better to just return the top suggested landmark result, rather than including other landmarks which might look only slightly similar. Implement this improvement as discussed earlier for logos.

**3. Display New Lines Correctly**

When the Google API returns the text detected in an image, it represents new lines in the normal way, using the "\n" character. However, when a web browser displays HTML, it ignores any "\n" characters because new lines are represented by the <br> tag in HTML. It could improve the formatting of the text results if the "\n" characters were converted to <br> tags.

1. Locate the section of code and variable that contains the text results from the Google API.
2. Before returning this text in the response to the client, change the "\n" characters to <br> tags. You might want to use the following function:

|  |
| --- |
| # Returns a copy of my\_string, with the newlines converted  my\_string.replace('\n', '<br>') |

**4. Only Relevant Labels**

The Vision API can return many labels associated with one image. However, some of these may be dubious. Rather than showing the user a very long list of possible labels, it may be preferable to list only those which have a high confidence rating. Luckily, the API returns a confidence rating for each suggested label, so this can be used to filter the labels.

1. Find the function that converts the labels annotations object to an HTML string to display to the user.
2. As mentioned earlier, the annotations object is a list of dictionaries, with each dictionary in this case representing a label suggestion. The example code simply prints out the description (name) of each label. However each dictionary also contains a "score": a number between 0 and 1 representing the Vision API’s confidence in the label. For example, a score of 0.743 means the system is 74.3% confident that the label is correct.

Add some code that removes all the labels below a certain score threshold before they are returned.

|  |
| --- |
| filtered\_annotations = []  for anno in annotations:  if anno['score'] > 0.6:  filtered\_annotations += [anno]  # Then print the labels in filtered\_annotations only |

**5. Only Relevant Web Results**

Likewise, each [WebEntity object](https://cloud.google.com/vision/docs/reference/rest/v1/images/annotate#WebEntity) contains a score as well as a description. However, these are not normalised (not in the range between 0 and 1) and are not comparable between different images. In other words, a result with a score of 8.1 for one image may be highly relevant, while another result with the same score for a different image may be less relevant.

1. Consider how to filter the results using the score so that only the most relevant results are shown. For example, you could show only the top 4 results, or you could show only results whose scores are close enough to the score of the top result.
2. Consider returning the confidence scores in the results, so that the user can see how they compare.

**Repurpose/Redesign**

You might choose to repurpose the given program so that it provides functionality specific to your app, by removing any irrelevant parts and adding extra processing so that the response to the client contains information the user is expecting. If you are particularly confident, you might choose to redesign your own back-end program from the beginning, using the program provided as an example to get started.

Below is a rough guide showing how you might approach this, for a sample application that gives as much information as possible about the faces in an image.

|  |  |  |
| --- | --- | --- |
| Question | What To Do | Example |
| What form will the request from the client take? | Consider the information included in the request from the client, and use it to perform any processing needed. In the web app class which serves the requests, make sure to extract any parameters given by the user and pass them to the relevant functions. | The face analysis application does not have any additional information passed by the client other than the image URL. |
| What requests will be made to Google APIs? | Some applications will always request the same features from the same Google API. In this case, consider removing the code which allows different requests to be made, and consider removing the code that handles responses from features that are not being used. | The face-analysis application is likely to only use the face detection feature, so the features requested in make\_api\_call can be hardcoded to be just FACE\_DETECTION. All methods related to handling responses for other features can also be removed. |
| What results processing does the back-end need to do? | Any results processing could either be done on the front-end or the back-end. If the processing is to happen on the back end, code needs to be added to process the API responses along with any information in the client’s request to compute a response to send back to the client. | If the back-end is to do the processing for the face-analysis app, then this needs to be done after the response is received from the API and before the response is sent to the client. |

# Invoking the Google Cloud Vision APIs

To access Google’s Cloud APIs directly, you need to set up a [Google Cloud Platform account](https://cloud.google.com/). However, to get you up and running quickly, we’ve set up a special “wrapper API” that you can call to invoke the image-detection functionality available in the Google Cloud Vision API. The wrapper supports the following functions, each of which take a single argument: a URL of an image on the Web.

* detect\_faces
* detect\_labels
* detect\_landmarks
* detect\_logos
* detect\_text
* detect\_properties
* detect\_web
* detect\_crop\_hints
* detect\_document

Any of the above functions can be invoked on the wrapper as follows:

response = service.ml-vision-wrapper()  
 .*detect\_faces*(request=*url*)  
 .execute()

Where:

* The **service** object refers to the API: its version and URL
* **ml-vision-wrapper** is the API’s name, as defined in its class definition
* ***detect\_faces*** is the function called on an image URL (the link to the image to process)
* **execute** makes the function call.

When any of the “detect” functions are invoked, the following steps are performed:

1. Confirm that a resource exists at the given URL. If so, proceed to step 2. If not, throw an exception.
2. Set up the Cloud Vision API client.
3. Read in input image at given.
4. Check for unsafe content in the image; if unsafe content is found, return without calling API detection functions.
5. Call the Cloud Vision API to detect labels using the client defined in step 2.
6. Extract labels from result object.
7. Return labels.