IBM Watson

# Classifying Images Using Node-RED

*Lab 7 Guide*

# Contents

# Section 1.Preface

## Industry Use-case

Industries have a lot of infrastructure. They range from thousands of miles of pipelines in the oil and gas industry to enormous acres of land in agriculture, not to mention the colossal transportation structures, such as roads, bridges and railway tracks. These structures, along with multitude of industrial equipment, need to be inspected and maintained so that these industries can continue to operate. Corporations employ various methods to do this, such as routine visual inspections, running tests and digging deeper. However, with the huge number of structures and equipment to keep track, many inefficiencies exist.

Visual inspection can prevent disaster and maintain seamless operations. In most cases a quick look can identify whether there is a defect, and with some experience we can identify what the defect is. With the Watson Visual Recognition service, we can determine whether the equipment in the image meets normal conditions or by training the service to identify particular defects and damage.

In our code pattern, Industrial Visual Analysis, we'll train the Watson Visual Recognition service to inspect oil and gas pipelines to classify the image into categories: Normal, Burst, Corrosion, Damaged Coating, Joint Failure and Leak. The training is performed by providing the service with a set of images for each category, by defining a particular classifier. This classifier can then analyze the image and provide a percent match to each category.

https://developer.ibm.com/code/2017/11/14/industrial-use-case-watson-visual-recognition

## Overview

The Visual Recognition service can be used for diverse applications and industries, such as:

- **Manufacturing:** Use images from a manufacturing setting to make sure that products are positioned correctly on an assembly line
- **Visual auditing:** Look for visual compliance or deterioration in a fleet of trucks, planes, or windmills out in the field, train custom models to understand what defects look like
- **Insurance:** Rapidly process claims by using images to classify claims into different categories
- **Social listening:** Use images from your product line or your logo to track buzz about your company on social media
- **Social commerce:** Use an image of a plated dish to find out which restaurant serves it and find reviews, use a travel photo to find vacation suggestions based on similar experiences
- **Retail:** Take a photo of a favorite outfit to find stores with those clothes in stock or on sale, use a travel image to find retail suggestions in that area
- **Education:** Create image-based applications to educate about taxonomies

https://www.ibm.com/watson/webinars/#

# Objectives

This lab is an adaptation of the Node-Ren flow developed by John Walicki, STSM, CTO IoT Developer Advocacy, IBM Cognitive Applications.

Here's how you will implement your image classifier in Node-RED:

You must first provision a Node-RED if you have not already done so in prior labs.

You will next import the Node-RED flow provided here.

You will note that you are missing the zip node. You must install that from the Manage Palette menu item, for it is an opensource node contributed by the community.

Next, you must connect your node-RED app with an existing or new Visual Recognition service. You are no ready to deploy the app and test the visual recognition classifier or build your own classifiers using positive and negative images.

The link below elaborates on best practices as you begin to classify and train Watson Visual Recognition on your own images. the second link, is a useful demo that helps with better understanding how the service works

https://www.ibm.com/cloud/blog/watson-visual-recognition-training-best-practices

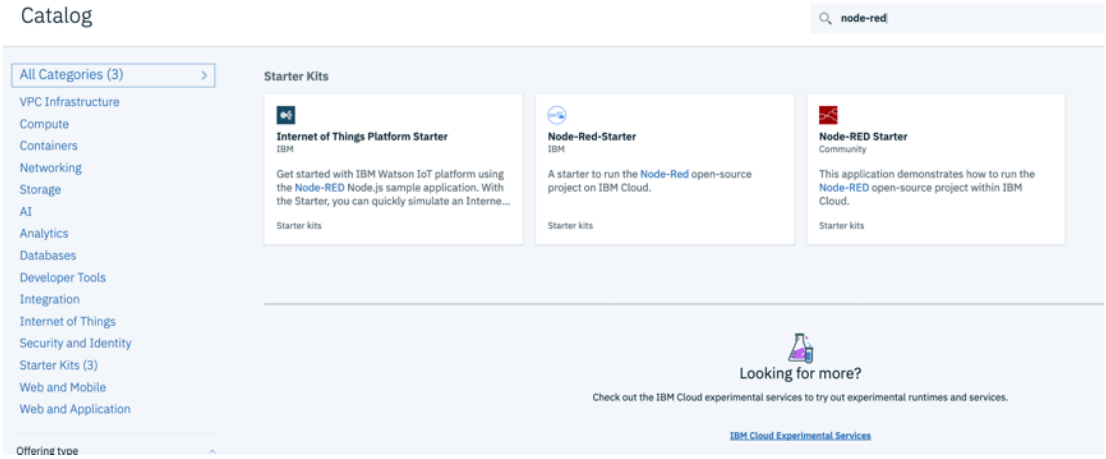
https://www.ibm.com/watson/services/visual-recognition/demo/

*Approximate time to complete: 1 hour (without the Optional lab towards the end)*

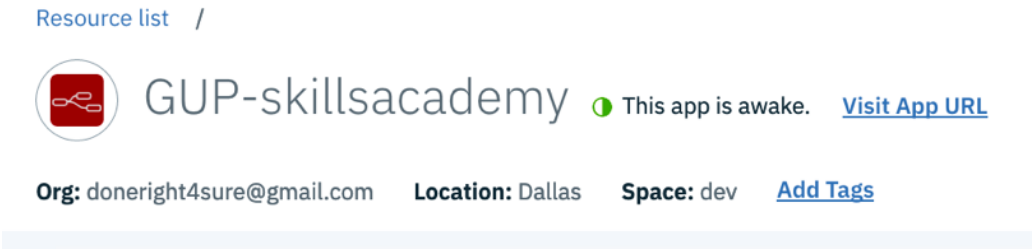
**Section 2. Build a Node-RED Starter Boilerplate**

This lab assumes that you have an IBM Cloud account, you have signed in and applied the promocode. If you have not yet done this, please refer to Lab 1 for instructions on these Pre-requisites.

1. Sign into IBM Cloud: https://cloud.ibm.com
2. From the IBM Cloud console, access the **Catalog** tab and search for **Node-RED Starter**.



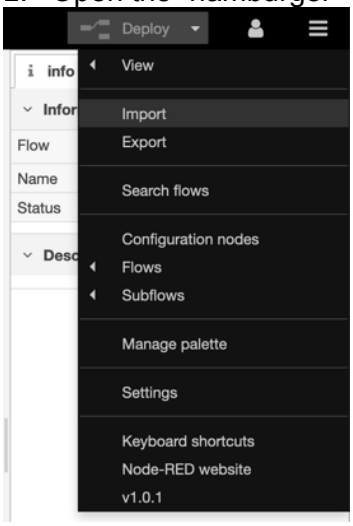

3. Specify a unique name for your app and click **Create**. Allow enough time for the app to stage and start. This may take a few minutes.
4. Click the URL in top left, in this example, it is *https://gup-skillsacademy.mybluemix.net/*



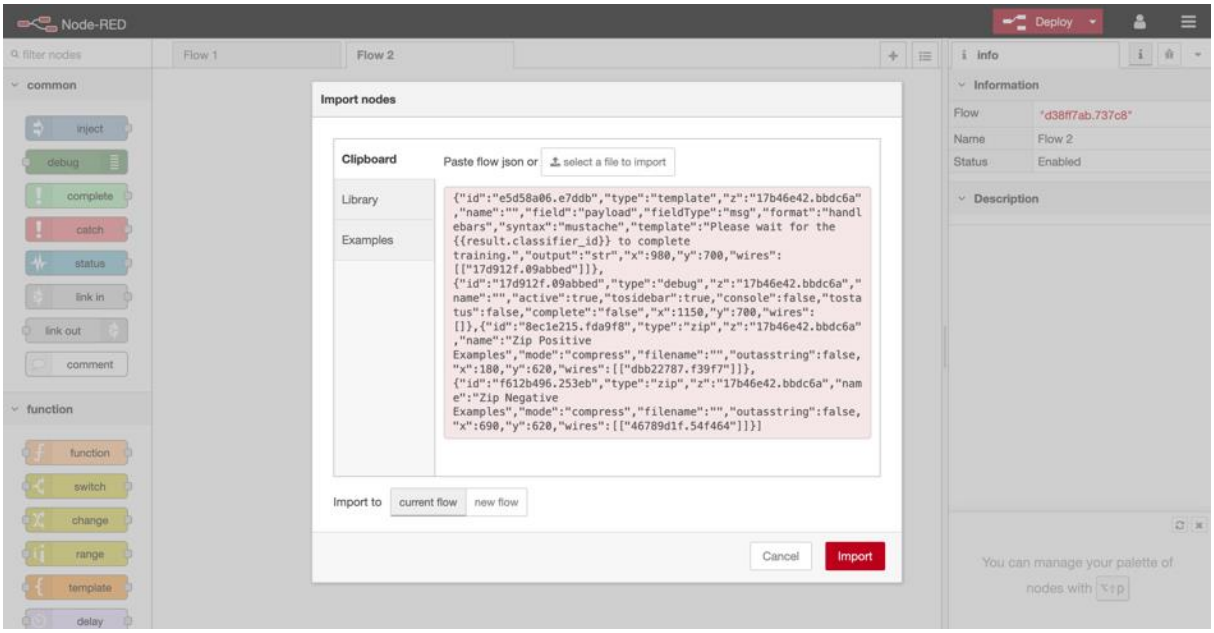

5. In the ensuing page, advance through the wizard (give it a username/password).
6. Click **Next** without selecting any of the provided starters; and click **Finish**.
7. In the ensuing page, click **Go to your Node-Red flow editor**.
8. Login with the username/password you created earlier in Step 5.

The remaining steps pertain to populating your node-RED canvas.

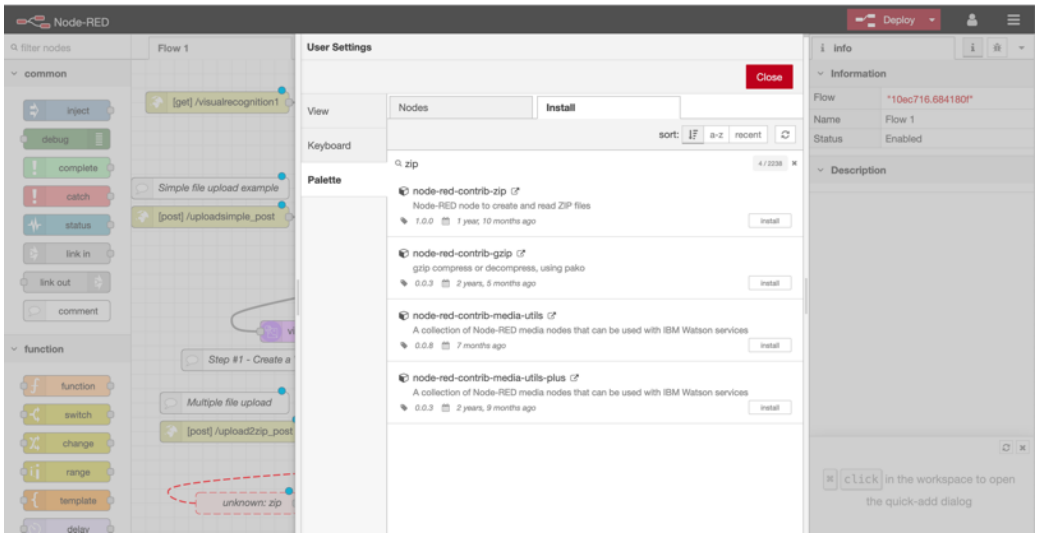
1. Open the “hamburger” drop-down menu from the top right and select Import.



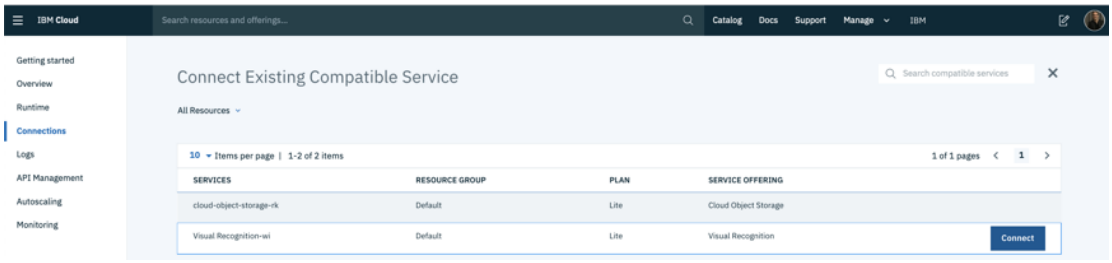

2. Copy the JSON flow from here and paste it onto the JSON editor and click **Import**



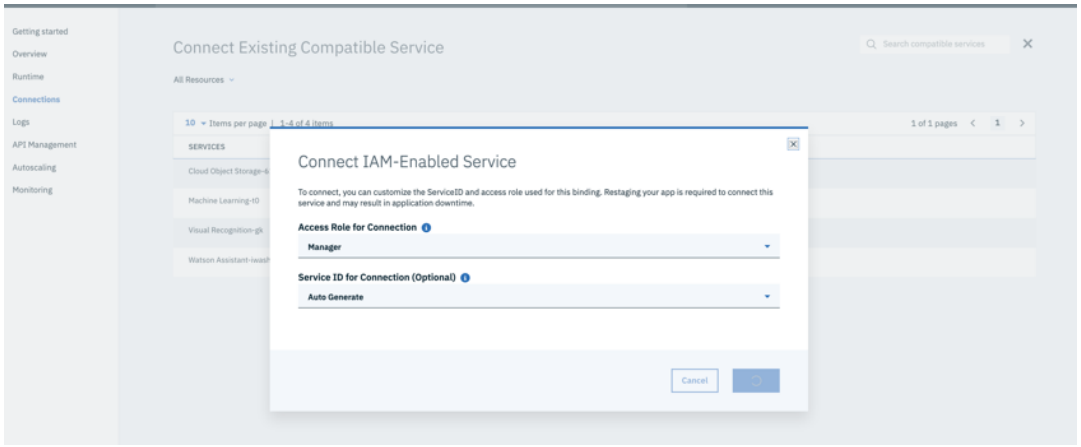

3. You will notice that you are missing the zip node.
4. Again, open the hamburger drop-down menu and click **Manage Palette**.
5. Click the **Install** tab and search for **zip**.
6. Install the first result: **node-red-contrib-zip**.




7. Click Deploy to ensure that the flow is saved.
   You must now connect your app to the Visual Recognition service. If you are doing the labs in sequence, you most likely have an existing visual recognition service already provisioned; if not, create a new one.
   This lab assumes that you have already created the visual recognition service from Lab 6.




8. Refer to the IBM Cloud web page (likely the open tab to the left) and access your Node-RED application.
9. Click the **Connections** link.
10. Click Connect. (if you are starting from this lab, you may need to provision a new Visual Recognition service. You can do so by clicking New, instead of Existing services).
11. Click **Restage**. This may take a few minutes.




12. Click **Deploy**.

To run the web page, copy and paste the entire URL as it appears on your Node-RED web page onto another tab in your browser.

1. In the URL, backspace all the way until the .net/
2. Append the context name that you specified in the HTTP In node as a GET function, in this example: *https://gup-skillsacademy.mybluemix.net/nicelydone*
3. Press Enter and you should see the web page.
4. From Google Images, Search for free images of anything you'd like.
5. Drag and drop the image (ensure it comes across as jpg or png) upon your desktop.
6. Browse from your app (the last line in your app) and click **Analyze image**.
7. Alternatively, you can upload an image URL. Just ensure it ends with .jpg or .png.

> ***INFO*** **Retrieving Image URL**
>
> Click **View image** to obtain the exact link to that image.

8. Copy the address location that ends with jpg or png image file onto the **Image URL** box of your app.

> ***INFO*** **Information Displayed**
>
> Notice, that the image classification app renders the analyzed image with respect to built-in Classifications.

# Appendix

The following guidelines pertain to your own use cases and are taken from the docs section that appears in Visual Recognition API documentation.

## Guidelines for training classifiers

You can classify your own data or create your own custom classifier.

### General classifier categories

The General classifier returns classes that are organized into categories and subcategories from thousands of possible tags. The following list shows the top-level categories:

- Animals (including birds, reptiles, amphibians, etc.)
- Person and people-oriented information and activities

- Food (including cooked food and beverages)
- Plants (including trees, shrubs, aquatic plants, vegetables)
- Sports
- Nature (including many types of natural formations, geological structures)
- Transportation (land, water, air)
- And many more, including furnishings, fruits, musical instruments, tools, colors, gadgets, devices, instruments, weapons, buildings, structures and manufactured objects, clothing and garments, and flowers, among others.

### Classify response hierarchy

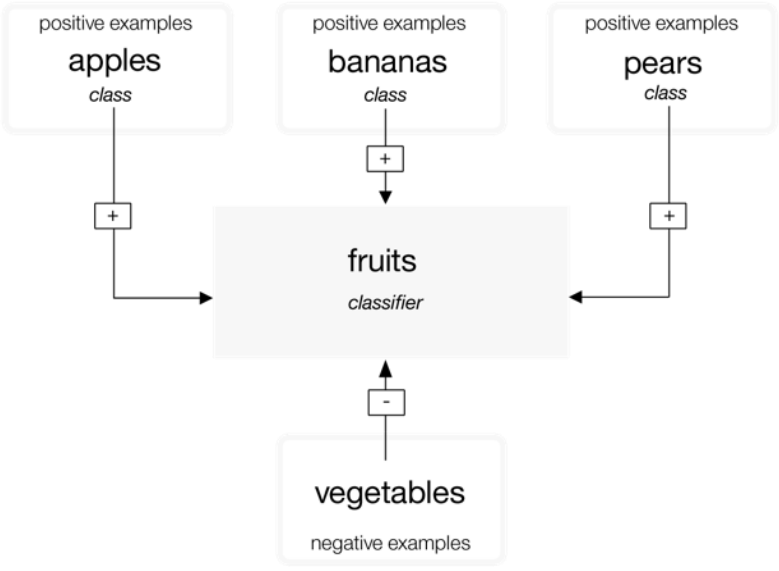
The **Classify** methods classify images within a hierarchy of related classes. For example, a picture of a beagle might be classified as "animal" and as the related "dog" and "beagle". A positive match with the related classes, in this case "dog" and "beagle", boosts the score of the parent response. In this example, the response includes all three classes: "animal", "dog", and "beagle". The score of the parent class ("animal") is boosted because it matches the related classes ("dog" and "beagle"). The parent is also a "type_hierarchy" to show that it is a parent of the hierarchy.

## Structure of the training data

A custom classifier is a group of classes that are trained against each other. Custom classifiers support multi-faceted classifiers that can identify highly specialized subjects while also providing a score for each individual class.

During training, classes are created when you upload separate compressed (.zip) files of positive examples for each class. For example, to create a classifier called "fruit", you might upload a .zip file of images of pears, a .zip file of images of apples, and a .zip file of images of bananas in a single training call.

You can also provide a .zip file of negative examples in the same training call to further hone your classifier. Negative example files are not used to create a class. For the custom classifier "fruit", you might provide a .zip file with images of various vegetables.



After training completes, when the service identifies fruit in an image, it returns the classifier "fruit" as an array that contains the classes "pears", "apples", and "bananas" with their respective confidence scores.

**Important:** The **Create a classifier** call requires that you provide at least two example .zip files: two positive examples files or one positive and one negative file.

Custom classifiers are accessible only to the specific service instance where they were created and cannot be shared with other IBM Cloud users who do not have access to your instance of the service.

## Updating custom classifiers

You can update an existing classifier by adding new classes or by adding new images to existing classes. To update the existing classifier, use several compressed (.zip) files, including files that contain positive or negative images (.jpg, or .png). You must supply at least one compressed file that contains additional positive or negative examples.

Compressed files that contain positive examples are used to create and update "classes" to affect all the classes in that classifier. The prefix that you specify for each positive example parameter is used as the class name within the new classifier. The "_positive_examples" suffix is required. There is no limit on the number of positive example files you can upload in a single call.

The compressed file that contains negative examples is not used to create a class within the created classifier, but does define what the updated classifier is not. Negative example files should contain images that do not depict the subject of any of the positive examples. You can specify only one negative example file in a single call.

### How retraining works

In the example, with three sets of positive class pictures, Apples, Bananas, and Pears, the system trains three models internally. For the Apples model, the group of pictures in "Apples" is trained as a positive example, and the group of pictures that are uploaded in "Bananas" and "Pears" are trained as negative examples. The system then knows that bananas and pears are not apples. The other classes are used as negative examples for the Bananas and Pears models as well.

You can retrain your classifier with new positive classes, for example, to recognize yellow and green pears. To retrain with these new classes, copy the original images into two new files: YellowPears.zip, with images of yellow pears, and GreenPears.zip, with images of green pears. Copy every yellow pear image from the original pears.zip training set to the YellowPears.zip file and copy every green pear image to the GreenPears.zip file.

Take care when you split a class definition by retraining. You must submit the **exact** original image files that you used during the original training in the new files. Do not resize or make other changes to the images. If the images are different, the original images are used only as negative examples for the new classes.

Retrain the system with YellowPears.zip and GreenPears.zip as positive examples. The service recognizes the duplicate images in the YellowPears and GreenPears files, and retrains with those images as positive examples for the new classes. A duplicate image is kept in the positive set if it is also found in both the negative and positive set for a class.

After you retrain, the YellowPears and GreenPears classes include Apples and Bananas as negative examples, but the duplicate images from the Pears class are not negative examples.

## Size limitations

There are size limitations for training calls and data.

- The service accepts a maximum of 10,000 images or 100 MB per .zip file
- The service requires a minimum of 10 images per .zip file.
- The service accepts a maximum of 256 MB per training call.
- Minimum recommended size of an image is 32X32 pixels.

There are also size limitations when you classify images.

- Limitations for the methods to classify images:
  - Maximum image size is 10 MB.
  - Maximum .zip file size is 100 MB with up to 20 images.

## Guidelines for good training

The following guidelines are not enforced by the API. However, the service tends to perform better when the training data adheres to them:

- Make sure that your images are at least 224 x 224 pixels.
  - If you encounter size limitations, you can resize your images to 224 x 224 pixels without compromising the quality of the training.
- For .png images, make sure that the pixel depth is set to at least 24 bits per pixel:
  - To check the depth on MacOS, run the `file` command. 24-bit depth is displayed as `8-bit/color`.
  - To check on Windows, right-click the file and choose **Properties** > **Details**. Look for **Bit depth**.
- Include at least 50 positive images per class before you assess your training results.
  - Assuming similar quality and content for your training data, more training images generally provide more accurate results than fewer images.
  - 150 - 200 images per .zip file provides the best balance between processing time and accuracy. More than 200 images increases the time and the accuracy, but with diminishing returns for the amount of time it takes.
- Include a negative class to help improve your results.
  - Include approximately the same number of negative images as positive ones. An unequal number of images might reduce the quality of the trained classifier.
- Make sure that the backgrounds in your training images are comparable to what you expect to classify. The accuracy of your classifier can be affected by the kinds of images you provide to train it.
  - For example, if you are training the "tiger" classifier, your classifier might be less accurate if you train only on images of tigers in a zoo that are taken by a mobile phone but analyze images that taken in the wild taken by professional photographers.
- Make sure that the subject matter of the classifier is at least one third of the image's overall size.

For more information about training, see Best practices for custom classifiers.

## Guidelines for high volume classifying

Maximize efficiency and performance of the service in the following ways when you submit many images:

- Crop or resize your images.
    - For the best performance without compromising classification quality, consider resizing your images to 224 x 224 pixels. The service is currently optimized for this size, although that might change.
    - Crop the image if it has an aspect ratio greater than 2:1 or under 1:2.
    - Consider cropping the image into multiple square images, or include only the center of the image, depending on what is most important to your use.
- Submit up to 20 images in a single .zip file. You don't need to use any compression because JPEG and PNG images are compressed files.
- Use the **classifier_ids** parameter to specify only the classifiers that you want to use.
- Although the service reads EXIF tags and rotates images, for the best throughput, send images that don't need to be rotated by the service (the EXIF **Orientation** tag is set to `1`).

## Custom classifier scores

The **Classify** methods produce a score between 0.0 and 1.0 for each image for each class. The scores identify different things for custom classifiers than for the General classifier.

### Background reading

- The service performs statistical classification.
- You can measure statistical classifiers in several ways.

### How to use the scores

- Think about possible actions to be taken in response to a classification. Specifically, analyze how you will use "true" or "false" positive or negative conditions. These conditions are described in the Background Reading.
- This cost-benefit balance is crucial to deciding what to do with each class score, and only someone who understands the final application can determine it. The score value needed for the application to take some action is called the "decision threshold." The service does not compute this for you.
- Custom classifiers use binary "one versus the rest" models to train each class against the other classes. The system assumes that two classes within a classifier cannot occur at the same time, so you should create separate classifiers to test for classes that can exist together, like `blue` and `sky`. Alternatively, you might create a distinct classifier for cases where both classes exist at the same time and test for a class like `blueSky`.

### Example

Imagine that you are monitoring an assigned parking space with a webcam. You train a custom classifier to recognize whether your car is in the spot, some other car is in the spot, the spot is empty, or the camera has been blocked. You collect training examples for each of these cases and train a custom classifier with four classes. Your application classifies images from the webcam to report the status of the spot, and the system notifies you with a message if the status is unexpected. Each time the service classifies the image from the camera, it produces four scores: `myCar`, `unknownCar`, `emptySpot`, and `blockedCamera`.

The first action to consider is whether to send a notification.

Suppose you park in your spot and have the service start classifying the images. You see the `myCar` score that is computed as 0.8 on average over a few hours, while the `unknownCar` score hovers around 0.3, `emptySpot` is around 0.15 and `blockedCamera` is around 0.1. Given this data, you write your code to notify you if the `myCar` score is less than 0.75, or if one of the others is greater than 0.6. During the day, you get about one false alarm

every three hours when people walk by and obscure the car. The system sends you the photo along with the notice, so you can see it's nothing to worry about. That seems fine, but at night those false alarms every three hours get annoying! Your preferences for day versus night notification reflect the higher cost of a false alarm at night time for your application.

So, the notification logic and threshold will probably vary, depending on the perceived risk of car theft, the accuracy of your classifiers, and the amount of annoyance caused by a false alarm.

Similarly, as a person, you might face the same tradeoff. If the system notifies you that the camera is blocked, the accompanying image will likely be all black or gray. Do you check the car or ignore it? Again, this choice depends on your other priorities and the perceived risks.

## Questions

- **What do the scores mean?**
  - The scores are comparable indicators, with a range from 0.0 to 1.0. You can compare the scores of two custom classes (from the same or different classifiers) on the same or different images, and the higher one should be more likely to appear in the image than the lower one. However, they may both be present. It is best to pick a decision threshold for each class individually.
  - The scores for custom classifiers are not comparable to the scores returned by the General classifier (which has `classifier_id: "default"`)
  - The service attempts to normalize the score output so that 0.5 is a good decision threshold. By default, scores below 0.5 are not reported in the results of `/classify`. You can override this behavior by setting the threshold parameter of the `/classify` method. This normalization is only computed on the training data, so with new data or different application contexts, you might find that a different threshold works better.
  - The scores are unitless and are neither percentages nor probabilities. (They do not add up to 100% or 1.0).
- **Why do I get scores between 0.5 and 0.6 for images that I would expect to return a high score, near 1.0?**

  You might get lower scores if there is a significant amount of similarity between your classes, so that in the feature space your examples are not in distinct clusters, and the scores reflect this closeness to the best boundary between positives and negatives that the system could learn.

- **How can I evaluate the accuracy of a custom classifier for my use case?**

  You can do this in multiple ways, including the following method:

1. Assemble a set of labeled images "L" that was not used in training the classifier.
2. Split L into two sets, V and T - validation and testing.
3. Run V through your classifier and pick a score threshold "R" that optimizes the correctness metric you value, such as top-5 precision, across all of V.
4. From T, select a random subset "Q" and classify it using your classifier and "R". Compute the probability of a correct classification on Q. That's one experiment.
5. Repeat step 4 with a different subset Q from T, then compute the average % correct across all experiments.