Creating the resources

You can create the Custom Vision resources via the Azure portal, or in R using the facilities provided by AzureVision. Note that Custom Vision requires at least *two* resources to be created: one for training, and one for prediction. The available service tiers for Custom Vision are `F0` (free, limited to 2 projects for training and 10k transactions/month for prediction) and `S0`. Here is the R code for creating the resources:

```
library(AzureVision)

# insert your tenant, subscription, resgroup name and location here
rg <- AzureRMR::get_azure_login(tenant)$
    get_subscription(sub_id)$
    create_resource_group(rg_name, location=rg_location)

# insert your desired Custom Vision resource names here
res <- rg$create_cognitive_service(custvis_resname,
    service_type="CustomVision.Training", service_tier="S0")
pred_res <- rg$create_cognitive_service(custvis_predresname,
    service_type="CustomVision.Prediction", service_tier="S0")
```

Training

Custom Vision defines two different types of endpoint: a training endpoint, and a prediction endpoint. Somewhat confusingly, they can both use the same hostname, but with different URL paths and authentication keys. To start, call the `customvision_training_endpoint` function with the service URL and key.

```
url <- res$properties$endpoint
key <- res$list_keys()[1]

endp <- customvision_training_endpoint(url=url, key=key)
```

Custom Vision is organised hierarchically. At the top level, we have a *project*, which represents the data and model for a specific task. Within a project, we have one or more *iterations* of the model, built on different sets of training images. Each iteration in a project is independent: you can create (train) an iteration, deploy it, and delete it without affecting other iterations.

In turn, there are three different types of projects:

- A *multiclass classification* project is for classifying images into a set of *tags*, or target labels. An image can be assigned to one tag only.
- A *multilabel classification* project is similar, but each image can have multiple tags assigned to it.
- An *object detection* project is for detecting which objects, if any, from a set of candidates are present in an image.

Let's create a classification project:

```
testproj <- create_classification_project(endp, "testproj",
    export_target="standard")
```

Here, we specify the export target to be `standard` to support exporting the final model to one of various standalone formats, eg TensorFlow, CoreML or ONNX. The default is `none`, in which case the model stays on the Custom Vision server. The advantage of `none` is that the model can be more complex, resulting in potentially better accuracy.

Adding and tagging images

Since a Custom Vision model is trained in Azure and not locally, we need to upload some images. The data we'll use comes from the Microsoft Computer Vision Best Practices project. This is a simple set of images containing 4 kinds of objects one might find in a fridge: cans, cartons, milk bottles, and water bottles.

```
download.file(
    "https://cvbp.blob.core.windows.net/public/datasets/image_classification/fridgeObjects.zip",
    "fridgeObjects.zip"
)
unzip("fridgeObjects.zip")
```

The generic function to add images to a project is `add_images`, which takes a vector of filenames, Internet URLs or raw vectors as the images to upload. It returns a vector of *image IDs*, which are how Custom Vision keeps track of the images it uses.

Let's upload the fridge objects to the project. The method for classification projects has a `tags` argument which can be used to assign labels to the images as they are uploaded. We'll keep aside 5 images from each class of object to use as validation data.

```
cans <- dir("fridgeObjects/can", full.names=TRUE)
cartons <- dir("fridgeObjects/carton", full.names=TRUE)
milk <- dir("fridgeObjects/milk_bottle", full.names=TRUE)
water <- dir("fridgeObjects/water_bottle", full.names=TRUE)

# upload all but 5 images from cans and cartons, and tag them
can_ids <- add_images(testproj, cans[-(1:5)], tags="can")
carton_ids <- add_images(testproj, cartons[-(1:5)], tags="carton")
```

If you don't tag the images at upload time, you can do so later with `add_image_tags`:

```
# upload all but 5 images from milk and water bottles
milk_ids <- add_images(testproj, milk[-(1:5)])
water_ids <- add_images(testproj, water[-(1:5)])

add_image_tags(testproj, milk_ids, tags="milk_bottle")
add_image_tags(testproj, water_ids, tags="water_bottle")
```

Other image functions to be aware of include `list_images`, `remove_images`, and `add_image_regions` (which is for object detection projects). A useful one is `browse_images`, which takes a vector of IDs and displays the corresponding images in your browser.

```
browse_images(testproj, water_ids[1:5])
```

Training the model

Having uploaded the data, we can train the Custom Vision model with `train_model`. This trains the model on the server and returns a *model iteration*, which is the result of running the training algorithm on the current set of images. Each time you call `train_model`, for example to update the model after adding or removing images, you will obtain a different model iteration. In general, you can rely on AzureVision to keep track of the iterations for you, and automatically return the relevant results for the latest iteration.

```
mod <- train_model(testproj)
```

We can examine the model performance on the training data with the `summary` method. For this toy problem, the model manages to obtain a perfect fit.

```
summary(mod)
```

Obtaining predictions from the trained model is done with the `predict` method. By default, this returns the predicted tag (class label) for the image, but you can also get the predicted class probabilities by specifying

```
type="prob".

validation_imgs <- c(cans[1:5], cartons[1:5], milk[1:5], water[1:5])
validation_tags <- rep(c("can", "carton", "milk_bottle", "water_bottle"),
each=5)

predicted_tags <- predict(mod, validation_imgs)

table(predicted_tags, validation_tags)
```

```
##               validation_tags
## predicted_tags can carton milk_bottle water_bottle
##   can            4      0           0            0
##   carton         0      5           0            0
##   milk_bottle    1      0           5            0
##   water_bottle   0      0           0            5
```

This shows that the model got 19 out of 20 predictions correct on the validation data, misclassifying one of the cans as a milk bottle.

Deployment

Publishing to a prediction resource

The code above demonstrates using the training endpoint to obtain predictions, which is really meant only for model testing and validation. In a production setting, we would normally *publish* a trained model to a Custom Vision prediction resource. Among other things, a user with access to the training endpoint has complete freedom to modify the model and the data, whereas access to the prediction endpoint only allows getting predictions.

Publishing a model requires knowing the Azure resource ID of the prediction resource. Here, we'll use the resource object that we created earlier; you can also obtain this information from the Azure Portal.

```
# publish to the prediction resource we created above
publish_model(mod, "iteration1", pred_res)
```

Once a model has been published, we can obtain predictions from the prediction endpoint in a manner very similar to previously. We create a predictive service object with `classification_service`, and then call the `predict` method. Note that a required input is the project ID; you can supply this directly or via the project object. It may also take some time before a published model shows up on the prediction endpoint.

```
Sys.sleep(60)  # wait for Azure to finish publishing

pred_url <- pred_res$properties$endpoint
pred_key <- pred_res$list_keys()[1]

pred_endp <- customvision_prediction_endpoint(url=pred_url, key=pred_key)

project_id <- testproj$project$id
pred_svc <- classification_service(pred_endp, project_id, "iteration1")

# predictions from prediction endpoint -- same as before
predsvc_tags <- predict(pred_svc, validation_imgs)
table(predsvc_tags, validation_tags)
```

```
##               validation_tags
## predsvc_tags   can carton milk_bottle water_bottle
##   can            4      0           0            0
##   carton         0      5           0            0
```

```
##   milk_bottle    1       0           5            0
##   water_bottle   0       0           0            5
```

Exporting as standalone

As an alternative to deploying the model to an online predictive service resource, for example if you want to create a custom deployment solution, you can also export the model as a standalone object. This is only possible if the project was created to support exporting. The formats supported include:

- ONNX 1.2
- CoreML
- TensorFlow or TensorFlow Lite
- A Docker image for either the Linux, Windows or Raspberry Pi environment
- Vision AI Development Kit (VAIDK)

To export the model, simply call `export_model` and specify the target format. This will download the model to your local machine.

```
export_model(mod, "tensorflow")
```

More information

AzureVision is part of the AzureR family of packages. This provides a range of tools to facilitate access to Azure services for data scientists working in R, such as AAD authentication, blob and file storage, Resource Manager, container services, Data Explorer (Kusto), and more.