Train and deploy your first R model with Azure ML

David Smith

2020-02-25

In this tutorial, you learn the foundational design patterns in Azure Machine Learning. You'll train and deploy a Generalized Linear Model to predict the likelihood of a fatality in an automobile accident. After completing this tutorial, you'll have the practical knowledge of the R SDK to scale up to developing more-complex experiments and workflows.

In this tutorial, you learn the following tasks:

- Connect your workspace
- Load data and prepare for training
- Upload data to the datastore so it is available for remote training
- Create a compute resource
- Train a caret model to predict probability of fatality
- Deploy a prediction endpoint
- Test the model from R

Prerequisites

If you don't have access to an Azure ML workspace, follow the setup tutorial to configure and create a workspace.

Set up your development environment

The setup for your development work in this tutorial includes the following actions:

- Install required packages
- Connect to a workspace, so that your local computer can communicate with remote resources
- Create an experiment to track your runs
- Create a remote compute target to use for training

If you are using RStudio from a Notebook VM, open this tutorial as a project in RStudio with File > Open Project and select your cloned `train-and-deploy-to-aci` folder.

Install required packages

This tutorial assumes you already have the Azure ML SDK installed. (If you are running this vignette from an RStudio instance in an Azure ML Compute Instance or Notebook VM, the package is already installed for you.) Go ahead and load the **azuremlsdk** package.

```
library(azuremlsdk)
```

The training and scoring scripts (`accidents.R` and `accident_predict.R`) have some additional dependencies. If you plan on running those scripts locally, make sure you have those required packages as well.

Load your workspace

Instantiate a workspace object from your existing workspace. The following code will load the workspace details from the **config.json** file. You can also retrieve a workspace using `get_workspace()`.

```
ws <- load_workspace_from_config()
```

Create an experiment

An Azure ML experiment tracks a grouping of runs, typically from the same training script. Create an experiment to track the runs for training the caret model on the accidents data.

```
experiment_name <- "accident-logreg"
exp <- experiment(ws, experiment_name)
```

Create a compute target

By using Azure Machine Learning Compute (AmlCompute), a managed service, data scientists can train machine learning models on clusters of Azure virtual machines. Examples include VMs with GPU support. In this tutorial, you create a single-node AmlCompute cluster as your training environment. The code below creates the compute cluster for you if it doesn't already exist in your workspace.

You may need to wait a few minutes for your compute cluster to be provisioned if it doesn't already exist.

```
cluster_name <- "rcluster"
compute_target <- get_compute(ws, cluster_name = cluster_name)
if (is.null(compute_target)) {
  vm_size <- "STANDARD_D2_V2"
  compute_target <- create_aml_compute(workspace = ws,
                                       cluster_name = cluster_name,
                                       vm_size = vm_size,
                                       max_nodes = 2)

  wait_for_provisioning_completion(compute_target, show_output = TRUE)
}
```

This cluster has a maximum size of two nodes, but only one will be provisioned for now. The second will only be provisioned as needed, and will automatically de-provision when no longer in use. You can even set `min_nodes=0` to make the first node provision on demand as well (and experiments will wait for the node to provision before starting).

Prepare data for training

This tutorial uses data from the US National Highway Traffic Safety Administration
(with thanks to Mary C. Meyer and Tremika Finney). This dataset includes data from over 25,000 car

crashes in the US, with variables you can use to predict the likelihood of a fatality. First, import the data into R and transform it into a new dataframe `accidents` for analysis, and export it to an `Rdata` file.

```
nassCDS <- read.csv("nassCDS.csv",
                    colClasses=c("factor","numeric","factor",
                                 "factor","factor","numeric",
                                 "factor","numeric","numeric",
                                 "numeric","character","character",
                                 "numeric","numeric","character"))

accidents <- na.omit(nassCDS[,c("dead","dvcat","seatbelt","frontal","sex","ageOFocc","yearVeh","airbag"
accidents$frontal <- factor(accidents$frontal, labels=c("notfrontal","frontal"))
accidents$occRole <- factor(accidents$occRole)
accidents$dvcat <- ordered(accidents$dvcat,
                           levels=c("1-9km/h","10-24","25-39","40-54","55+"))

saveRDS(accidents, file="accidents.Rd")
```

Upload data to the datastore

Upload data to the cloud so that it can be access by your remote training environment. Each Azure ML workspace comes with a default datastore that stores the connection information to the Azure blob container that is provisioned in the storage account attached to the workspace. The following code will upload the accidents data you created above to that datastore.

```
ds <- get_default_datastore(ws)

target_path <- "accidentdata"
upload_files_to_datastore(ds,
                          list("./accidents.Rd"),
                          target_path = target_path,
                          overwrite = TRUE)
```

Train a model

For this tutorial, fit a logistic regression model on your uploaded data using your remote compute cluster. To submit a job, you need to:

- Prepare the training script
- Create an estimator
- Submit the job

Prepare the training script

A training script called `accidents.R` has been provided for you in the same directory as this tutorial. Notice the following details **inside the training script** that have been done to leverage the Azure ML service for training:

- The training script takes an argument `-d` to find the directory that contains the training data. When you define and submit your job later, you point to the datastore for this argument. Azure ML will mount the storage folder to the remote cluster for the training job.

- The training script logs the final accuracy as a metric to the run record in Azure ML using `log_metric_to_run()`. The Azure ML SDK provides a set of logging APIs for logging various metrics during training runs. These metrics are recorded and persisted in the experiment run record. The metrics can then be accessed at any time or viewed in the run details page in Azure Machine Learning studio. See the reference for the full set of logging methods `log_*()`.
- The training script saves your model into a directory named **outputs**. The `./outputs` folder receives special treatment by Azure ML. During training, files written to `./outputs` are automatically uploaded to your run record by Azure ML and persisted as artifacts. By saving the trained model to `./outputs`, you'll be able to access and retrieve your model file even after the run is over and you no longer have access to your remote training environment.

Create an estimator

An Azure ML estimator encapsulates the run configuration information needed for executing a training script on the compute target. Azure ML runs are run as containerized jobs on the specified compute target. By default, the Docker image built for your training job will include R, the Azure ML SDK, and a set of commonly used R packages. See the full list of default packages included here.

To create the estimator, define:

- The directory that contains your scripts needed for training (`source_directory`). All the files in this directory are uploaded to the cluster node(s) for execution. The directory must contain your training script and any additional scripts required.
- The training script that will be executed (`entry_script`).
- The compute target (`compute_target`), in this case the AmlCompute cluster you created earlier.
- The parameters required from the training script (`script_params`). Azure ML will run your training script as a command-line script with `Rscript`. In this tutorial you specify one argument to the script, the data directory mounting point, which you can access with `ds$path(target_path)`.
- Any environment dependencies required for training. The default Docker image built for training already contains the three packages (`caret`, `e1071`, and `optparse`) needed in the training script. So you don't need to specify additional information. If you are using R packages that are not included by default, use the estimator's `cran_packages` parameter to add additional CRAN packages. See the `estimator()` reference for the full set of configurable options.

```
est <- estimator(source_directory = ".",
                 entry_script = "accidents.R",
                 script_params = list("--data_folder" = ds$path(target_path)),
                 compute_target = compute_target
                 )
```

Submit the job on the remote cluster

Finally submit the job to run on your cluster. `submit_experiment()` returns a Run object that you then use to interface with the run. In total, the first run takes **about 10 minutes**. But for later runs, the same Docker image is reused as long as the script dependencies don't change. In this case, the image is cached and the container startup time is much faster.

```
run <- submit_experiment(exp, est)
```

You can view a table of the run's details. Clicking the "Web View" link provided will bring you to Azure Machine Learning studio, where you can monitor the run in the UI.

```
view_run_details(run)
```

Model training happens in the background. Wait until the model has finished training before you run more code.

```
wait_for_run_completion(run, show_output = TRUE)
```

You – and colleagues with access to the workspace – can submit multiple experiments in parallel, and Azure ML will take of scheduling the tasks on the compute cluster. You can even configure the cluster to automatically scale up to multiple nodes, and scale back when there are no more compute tasks in the queue. This configuration is a cost-effective way for teams to share compute resources.

Retrieve training results

Once your model has finished training, you can access the artifacts of your job that were persisted to the run record, including any metrics logged and the final trained model.

Get the logged metrics

In the training script `accidents.R`, you logged a metric from your model: the accuracy of the predictions in the training data. You can see metrics in the studio, or extract them to the local session as an R list as follows:

```
metrics <- get_run_metrics(run)
metrics
```

If you've run multiple experiments (say, using differing variables, algorithms, or hyperparamers), you can use the metrics from each run to compare and choose the model you'll use in production.

Get the trained model

You can retrieve the trained model and look at the results in your local R session. The following code will download the contents of the `./outputs` directory, which includes the model file.

```
download_files_from_run(run, prefix="outputs/")
accident_model <- readRDS("outputs/model.rds")
summary(accident_model)
```

You see some factors that contribute to an increase in the estimated probability of death:

- higher impact speed
- male driver
- older occupant
- passenger

You see lower probabilities of death with:

- presence of airbags
- presence seatbelts

- frontal collision

The vehicle year of manufacture does not have a significant effect.

You can use this model to make new predictions:

```
newdata <- data.frame( # valid values shown below
 dvcat="10-24",        # "1-9km/h" "10-24"   "25-39"   "40-54"   "55+"
 seatbelt="none",      # "none"   "belted"
 frontal="frontal",    # "notfrontal" "frontal"
 sex="f",              # "f" "m"
 ageOFocc=16,          # age in years, 16-97
 yearVeh=2002,         # year of vehicle, 1955-2003
 airbag="none",        # "none"   "airbag"
 occRole="pass"        # "driver" "pass"
 )

## predicted probability of death for these variables, as a percentage
as.numeric(predict(accident_model,newdata, type="response")*100)
```

Deploy as a web service

With your model, you can predict the danger of death from a collision. Use Azure ML to deploy your model as a prediction service. In this tutorial, you will deploy the web service in Azure Container Instances (ACI).

Register the model

First, register the model you downloaded to your workspace with `register_model()`. A registered model can be any collection of files, but in this case the R model object is sufficient. Azure ML will use the registered model for deployment.

```
model <- register_model(ws,
                        model_path = "outputs/model.rds",
                        model_name = "accidents_model",
                        description = "Predict probablity of auto accident")
```

Define the inference dependencies

To create a web service for your model, you first need to create a scoring script (`entry_script`), an R script that will take as input variable values (in JSON format) and output a prediction from your model. For this tutorial, use the provided scoring file `accident_predict.R`. The scoring script must contain an `init()` method that loads your model and returns a function that uses the model to make a prediction based on the input data. See the documentation for more details.

Next, define an Azure ML **environment** for your script's package dependencies. With an environment, you specify R packages (from CRAN or elsewhere) that are needed for your script to run. You can also provide the values of environment variables that your script can reference to modify its behavior. By default, Azure ML will build the same default Docker image used with the estimator for training. Since the tutorial has no special requirements, create an environment with no special attributes.

```
r_env <- r_environment(name = "basic_env")
```

If you want to use your own Docker image for deployment instead, specify the `custom_docker_image` parameter. See the `r_environment()` reference for the full set of configurable options for defining an environment.

Now you have everything you need to create an **inference config** for encapsulating your scoring script and environment dependencies.

```
inference_config <- inference_config(
  entry_script = "accident_predict.R",
  source_directory = ".",
  environment = r_env)
```

Deploy to ACI

In this tutorial, you will deploy your service to ACI. This code provisions a single container to respond to inbound requests, which is suitable for testing and light loads. See `aci_webservice_deployment_config()` for additional configurable options. (For production-scale deployments, you can also deploy to Azure Kubernetes Service.)

```
aci_config <- aci_webservice_deployment_config(cpu_cores = 1, memory_gb = 0.5)
```

Now you deploy your model as a web service. Deployment **can take several minutes**.

```
aci_service <- deploy_model(ws,
                            'accident-pred',
                            list(model),
                            inference_config,
                            aci_config)

wait_for_deployment(aci_service, show_output = TRUE)
```

If you encounter any issue in deploying the web service, please visit the troubleshooting guide.

Test the deployed service

Now that your model is deployed as a service, you can test the service from R using `invoke_webservice()`. Provide a new set of data to predict from, convert it to JSON, and send it to the service.

```
library(jsonlite)

newdata <- data.frame( # valid values shown below
 dvcat="10-24",        # "1-9km/h" "10-24"   "25-39"   "40-54"   "55+"
 seatbelt="none",      # "none"   "belted"
 frontal="frontal",    # "notfrontal" "frontal"
 sex="f",              # "f" "m"
 ageOFocc=22,          # age in years, 16-97
 yearVeh=2002,         # year of vehicle, 1955-2003
 airbag="none",        # "none"   "airbag"
 occRole="pass"        # "driver" "pass"
 )
```

```
prob <- invoke_webservice(aci_service, toJSON(newdata))
prob
```

Clean up resources

Delete the resources once you no longer need them. Don't delete any resource you plan to still use.

Delete the web service:

```
delete_webservice(aci_service)
```

Delete the registered model:

```
delete_model(model)
```

Delete the compute cluster:

```
delete_compute(compute_target)
```