R on Databricks Compendium

Zac Davies

2024-06-20

Table of contents

What is this?

Under Development

This is not intended to be an exhaustive guide, it's currently a place for me to document and collate useful information regarding R on Databricks.

There aren't many definitive examples of how to use R and Databricks together - hopefully the content here will serve as a useful resource.

Part I

Mlflow

1 Log R Models to Unity Catalog

Currently {mlflow} doesn't support directly logging R models to Unity Catalog. This section will cover why, and then how to overcome each roadblock.

1.1 Unity Catalog Model Requirements

For models to be logged into Unity Catalog they **must** have a model signature. The Model signature defines the schema for model inputs/outputs.

Typically when using python this would be inferred via model input examples. Input examples are optional but strongly recommended.

The documentation discusses signature enforcement, currently this isn't implemented for R. Therefore you can decide if the signature is a dummy value for the sake of moving forward, or correct to clearly communicate the behaviour of the model.

> **!** Important
>
> It's important to clarify that for python the signature is enforced at time of inference *not* when registering the model to Unity Catalog.
> The signature correctness is not validated when registering the model, it just has to be syntactically valid.

So, let's look at the existing code to log models in the `crate` flavour:

```
mlflow_save_model.crate <- function(model, path, model_spec=list(), ...) {
  if (dir.exists(path)) unlink(path, recursive = TRUE)                  ①
  dir.create(path)

  serialized <- serialize(model, NULL)                                  ②

  saveRDS(                                                              ③
    serialized,
    file.path(path, "crate.bin")
  )
```

```r
  model_spec$flavors <- append(model_spec$flavors, list(          # (4)
    crate = list(
      version = "0.1.0",
      model = "crate.bin"
    )
  ))
  mlflow_write_model_spec(path, model_spec)
  model_spec
}
```

1. Create the directory to save the model if it doesn't exist, if it does, empty it
2. Serialise the model, which is an object of class `crate` (from `{carrier}` package)
3. Save the serialised model via `saveRDS` to the directory as `crate.bin`
4. Define the model specification, this contains metadata required ensure reproducibility. In this case it's only specifying a version and what file the model can be found within.

The missing puzzle piece is the definition of a signature. Instead of explicitly adding code to the crate flavour itself, we'll take advantage of the `model_spec` parameter.

That means we can focus on `mlflow::mlflow_log_model` directly, we'd need to adjust the code as follows:

```r
mlflow_log_model <- function(model, artifact_path, signature, ...) {          # (1)

  temp_path <- fs::path_temp(artifact_path)

  model_spec <- mlflow_save_model(
    model, path = temp_path,
    model_spec = list(                                                         # (2)
      utc_time_created = mlflow_timestamp(),
      run_id = mlflow_get_active_run_id_or_start_run(),
      artifact_path = artifact_path,
      flavors = list(),
      signature = signature
    ),
  ...)

  res <- mlflow_log_artifact(path = temp_path, artifact_path = artifact_path)

  tryCatch({
    mlflow:::mlflow_record_logged_model(model_spec)
  },
```

```
  error = function(e) {
    warning(
      paste("Logging model metadata to the tracking server has failed, possibly due to older"
            "server version. The model artifacts have been logged successfully.",
            "In addition to exporting model artifacts, MLflow clients 1.7.0 and above",
            "attempt to record model metadata to the  tracking store. If logging to a",
            "mlflow server via REST, consider  upgrading the server version to MLflow",
            "1.7.0 or above.", sep=" ")
    )
  })
  res
}
```

① Add a new parameter `signature`
② Propagate `signature` to the `model_spec` parameter when invoking `mlflow::mlflow_save_model`

Benefit of this method is that all model flavors will inherit the capability to log a signature.

1.2 Working Through the Solution

To keep things simple we'll be logging a "model" (a function which divides by two).

```
half <- function(x) x / 2

half(1:10)
```

```
 [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Without any changes, a simplified example of logging to `{mlflow}` would look like:

```
library(carrier)
library(mlflow)

with(mlflow_start_run(), {
  # typically you'd do more modelling related activities here
  model <- carrier::crate(~half(.x))
  mlflow_log_model(model, "model")                                              ①
})
```

① As discussed earlier, this is where things start to go awry with respect to Unity Catalog

1.2.1 Patching `mlflow_log_model`

> **Note**
>
> Technically, patching `mlflow_log_model` isn't the only way to achieve this fix - you could modify the yaml after it's written.
> I won't be showing that method as It's just as tedious and can change depending on the model flavour (with respect to where artifacts may reside), patching is more robust.

```
mlflow_log_model <- function(model, artifact_path, signature = NULL, ...) { ①

  format_signature <- function(signature) {                                  ②
    lapply(signature, function(x) {
      jsonlite::toJSON(x, auto_unbox = TRUE)
    })
  }

  temp_path <- fs::path_temp(artifact_path)

  model_spec <- mlflow_save_model(model, path = temp_path, model_spec = list(
    utc_time_created = mlflow:::mlflow_timestamp(),
    run_id = mlflow:::mlflow_get_active_run_id_or_start_run(),
    artifact_path = artifact_path,
    flavors = list(),
    signature = format_signature(signature)                                  ③
  ), ...)

  res <- mlflow_log_artifact(path = temp_path, artifact_path = artifact_path)

  tryCatch({
    mlflow:::mlflow_record_logged_model(model_spec)
  },
  error = function(e) {
    warning(
      paste("Logging model metadata to the tracking server has failed, possibly due to older"
            "server version. The model artifacts have been logged successfully.",
            "In addition to exporting model artifacts, MLflow clients 1.7.0 and above",
            "attempt to record model metadata to the  tracking store. If logging to a",
            "mlflow server via REST, consider  upgrading the server version to MLflow",
            "1.7.0 or above.", sep=" ")
    )
  })
```

```
  res
}

# overriding the function in the existing mlflow namespace
assignInNamespace("mlflow_log_model", mlflow_log_model, ns = "mlflow")
```

1. `signature` has been added to function parameters, it's defaulting to `NULL` so that existing code won't break
2. Adding `format_signature` function so don't need to write JSON by hand, adding this within function for simplicity
3. `signature` is propagated to `mlflow_save_model`'s `model_spec` parameter which will write a valid signature

1.2.2 Logging Model with a Signature

```
with(mlflow_start_run(), {
  # typically you'd do more modelling related activities here
  model <- carrier::crate(~half(.x))
  signature <- list(                                                    (1)
    inputs = list(list(type = "double", name = "x")),
    outputs = list(list(type = "double"))
  )
  mlflow_log_model(model, "model", signature = signature)               (2)
})
```

1. Explicitly defining a `signature`, a list that contains `input` and `outputs`, each are lists of lists respectively
2. Passing defined signature to the now patched `mlflow_log_model` function

1.2.3 Registration to Unity Catalog

Now that the prerequisite of adding a model signature has been satisfied there is one last hurdle to overcome, registering to Unity Catalog.

The hurdle is due to `{mlflow}` not having been updated yet to support registration to Unity Catalog directly. The easiest way to overcome this is to simply register the run via python.

For example:

```python
import mlflow
mlflow.set_registry_uri("databricks-uc")

catalog = "main"
schema = "default"
model_name = "my_model"
run_uri = "runs:/<run_id>/model"                                   # (1)

mlflow.register_model(run_uri, f"{catalog}.{schema}.{model_name}")
```

(1) You'll need to either get the `run_uri` programmatically or copy it manually

To do this with R you'll need to make a series of requests to Unity Catalog endpoints for registering model, the specific steps are:

1. **(Optional) Create a new model in Unity Catalog**
    - `POST` request on `/api/2.0/mlflow/unity-catalog/registered-models/create`
        - `name`: 3 tiered namespace (e.g. `main.default.my_model`)
2. **Create a version for the model**
    - `POST` request on `/api/2.0/mlflow/unity-catalog/model-versions/create`
        - `name`: 3 tiered namespace (e.g. `main.default.my_model`)
        - `source`: URI indicating the location of the model artifacts
        - `run_id`: `run_id` from tracking server that generated the model
        - `run_tracking_server_id`: Workspace ID of run that generated the model
    - This will return `storage_location` and `version`
3. **Copy model artifacts**
    - Need to copy the artifacts to `storage_location` from step (2)
4. **Finalise the model version**
    - `POST` request on `/api/2.0/mlflow/unity-catalog/model-versions/finalize`
        - `name`: 3 tiered namespace (e.g. `main.default.my_model`)
        - `version`: `version` returned from step (2)

It's *considerably* easier to just use Python to register the model at this time.

1.3 Fixing `mlflow`

Ideally this page wouldn't exist and `{mlflow}` would support Unity Catalog. Hopefully sometime soon I find the time to make a pull request myself - until then this serves as a guide.

2 Model Serving

> Under Development

Model serving on Databricks is well documented for Python. It is a common misconception that you cannot serve R models on Databricks, you can!

For most model flavours the amount of effort is not worth it - but that decision is not mine to make, I will show you how and you can decide for yourself.

2.1 Serving Overview

> Caution
>
> Work In Progress

2.2 XGBoost

> Caution
>
> Work In Progress

XGBoost is relatively straightforward since it the artifacts generated when saving the model are universally understood across each language the library is available in.

2.3 ONNX

> Caution
>
> Work In Progress

2.4 Crate

> Caution
>
> Work In Progress

Part II

Package Management

3 Faster Package Installs

You may have noticed that when installing packages in the notebook it can take a while. It could be minutes, hours in extreme cases, to install the suite of packages your project requires. This is especially tedious if you need to do this every time a job runs, or each morning when your cluster is started.

Clusters are ephemeral and by default have no persistent storage, therefore installed packages will not be available on restart.

By default Databricks installs packages from CRAN. CRAN does not provide pre-compiled binaries for Linux (Databricks clusters' underlying virtual machines are Linux, Ubuntu specifically).

Posit to save the day! Posit provides a public package manager that has all packages from CRAN (and Bioconductor!). There is a helpful wizard to get started.

With our new found knowledge we can make installing R packages within Databricks significantly faster. There are multiple ways to solve this, each differing slightly, but fundamentally the same.

3.1 Setting Repo within Notebook

The quickest method is to follow the wizard and adjust the `repos` option:

```r
# set the user agent string otherwise pre-compiled binarys aren't used
# e.g. selecting Ubuntu 22.04 in wizard
options(
  HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["plat
  repos = "https://packagemanager.posit.co/cran/__linux__/jammy/latest"
)
```

① `HTTPUserAgent` is required when using R 3.6 or later

This works well but not all versions of the Databricks Runtime use the same version of Ubuntu.

It's easier to detect the Ubuntu release code name dynamically:

```
release <- system("lsb_release -c --short", intern = T)                    ①

# set the user agent string otherwise pre-compiled binarys aren't used
options(
  HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["plat
  repos = paste0("https://packagemanager.posit.co/cran/__linux__/", release, "/latest")
)
```

① `system` is used to run the command to retrieve the release code name

The downside of this method is that it requires every notebook to adjust the `repos` and `HTTPUserAgent` options.

3.2 Cluster Environment Variable & Init Script

Databricks clusters allow specification of environment variables, there is a specific variable (`DATABRICKS_DEFAULT_R_REPOS`) that can be set to adjust the default repository for the entire cluster.

You can again refer to the wizard, the environment variables section of cluster should be:

```
DATABRICKS_DEFAULT_R_REPOS=<posit-package-manager-url-goes-here>
```

Unfortunately this isn't as dynamic as the first option and you still need to set the `HTTPUserAgent` in `Rprofile.site` via an init script.

The init script will be:

```
#!/bin/bash
# Append changes to Rprofile.site
cat <<EOF >> "/etc/R/Rprofile.site"
options(
  HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["plat
)
EOF
```

> **! Important**
>
> Due to how Databricks starts up the R shell for notebook sessions it's not straightforward to adjust the `repos` option in an init script alone.
> `DATABRICKS_DEFAULT_R_REPOS` is referenced as part of the startup process *after*

`Rprofile.site` is executed and will override any earlier attempt to adjust `repos`. Therefore you'll need to use both the init script and the environment variable configuration.

3.3 Setting Repo for Cluster Library

> **Note**
>
> Similar to setting `DATABRICKS_DEFAULT_R_REPOS` this requires the `HTTPUserAgent` also to be set and it's unlikely to be helpful other than for it's purpose of installing a package to make it available for all cluster users.

Cluster libraries can install R packages and support specification of the repository.

4 Persisting Packages

> **!** Important
>
> Evaluate if faster package installs is able to solve any installation pain-points before investigating persisting packages, faster installs is easier to set-up and manage.

Databricks clusters are ephemeral and therefore any installed packages will not be available on restart. If the cluster has cluster libraries defined then those libraries are installed after the cluster is started - this can be time consuming when there are multiple packages.

The article on faster package installs details how to reduce the time it takes to install each package. Faster installs are great, but sometimes it's preferable to not install at all and persist the packages required, similar to how you'd use R locally.

4.1 Where Packages are Installed

When installing packages with `install.packages` the default behaviour is that they'll be installed to the first element of `.libPaths()`.

`.libPaths()` returns the paths of "R library trees", directories that R packages can reside. When you load a package it will be loaded from the first location it is found as dictated by `.libPaths()`.

When working within a Databricks notebook `.libPaths()` will return 6 values by default, in order they are:

Path	Details
`/local_disk0/.ephemeral_nfs/envs/rEnv-<session-id>`	The first location is always a notebook specific directory, this is what allows each notebook session to have different libraries installed.

Path	Details
`/databricks/spark/R/lib`	Only `{SparkR}` is found here
`/local_disk0/.ephemeral_nfs/cluster_libraries/r`	Cluster libraries - you could also install packages here explicitly to share amongst all users (e.g. `lib` parameter of `install.packages`)
`/usr/local/lib/R/site-library`	Packages built into the Databricks Runtime
`/usr/lib/R/site-library`	Empty
`/usr/lib/R/library`	Base R packages

It's important to understand that the order defines the default behaviour as It's possible to add or remove values in `.libPaths()`. You'll almost certainly be adding values, there's little reason to remove values.

> **Note**
>
> All following examples will use Unity Catalog Volumes. DBFS can be used but it's not recommended.

4.2 Persisting a Package

The recommended approach is to first install the library(s) you want to persist on a cluster via a notebook.

For example, let's persist `{leaflet}` to a volume:

```r
install.packages("leaflet")                                                    # (1)

# determine where the package was installed
pkg_location <- find.package("leaflet")                                        # (2)

# move package to volume
new_pkg_location <- "/Volumes/<catalog>/<schema>/<volume>/my_packages"         # (3)
file.copy(from = pkg_location, to = new_pkg_location, recursive = TRUE)        # (4)
```

1. Installing `{leaflet}`
2. Return path to package files, from what was explained before we know this will be a sub-directory of `.libPaths()` first path
3. Define the path to volume where package will be persisted, make sure to adjust as needed
4. Copy the folder contents recursively to the volume

At this point the package is persisted, but if you restart the cluster or detach and reattach and try to load `{leaflet}` it will fail to load.

The last step is to adjust `.libPaths()` to include the volume path. You could make it the first value by:

```
# adjust .libPaths
.libPaths(c(new_pkg_location, .libPaths()))  # (1)
```

1. I recommend against making it the first value, will detail why in Ordering

4.3 Adjusting `.libPaths()`

4.3.1 Ordering

Given that `.libPaths()` can return 6 values in a notebook you might wonder if there a "best" position to add your new volume path(s) to, that will depend on how you want packages to behave.

A safe default is to add a path *after* the cluster libraries location (currently 3rd), this will make it appear as if the Databricks Runtime has been extended to include packages in the volume path(s).

Alternatively you could add it after the first path and all users will still have the notebook scope package behaviour by default but cluster libraries may not load if they appear in the earlier paths under a different version.

It will be up to you to decide what works best.

> **!** Important
>
> I don't recommend pre-pending `.libPaths()` with volume paths as packages will attempt to install to the first value and you cannot directly install packages to a volume path (due to volumes being backed onto cloud storage). This is why the example for persisting copies after installation.

An example of adjusting `.libPaths()` looks like:

```r
volume_pkgs <- "/Volumes/<catalog>/<schema>/<volume>/my_packages"
.libPaths(new = append(.libPaths(), volume_pkgs, after = 3))
```

4.3.2 Helpful Functions

The examples can be used to build a set of functions to make this easier.

Copying a Package

```r
copy_package <- function(name, destination) {
  package_loc <- find.package(name)
  file.copy(from = package_loc, to = destination, recursive = TRUE)
}

# e.g. move {ggplot2} to volume
copy_package("ggplot2", "/Volumes/<catalog>/<schema>/<volume>/my_packages")
```

Alter `.libPaths()`

```r
add_lib_paths <- function(path, after, version = FALSE) {
  if (version) {                                                    # (1)
    rver <- getRversion()
    lib_path <- file.path(path, rver)
  } else {
    lib_path <- file.path(path)
  }

  # ensure directory exists
  if (!file.exists(lib_path)) {
    dir.create(lib_path, recursive = TRUE)
  }

  lib_path <- normalizePath(lib_path, "/")

  message("primary package path is now ", lib_path)
  .libPaths(new = append(.libPaths(), lib_path, after = after))
  lib_path
}
```

(1) Allows specifying `version` as `TRUE` or `FALSE` to suffix the supplied `path` with the current R version

4.3.3 Avoiding Repetition

To avoid manually adjusting `.libPaths()` every notebook you can craft an init script or set environment variables, depending on the desired outcome.

> **Caution**
>
> In practice this interferes with how Databricks sets up the environment, validate any changes thoroughly before rolling out to users.

4.3.3.1 Init Script

> **Note**
>
> This example appends to the existing `Renviron.site` file to ensure any settings defined as part of runtime are preserved.
> The last two lines of the script are setting `R_LIBS_SITE` and `R_LIBS_USER`. Changing these lines can give you granular control over order for anything after the 1st value of `.libPaths()` as it's injected when the notebook session starts.

```
#!/bin/bash
volume_pkgs=/Volumes/<catalog>/<schema>/<volume>/my_packages                    (1)
cat <<EOF >> "/etc/R/Renviron.site"
R_LIBS_USER=%U:/databricks/spark/R/lib:/local_disk0/.ephemeral_nfs/cluster_libraries/r:$volu
EOF
```

(1) Define the path(s) to add to `R_LIBS_USER`
(2) Append line to `/etc/R/Renviron.site` with location after cluster libraries, you can rearrange the paths as long as they remain `:` separated

4.3.3.2 Environment Variables

> **Caution**
>
> How the Databricks Runtime defines and uses the R environment variables is something that may change and should be tested carefully, especially if upgrading runtime versions.

There are particular environment variables (`R_LIBS`, `R_LIBS_USER`, `R_LIBS_SITE`) that can be set to initialise the library search path (`.libPaths()`).

`R_LIBS` and `R_LIBS_USER` are defined as part of start-up processes in Databricks Runtime and they'll be overridden, it's easier to adjust via an Init Script.

`R_LIBS_SITE` can be set via an environment variable but is referenced by `/etc/R/Renviron.site` and will provides limited control over where the path will appear in the `.libPaths()` order (it will appear 5th, after the packages included in the Databricks runtime) unless using an init script to alter `/etc/R/Renviron.site` directly.

4.4 Organising Packages

When going down this route of persisting packages you should consider how this is organised and managed long term to avoid making things messy.

Some practices you can consider include:

- Maintaining directories of packages per project, team, or user
- Ensuring directories are specific to an R version (and potentially even Databricks Runtime version)
- Coupling the use of persistence with `{renv}`

5 {renv}

Under Development

Part III

Data Engineering

6 {dbplyr} & {odbc}

Under Development

Part IV

Miscellaneous

7 {htmlwidgets} in notebooks

When you try to view a {htmlwidget} based visualisation (e.g. {leaflet}) in a Databricks notebook you'll find there is no rendered output by default.

The Databricks documentation details how to get this working but requires specification of the workspace URL explicitly and writes out files to DBFS's FileStore without cleaning up after itself.

The new method avoids those steps and is drastically simplified and easier to use, just run the below function in a Databricks notebook:

```
enable_htmlwidgets <- function(height = 450) {

  # new option to control default widget height, default is 450px
  options(db_htmlwidget_height = height)                                    ①

  system("apt-get update && apt-get --yes install pandoc", intern = T)      ②
  if (!base::require("htmlwidgets")) {                                      ③
    utils::install.packages("htmlwidgets")
  }

  # new method will fetch height based on new option, or default to 450px
  new_method <- function(x, ...) {                                          ④
    x$height <- getOption("db_htmlwidget_height", 450)
    file <- tempfile(fileext = ".html")
    htmlwidgets::saveWidget(x, file = file)
    contents <- as.character(rvest::read_html(file))
    displayHTML(contents)
  }

  utils::assignInNamespace("print.htmlwidget", new_method, ns = "htmlwidgets") ⑤
  invisible(list(default_height = height, print = new_method))

}
```

① The height of the htmlwidget output is controlled via an option (`db_htmlwidget_height`), this allows the height to be adjusted without re-running the function

② Installing `pandoc` as it's required to use `htmlwidgets::saveWidget`
③ Ensure that `{htmlwidgets}` is installed
④ Function that writes the widget to a temporary file as a self-contained `html` file and then reads the contents and presents via `displayHTML`
⑤ Override the `htmlwidgets::print.htmlwidget` method

8 OAuth {odbc}

When using `{odbc}` to connect to Databricks clusters and SQL warehouses you'll likely have used a personal access token (PAT). It's not uncommon for workspace administrators to disable the use of PATs.

If you are unable to create a PAT you are still able to connect to Databricks but you'll need to use OAuth (either M2M or U2M).

User-to-machine (U2M) is typically what you'd want to use. Good news, the Databricks ODBC driver supports both since `2.7.5`.

8.1 U2M Example

> **Note**
>
> OAuth U2M or OAuth 2.0 browser-based authentication works only with applications that run locally. It does not work with server-based or cloud-based applications.

When running this code you should be prompted to login to the workspace or you'll see a window that says "success". You can close the window and continue working in R.

```
library(odbc)
library(DBI)

con <- DBI::dbConnect(
  drv = odbc::databricks(),                                  # (1)
  httpPath = "/sql/1.0/warehouses/<warehouse-id>",           # (2)
  workspace = "<workspace-name>.cloud.databricks.com",       # (3)
  authMech = 11,                                             # (4)
  auth_flow = 2
)
```

① `{odbc}` recently added `odbc::databricks()` to simplify connecting to Databricks (requires version `>=1.4.0`)
② The `httpPath` can be found in the 'Connection Details' tab of a SQL warehouse

③ `workspace` refers to the workspace URL, also found in 'Connection Details' tab as 'Server hostname'

④ The docs mention setting `AuthMech` to `11` and `Auth_Flow` to `2`