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(
    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, ...) {

  temp_path <- fs::path_temp(artifact_path)

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

  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

Part II

Package Management

3 Faster Package Installs

Under Development

4 Persisting Packages

Under Development

5 {renv}

Under Development

Part III

Data Engineering

6 {dbplyr} & {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`.

6.1 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.

```
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`

Part IV

Miscellaneous

7 {htmlwidgets} in notebooks

Under Development

8 OAuth `{odbc}`

> Under Development