R-bridge-script-tool-functionality

Create R Script Tools

Another main use of the bridge is to wrap R functionality into a geoprocessing script so it can be called as a tool in ArcGIS. This is useful when you have functions in R that you use repeatedly and want to visualize the results on a map, to continue your analysis in ArcGIS, or to share with others. R script tools can be fully integrated into ArcGIS and seamlessly linked with other Geoprocessing tools, or Python script tools to create thorough and powerful workflows. This makes it easier to share your workflows with other members of your organization, especially with those that are not familiar with the R language.

See below for the various topics related to R script tool creation:

- arcgisbinding Script Tool Functionality
  - arc.env Functionality
  - arc.progress_label and arc.progress_pos Functionality
- Components of an R Script Tool
  - Required R Packages
  - Define Inputs and Outputs
  - Open and Read-in Data
  - Custom R Functionality
  - Generate Results and Messages
  - Example R Script Tools
- Components of ArcGIS Script Tool UI Creation
- Hands-on Practice: Script Tool Creation

arcgisbinding Script Tool Functionality

The **arcgisbinding** package contains several functions to help you design and customize the new script tools you create:

- `arc.env()`
- `arc.progress_label()`
- `arc.progress_pos()`

arc.env Functionality

When working in ArcGIS, a user has the ability to customize their project environment using geoprocessing environment settings. This includes things such as setting an output coordinate system, defining a processing extent, setting a random number seed, etc. If a user has preset some of their environment variables, this may impact the results of your R script tool. The `arc.env()` function enables you to get the local ArcGIS geoprocessing tool environment settings and to check if they are set appropriately.

When used in the sample code snippet below, the geoprocessing tool environment settings are stored in the `env` variable by using the `arc.env()` function. This allows for you to check a user's settings as needed within your script. While you are unable to override a user's local settings, you can throw a warning or error message to alert the user that they need to adjust their environments before continuing with your tool if a certain setting impacts your tool's results.

> *Note: The code snippets below do not represent a complete script tool on their own. These examples are for teaching purposes only. To view completed script tools that you can run as is, please check out our sample script tools.*

For example, below we check to see what the package workspace path is set to.

```
tool_exec <- function(in_params, out_params)
 {
  env = arc.env()
  wkspath <- env$workspace
  .
  .
  .
  return (out_params)
 }
```

An overview of all the different environment settings possible can be found here.

arc.progress__label and arc.progress__pos Functionality

The **arcgisbinding** package also allows you to customize the tool run experience users have with your script tool through use of the `arc.progress_label()` and `arc.progress_pos()` functions.

> *Note: Currently, these functions only work with ArcGIS Pro.*

When running a geoprocessing tool in ArcGIS, a tool run status bar appears that can display custom messages to indicate the status of the run. In addition, the tool run progress bar can be customized to appear at different levels of completion based on the current stage of the run. With the `arc.progress_label()` function, you can customize what messages your users see based on where the script is at while they are running your tool. While the tool run status bar typically continuously moves back and forth by default, if you wish, you can customize it to appear at a certain percentage of completion by entering a value from 0 to 100 to represent the current percentage. Both of these functions are used multiple times in the sample script tool below to demonstrate how they might be used when creating a new geoprocessing script tool.

Here are some examples excerpted from a script tool:

```
tool_exec <- function(in_params, out_params)
 {
  arc.progress_label("Loading Dataset...")
  arc.progress_pos(25)
  .
  .
  .
  arc.progress_label("Obtaining Attribute Names...")
  arc.progress_pos(50)
  .
  .
  .
  arc.progress_label("Obtaining Geometry Information...")
  arc.progress_pos(75)
  .
  .
  .
  arc.progress_label("Writing result dataset...")
  arc.progress_pos(100)
  .
  .
  .
  return (out_params)
 }
```

Components of an R Script Tool

While the **arcgisbinding** package offers a high level of customization over the resulting script tools that are produced from wrapped R functionality, each script, no matter the analysis, will follow a similar framework and will contain certain elements.

To begin, the `tool_exec()` function is the container each of these elements will fall within. Since R script tools expect a return by default, we have included the line `return (out_params)` to avoid setting a bad precendent however, for most script tools you create, you will not need this line as you will be returning back some results from R using the `arc.write()` function.

```
tool_exec <- function(in_params, out_params)
 {
  .
  .
  .
  return (out_params)
 }
```

Within the `tool_exec()` function, you can leverage **arcgisbinding** functions like `arc.env()` to check the geoprocessing environment settings of the Pro version users of your tool are working with. This enables you to check the status of settings that might impact the result of your tool. Addtionally, the `print()` function can be used to construct message window printouts for the users of your tool.

```
  env <- arc.env()
  workspace <- env$workspace

  print(workspace)
```

Required R Packages

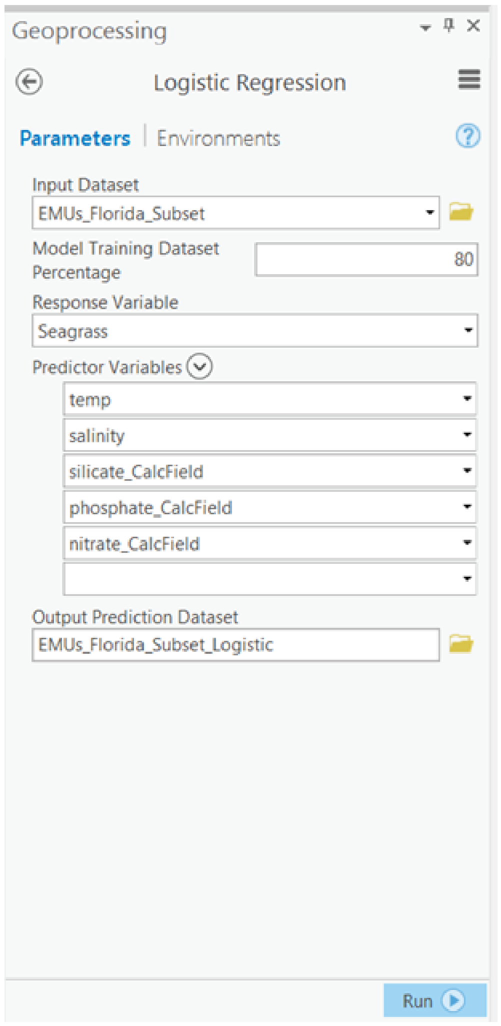
Users of your script tool will need to already have R installed on their local machines. Additionally, any R packages that your script requires, will need to be installed and loaded into their local R library. This latter step can be done directly by your script tool. By performing a check, missing packages and their dependencies can be installed through the `install.packages()` R function. All needed packages, can then be directly loaded into the user's R library so your tool can perform its analysis in ArcGIS without the user ever needing to open R.

```
  if(!requireNamespace("caret", quietly = TRUE))
    install.packages("caret", quiet = TRUE, dependencies = TRUE)

  require(caret)
```

Define Inputs and Outputs

On the script side, the input and the output parameters of your tool must be specified so they can be used in your script. These are referenced based on the order you want them to appear in your script tool's UI. The first parameter box has an index value of 1. The second parameter box has an index value of 2, and so on. For perspective, the following input and output parameter values correspond to the following script tool UI.

```
input_data <- in_params[[1]]
train_percentage_size <- (in_params[[2]])/100
dependent_variable <- in_params[[3]]
independent_variables <- in_params[[4]]

output_prediction_data <- out_params[[1]]
```



Input Dataset
EMUs_Florida_Subset
Model Training Dataset Percentage
80
Response Variable
Seagrass
Predictor Variables
temp
salinity
silicate_CalcField
phosphate_CalcField
nitrate_CalcField
Output Prediction Dataset
EMUs_Florida_Subset_Logistic

Figure 1: image

Open and Read-in Data

arcgisbinding functions like `arc.open()`, `arc.select()`, and, if needed, `arc.data2sp()` or `arc.data2sf()` are then used to bring the data from your created geoprocessing script tool user interface into R and convert it into the needed format to perform your desired R functions on it.

Here is one particular example. In this case, both the full data set and a subset of the data set are brought in for use in the script. What you choose to do will be based on the requirements of your R functions.

```r
d <- arc.open(input_data)
fields_list <- append(c(dependent_variable), independent_variables)
d_df_full <- arc.select(d)
d_df <- arc.select(d, fields = fields_list)
```

Custom R Functionality

So far, we've covered pieces from within the `tool_exec()` function that will likely be ubiqutous to all scripts wrapping R functionality. Once your desired data is inside R, what happens next and the resulting output of your tool is not limited and can be as creative as your coding. Any R function that works with data frames, spatial data frames, or rasters can be used. Additionally, any R diagnostic measures, be they statistical measures or charts, can either be printed out in your tool's messages window or produced by R when you run your tool.

Generate Results and Messages

To generate results from R functions for your tool's messages, you can make use of print statements. Here is one particular example of how this can be done.

```r
arc.progress_label("Running diagnostics on fitted model...")
arc.progress_pos(80)

#Summary of model fit
cat(paste0("\n", "............................................", "\n"))
cat(paste0("\n", "............................................", "\n"))
cat(paste0("\n"))
cat(paste0("\n", "Summary of Fitted Logistic Regression Model", "\n"))
cat(paste0("\n", "............................................", "\n"))
cat(paste0("\n"))
print(summary(d_df_train.log))

#Hosmer-Lemeshow Test
cat(paste0("\n", "............................................", "\n"))
cat(paste0("\n", "............................................", "\n"))
cat(paste0("\n"))
cat(paste0("\n", "Hosmer-Lemeshow Goodness of Fit Test Results", "\n"))
cat(paste0("\n", "............................................", "\n"))
cat(paste0("\n"))
HL <- HLgof.test(fit = fitted(d_df_train.log), obs = d_df_train$Seagrass)
print(HL)
```

Additionally, R charts can be used to further communicate results from your analysis in R. To have a chart from R pop-up when you or others run your tool, you will simply make a call to your designated chart function of choice.

Here is an example of producing an ROC curve when running a script tool designed to perform logistic regression. The R `plot()` call is all you need for this chart to appear when someone else runs your script tool from within ArcGIS.

```r
#ROC Curve
d_df_test.log.pred <- predict(d_df_train.log, newdata = d_df_test, type = 'response')
pred <- prediction(d_df_test.log.pred, d_df_test$Seagrass)
perf <- performance(pred, measure = "tpr", x.measure = "fpr")
plot(perf)
```

Finally, to return any results from R to ArcGIS, you can use the `arc.write()` function. For full documentation on the `arc.write()` function, see the Jupyter Notebook called R-bridge-reading-converting-writing-data.

```r
arc.progress_label("Writing output...")
arc.progress_pos(80)

if(!is.null(output_prediction_data) && output_prediction_data != "NA")
  arc.write(output_prediction_data, d_df_full, shape_info = arc.shapeinfo(d))
```

Example R Script Tools

Examples of R script tool can be found on the GitHub project associated with the R-ArcGIS bridge. Each one of the sample script tools comes with some associated data and documentation. These script tools can also be used as templates for your own script tool creation. Feel free to download them and to modify them as needed. If you create a script tool you would like to share with the community, we would love to help you host it. For reference, check out some of the tools created by our users, including CHANS-tools.

Components of ArcGIS Script Tool UI Creation

The customization of the user interface of your tool is constructed by the tool properties of the tool within the ArcGIS toolbox, full details of which can be found here.

Through these properties, you have complete control over how users interact with your tool and its options. You can customize the style and type of parameter box and the allowed options available for each. Parameter defaults, ranges, and drop-downs can all be tailored to the tool and the user experience you are designing.