Using Python and R Together

Keith McNulty

Key resources

- Github repo containing everything you need for this talk
- Details and tutorials on the `reticulate` package which is used to translate between R and Python

Python Environments

All Python projects need an environment where all supporting packages are installed. Virtualenv and Conda are the two most common environment management tools.

For this project you'll need a Python environment with the following packages installed: `pandas`, `scipy`, `python-pptx`, `scikit-learn`, `xgboost`.

Conda example terminal commands:

```
# create conda env and install packages
conda create --name r_and_py_models python=3.7
conda activate
conda install <list of packages>

# get environment path for use with reticulate
conda info
```

Why would someone even need to use two languages

- In general, each language has its strengths. There are things its generally easier to do in Python (eg Machine Learning), and there are things that its easier to do in R (eg, inferential statistics, tidy data).
- You may want to work primarily in one language but need specific functionality that's more easily available in the other language.
- You may have been 'handed' code in Python by someone else but you need to get it working in R.
- You don't have the time or interest to recode into a single language.

Setting up a project involving both R and Python

- Work in RStudio
- Use the `reticulate` package in R
- Point to a Python *executable* inside an environment with all the required packages by setting the `RETICULATE_PYTHON` environment variable in a `.Rprofile` file which executes at project startup. Here is what mine looks like.

```
# Force the use of a specific python environment - note that path mus
Sys.setenv(RETICULATE_PYTHON = "/Users/keithmcnulty/opt/anaconda3/env

# print a confirmation on project startup/R restart
print(paste("Python environment forced to", Sys.getenv("RETICULATE_PY
```

Ways to use Python in RStudio

1. Write a `.py` script. File > New File > Python Script
2. Code directly in the Python interpreter to test code: `reticulate::repl_python()`
3. Write an R Markdown document with R code wrapped in `{r}` and Python code wrapped in `{python}`

Exchanging objects between R and Python

Remember that you always need `reticulate` loaded:

```
library(reticulate)
```

- The `reticulate` package makes it easy to access Python objects in R and vice versa.
- If `my_python_object` is a Python object, it can be accessed in R using `py$my_python_object`.
- If `my_r_object` is an R object, it can be accessed in Python using `r.my_r_object`.

Let's create a couple of things in Python and use them in R

```
## create a dict in Python
my_dict={'team_python': ['dale', 'brenden', 'matthieu'], 'team_r': ['

## define a function in Python
def is_awesome(who: str) -> str:
  return '{x} is awesome!'.format(x=who)
```

```
my_list <- py$my_dict
str(my_list)
```

```
## List of 2
##  $ team_python: chr [1:3] "dale" "brenden" "matthieu"
##  $ team_r     : chr [1:4] "liz" "rachel" "alex" "jordan"
```

```
my_list$team_python
```

```
## [1] "dale"     "brenden"  "matthieu"
```

```
is_awesome <- py$is_awesome
is_awesome('R')
```

```
## [1] "R is awesome!"
```

Now let's do the opposite

```r
# a vector in R
my_vec <- c("data engineering", "data science")

# a function in R
unique_words <- function(string) {
  unique(unlist(strsplit(string, " ")))
}
```

```
my_list=r.my_vec
my_list
```

```
## ['data engineering', 'data science']
```

```
unique_words=r.unique_words
unique_words(my_list)
```

```
## ['data', 'engineering', 'science']
```

More details on type conversions

Type conversions

When calling into Python, R data types are automatically converted to their equivalent Python types. When values are returned from Python to R they are converted back to R types. Types are converted as follows:

R	Python	Examples
Single-element vector	Scalar	`1`, `1L`, `TRUE`, `"foo"`
Multi-element vector	List	`c(1.0, 2.0, 3.0)`, `c(1L, 2L, 3L)`
List of multiple types	Tuple	`list(1L, TRUE, "foo")`
Named list	Dict	`list(a = 1L, b = 2.0)`, `dict(x = x_data)`
Matrix/Array	NumPy ndarray	`matrix(c(1,2,3,4), nrow = 2, ncol = 2)`
Data Frame	Pandas DataFrame	`data.frame(x = c(1,2,3), y = c("a", "b", "c"))`
Function	Python function	`function(x) x + 1`
NULL, TRUE, FALSE	None, True, False	`NULL`, `TRUE`, `FALSE`

If a Python object of a custom class is returned then an R reference to that object is returned. You can call methods and access properties of the object just as if it was an instance of an R reference class.

Example Scenario 1: Editing Powerpoint

You have a simple Powerpoint document in the `templates` folder of this project called `ppt-template.pptx`. You want to automatically edit it by replacing some of the content with data from csv files for 20 different groups, creating 20 different Powerpoint documents - one for each group.

You have a function provided to you in Python which does this replacement. It is in the file `edit_pres.py` in the `python` folder of this project. However, you are not great with Python and you much prefer to manage data in R.

First, you source the Python function into your R session and take a look at the function, which is now automatically an R function:

```
source_python("python/edit_pres.py")

edit_pres
```

```
## <function edit_pres at 0x7fc2190355f0>
```

Example Scenario 1: Editing Powerpoint

The function takes five arguments, a target group name, a table of summary statistics for all groups, a specific data table for the target group, the name of the input file and the name of the output file.

Let's run the function for one group using some of the data in our `data` folder:

```
# all summary stats
chart_df <- read.csv("data/chart_df.csv")

# Group A table
table_A <- read.csv("data/table_A.csv")

input <- "templates/ppt-template.pptx"
output <- "group_A.pptx"

edit_pres("A", chart_df, table_A, input, output)
```

```
## [1] "Successfully saved version A!"
```

Example Scenario 1: Editing Powerpoint

Now we can get all of our data into a tidy dataframe:

```
library(dplyr)

# load in data files
for (file in list.files("data")) {
  splits <- strsplit(file, "\\.")
  assign(splits[[1]][1],
         read.csv(paste0("data/", file)))
}

# rowwise mutate a list column onto chart_df containing the table dat
full_data <- chart_df %>%
  rowwise() %>%
  dplyr::mutate(table = list(get(paste0("table_", group))))
```

Example Scenario 1: Editing Powerpoint

Let's look at a few rows and columns:

```
head(full_data) %>%
  dplyr::select(group, cat1_1, cat1_2, table)
```

```
## # A tibble: 6 x 4
## # Rowwise:
##   group cat1_1 cat1_2 table
##   <chr>  <dbl>  <dbl> <list>
## 1 A        5.7    2.6 <df[,5] [8 × 5]>
## 2 B        5.2    1.6 <df[,5] [8 × 5]>
## 3 C        3.6    1.3 <df[,5] [8 × 5]>
## 4 D        1.1    4.5 <df[,5] [8 × 5]>
## 5 E        3.8    3.6 <df[,5] [8 × 5]>
## 6 F        1.7    5.3 <df[,5] [8 × 5]>
```

Example Scenario 1: Editing Powerpoint

Now we can mutate our `edit_pres()` function to generate all the powerpoint in a single command.

```
# rowwise mutate the edit_pres function to generate parametrized powe
generate_ppt <- full_data %>%
  rowwise() %>%
  dplyr::mutate(
    ppt = edit_pres(group, ., table, "templates/ppt-template.pptx",
                                  paste0("report_group_", group, ".pptx
    )

# let's see what happened
head(generate_ppt) %>%
  dplyr::select(group, ppt)
```

```
## # A tibble: 6 x 2
## # Rowwise:
##   group ppt
##   <chr> <chr>
## 1 A     Successfully saved version A!
## 2 B     Successfully saved version B!
## 3 C     Successfully saved version C!
## 4 D     Successfully saved version D!
## 5 E     Successfully saved version E!
## 6 F     Successfully saved version F!
```

Example Scenario 2: Running XGBoost in R

You've been asked to train a 10-fold cross-validated XGBoost model on a set of data about wines. You want to see how accurately you can predict a high quality wine.

You have never run XGBoost before and you're not great with Python.

However, a colleague has given you a set of Python functions which they use for training XGBoost models. These functions are in `python_functions.py`. You source them into R.

```
source_python("python_functions.py")
```

Example Scenario 2: Running XGBoost in R

We create our data set by downloading the data, adding a binary 'red' wine feature and defining 'high quality' to be a quality score of 7 or more.

```
white_wines <- read.csv("https://archive.ics.uci.edu/ml/machine-learn
red_wines <- read.csv("https://archive.ics.uci.edu/ml/machine-learnin

white_wines$red <- 0
red_wines$red <- 1

wine_data <- white_wines %>%
  bind_rows(red_wines) %>%
  mutate(high_quality = ifelse(quality >= 7, 1, 0)) %>%
  select(-quality)
```

Example Scenario 2: Running XGBoost in R

If we look in the Python code, we can see that all our parameters are expected to be in a dict. In R, this means they need to be in a named list, so let's create the list of parameters we will use:

```
params <- list(
  input_cols = colnames(wine_data)[colnames(wine_data) != 'high_quali
  target_col = 'high_quality',
  test_size = 0.3,
  random_state = 123,
  subsample = (3:9)/10,
  xgb_max_depth = 3:9,
  colsample_bytree = (3:9)/10,
  xgb_min_child_weight = 1:4,
  k = 10,
  k_shuffle = TRUE,
  n_iter = 10,
  scoring = 'f1',
  error_score = 0,
  verbose = 1,
  n_jobs = -1
)
```

Example Scenario 2: Running XGBoost in R

Our first function `split_data()` expects a data frame input and will output a list of four data frames - two for training and two for testing.

```
split <- split_data(wine_data, parameters = params)

# check we got what we wanted
names(split)
```

```
## [1] "X_train" "X_test"  "y_train" "y_test"
```

Example Scenario 2: Running XGBoost in R

Our next function `scale_data()` scales the features to prepare them for XGBoost. It expects two feature dataframes for train and test and outputs a list of two scaled dataframes.

```
scaled <- scale_data(split$X_train, split$X_test)

# check we got what we wanted
names(scaled)
```

```
## [1] "X_train_scaled" "X_test_scaled"
```

Example Scenario 2: Running XGBoost in R

Next we train our XGBoost model with 10-fold cross-validation. This function expects a scaled feature dataframe, a target dataframe and some parameters.

```
# created trained model object
trained <- train_xgb_crossvalidated(
  scaled$X_train_scaled,
  split$y_train,
  parameters = params
)

# we can check that we can predict from the trained model
test_obs <- py_to_r(scaled$X_test_scaled)
trained$predict(test_obs[1:5, ])
```

```
## [1] 0 0 0 1 0
```

Example Scenario 2: Running XGBoost in R

Our last function generates a classification report - it expects a trained model, a set of test features and targets, and outputs a report dataframe:

```
generate_classification_report(
  trained,
  scaled$X_test_scaled,
  split$y_test
)
```

```
##                precision    recall  f1-score
## 0.0            0.8965937 0.9460847 0.9206746
## 1.0            0.7254902 0.5663265 0.6361032
## accuracy       0.8697436 0.8697436 0.8697436
## macro avg      0.8110419 0.7562056 0.7783889
## weighted avg   0.8621975 0.8697436 0.8634684
```

Deploying Shiny Apps that use R and Python together

- The server (eg ShinyServer or RStudioConnect) will need to have Python enabled and a Python version installed
- Your local Python version on which you built the app will need to be compatible with the one that's on the server - you can ensure this in you conda/virtualenv setup.
- If deploying from Github, when you run `rsconnect::writeManifest()` it will also create the `requirements.txt` file for your Python packages. This should be pushed to Github along with `manifest.json`
- DO NOT push `.Rprofile` to Github. This will cause deployment to fail. For safety, add `.Rprofile` to `.gitignore` if you are intending to build a deployed app.