Portfolio Project

William Tirone

Motivation

Throughout my degree, I have spent more time than necessary writing code to compare model metrics for several models in a dataframe. I have done this in Study Design, Hierarchical Models, Spatial and Time Series, Statistical Learning, and probably a few others. For example:

```
lm_1 = lm(mpg ~ cyl, data = mtcars)
lm_2 = lm(mpg ~ hp, data = mtcars)

bind_rows(data.frame(model = "lm_1",
                     MSE = mean(lm_1$residuals^2),
                     RMSE = sqrt(mean(lm_1$residuals^2))),
          data.frame(model = "lm_2",
                     MSE = mean(lm_2$residuals^2),
                     RMSE = sqrt(mean(lm_2$residuals^2)))
)
```

```
  model       MSE     RMSE
1  lm_1  9.635445 3.104101
2  lm_2 13.989822 3.740297
```

There are a few things that are inconvenient about this:

1. The model names have to be hardcoded as strings.
2. We have to calculate the MSE and RMSE by hand.
3. The code is very repetitive.

I wanted to write an R package to learn about the process of creating a package, but I also wanted to do something useful. My goal was to wrap everything above in a function to make it usable repeatedly and cut down on unnecessary code.

Scope

My first goal was to check what had been done before and already implemented in R. However, searching for code on CRAN is not trivially easy and requires a good amount of effort. There are 20,000+ packages, and it is not always easy to understand what a package does or whether or not a package does precisely what I want it to do. The closest I could find was this stackoverflow post (https://github.com/tidymodels/broom/issues/2) from the broom R package, though it looks like this feature was never implemented. Regardless, I think if it takes me at least an hour to try to find code that does something, I might as well write a package to do it. My primary resource, R Packages by Wickham and Bryan mentions the following: *"there are plenty of good reasons to make your own package, even if there is relevant prior work. The way experts got that way is by actually building things, often very basic things, and you deserve the same chance to learn by tinkering"* (Wickham and Bryan 2023). I think it is a worthy enough cause to make a package for the sake of learning how to do it, so I continued on with my idea.

I frequently switch between different programming languages like R, Python, SQL, and MATLAB. While these have very similar syntaxes, that makes them all the more difficult to switch between. I confuse indexing dataframes and vectors in R and Python continually, and I can never remember what attributes an `lm()` model has. The only syntax I can remember without any references is something like the R package `cowplot`. You can use it like this to arrange several plots on a grid (similar to facet wrapping): `cowplot::plot_grid(plot1, plot2, plot3)`. I think this is beautifully simple, and I wanted something similar for model metrics and coefficients.

Implementation

Part of the challenge of writing a function to compare different classes of models like `lm`, `glm`, or `lmer` is that they were developed by different people at different times in the development history of the language, and thus do not share the same attributes. For example, calling `names(glm)` on a `glm` model reveals that it has deviance and AIC as attributes of the model itself, but `lmer` models do not! There are similar challenges with different summary objects of the models as well. The point is, I perpetually cannot remember if the object I need is in a `summary()` object or the base model itself, and this changes across model types. To solve this, `modelfactory` detects which type of model is passed in using the base `class()` function, then pulls the appropriate metrics or values out. There are only two functions in `modelfactory`, and they do the following (these definitions are pulled from the roxygen text in the functions):

1. `modelfactory::stack_metrics` : calculates basic model metrics like MSE for the models passed in, then stacks them in a dataframe for comparison. This supports `lm`, `glm`, and `lmer` models, and different metrics are calculated for each. This does not perform

model selection based on a given criteria, but it makes the tedious task of, say, comparing R-squared across several models very easy.
2. `modelfactory::stack_coefficients` : takes several `lm` or `glm` models, pulls out their coefficients, standard errors, and confidence intervals, and stacks everything into a tibble for easy comparison across models. Note that this doesn't support `lmer` models since those may have hundreds of point estimates for random effects, so it did not seem as useful while I was developing it. Maybe I will add some form of this in the future.

I have developed a Python package, but not R. I turned to R Packages (2e), which lays out very clear steps to start building a package from scratch. Much of the structure of the package is not made by hand, but with various `devtools` and `usethis` functions. For example, `devtools::create_package()` builds a skeleton of a package that includes things like the DESCRIPTION file, R/ folder, and a handful of other files necessary for a user to install the package on their own machine.

Workflow

Here is my typical workflow:

1. Edit some code in R/
2. `devtools::load_all()` and try out the code interactively
3. Update documentation with `document()`
4. Update or add a new test, and run `test_active_file()`
5. `devtools::check()` to see if everything runs and installs smoothly
6. Commit and push changes to my GitHub repo

Sometimes in a different order, but always run `check()` at many different stages to avoid bigger problems later.

Testing

One of the most significant differences between package writing and a standard data science workflow is the existence of testing. That is, a particular file or group of files that perform some kind of tests to ensure either 1) that a bug has been successfully fixed and won't sneak back in or 2) ensure a critical part of a package works after some related or unrelated code is added. Since my primary output is a `tibble`, this is a bit challenging to write tests for. Do we want to make sure that certain functions give us the numbers we expect? Or should we check the number of rows and columns? Or do we simply want to match some pre-determined "correct" `tibble` output? I chose a combination of these.

Examples

To share examples of the package's functions with others, the `pkgdown` library makes a very nice website in just a few commands. You can view that here: https://willtirone.github.io/modelfactory/. The "Reference" tab of this website just renders the roxygen block above each function, which shows the possible arguments, a description, and code examples with output.

Here's an example of `modelfactory::stack_metrics()` to compare several `glm` models to see which models might be the best fit based on several metrics. We see that `glm_2` achieves the lowest AIC and BIC, but `glm_3` achieves the lowest deviance. This provides no statistical guarantees that `glm_2` is better than `glm_3` but it provides a starting point to examine the models more in depth.

```
glm_1 = glm(vs ~ drat + hp, data = mtcars)
glm_2 = glm(vs ~ wt + qsec, data = mtcars)
glm_3 = glm(vs ~ ., data = mtcars)

modelfactory::stack_metrics(glm_1, glm_2, glm_3)
```

```
# A tibble: 3 x 4
  model                                          deviance   AIC   BIC
  <chr>                                             <dbl> <dbl> <dbl>
1 glm(formula = vs ~ drat + hp, data = mtcars)      3.63  29.1  35.0
2 glm(formula = vs ~ wt + qsec, data = mtcars)      2.04  10.8  16.6
3 glm(formula = vs ~ ., data = mtcars)              1.58  18.6  36.2
```

Here's an example using `modelfactory::stack_coeff()` to compare point estimates, confidence intervals, and standard errors for several different `lm` models. With one line of code, we can quickly look at the different 95% C.I.'s, and see how the different covariates affect each of these.

```
lm_1 = lm(mpg ~ cyl, data = mtcars)
lm_2 = lm(mpg ~ hp, data = mtcars)
lm_3 = lm(mpg ~ disp, data = mtcars)

modelfactory::stack_coeff(lm_1, lm_2, lm_3)
```

```
# A tibble: 6 x 7
  coefficient model_name estimate std_error  p_value lower_ci upper_ci
  <chr>       <chr>         <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1 (Intercept) mpg ~ cyl   37.9      2.07    8.37e-18  33.6     42.1
```

```
2 cyl         mpg ~ cyl   -2.88     0.322   6.11e-10  -3.53    -2.22
3 (Intercept) mpg ~ hp    30.1      1.63    6.64e-18  26.8     33.4
4 hp          mpg ~ hp    -0.0682   0.0101  1.79e- 7  -0.0889  -0.0476
5 (Intercept) mpg ~ disp  29.6      1.23    3.58e-21  27.1     32.1
6 disp        mpg ~ disp  -0.0412   0.00471 9.38e-10  -0.0508  -0.0316
```

Challenges

I spent 90% of my time understanding how to build the package, and 10% of my time actually writing the main functions. Dependencies were also a bit tricky to understand initially. It is obvious to include `dplyr` as a dependency since that is used directly in the package. There are also packages like `lmer` that are not used directly in the code, but used in the roxygen blocks. In general, I wanted to avoid pinning specific versions of packages to ensure there weren't any installation issues, but I had a repeat issue with the version of the `Matrix` package installed on my machine and in the GitHub check that runs automatically when I push new code. It turns out that `Matrix` version

CRAN Submission

References

Wickham, Hadley, and Jennifer Bryan. 2023. *R Packages: Organize, Test, Document, and Share Your Code*. 2nd ed. O'REILLY MEDIA.