Seeing the Light with Refactoring and Testing

Richie Morrisroe

June 14, 2020

Data Scientists and Code

- An awful lot of us produce our results/analyses through code
- Much of this is ad-hoc investigations
- Often these investigations produce output which others want
- You create a cron job
- Congratulations, you are now supporting production code [1]

[1] in the sense that others rely on it. SQL tables are often the worst offenders (as they are often the easiest to create)

My background

- I trained as a psychologist
- I like numbers
- So I ended up as a data scientist
- At no point have I ever had a class on coding
- I haven't even had that many classes on statistics

So why are you even doing this talk?

- Because I have learned, from bitter, bitter experience that this stuff is important
- People (hopefully) make decisions as a result of our analyses
- Often, these decisions can have far-ranging impacts, most of which are impossible [2] to foresee
- As professionals, we have an obligation to make sure these results are correct
- This means we need to make sure our code is correct

[2] i.e. hard

This Talk

- I will talk about tools to improve the quality of code, and our confidence in it
- These tools are:
    - Automated testing
    - Refactoring
    - Test-Driven Development

Before Refactoring

- You must write tests
- Otherwise you [3] will introduce bugs

[3] or at least me

Types of Testing

- Individual functions: unit testing
- Processes working together: integration testing
- Overall application: functional testing
- Lots of space between these points

My Code is so awful that it can't be tested

- This is depressingly normal
- You want some kind of characteristic test
- This is often a dataframe type structure
- The quickest test is to ensure that both dataframes are equal
- There is a *wonderful* book that guides you through this

Working Effectively with Legacy Code

- The book, by Michael Feathers is really useful
- Legacy code = code without tests
- Book is based on particular problems
- And provides approaches for solving them
- Catolog of refactorings that can (theoretically) be done without tests [4]

[4] you always want at least some characterisation tests (i.e. output)

Characterisation Tests

```
require(testthat)
output_old <- readr::read_csv(
                       "older_data.csv",
                       ## you'll need this argument
                       col_types=NULL)
test_that('output old = output new',
          all.equal(output_old, output_new))
```

- capture how the system behaves right now, and make sure it doesn't change
- This simple test can allow you to make a *lot* of progress relatively quickly
- this tests are useful to help build structure and get other tests in place
- but they prevent you from improving the code in pretty much any way

Seams

- This is one of the most interesting parts of WELC, which defines a seam as a place where you can alter what code runs without changing any code
- For instance, if you have some kind of folder with different versions of code then you can create a seam by changing which folder is pointed to
- This is also a very quick way of getting characterisation tests
- The simplest example is mocking out a package/script file/module by changing the path

Pinch Points

- Another concept from WELC
- Refers to the point where you can test as much as possible of functionalities
- Normally a dataframe of some description (input data, output data, predictions etc)

First Steps

- Gather the source files you are interested in
- Run them, make sure they work.
- DO NOT SKIP THIS STEP!!!

My Example

- Its based on the irish property market data
- I have been working on and off on this for a number of years
- I have multiple folders, datasets and scripts that perform data loading, processing and modelling
- I now want to turn this into an interactive app, and keep it updated
- I aim to add tests and refactor to make it easier for me to build on top of this foundation

First Practical Step

```
require(devtools)
if (!dir.exists("ppr")) {
  usethis::create_package("ppr")
}

usethis::use_package("dplyr")
usethis::use_package("rlang")
usethis::use_package("readxl")
usethis::use_package("caret")
usethis::use_package("rgdal")
usethis::use_package("sp")
usethis::use_package("glmnet")
usethis::use_package("sf")
usethis::use_package("vtreat")
usethis::use_data_raw()
usethis::use_pipe()
```

Why a package?

- Lots of good things:
  - automated dependency management
  - easy documenting facilities (with roxygen2)
  - easy to run tests (with testthat)
  - check is just a bunch of better programming practices
- Bad things:
  - more effort
  - appeasing the mighty gods of check takes time

Next steps

- We have three scripts:
  - prep_modelling.R
  - feature_engineering.R
  - models.R

Step 0: Create a custom environment

- I use conda for this
- It supports multiple languages, and compiled dependencies
- This is pretty awesome for python, and its a little more light-weight than Docker
- It supports R (and apparently loads of other languages too)

```
conda create -n PPR-r r=3.6.3
conda env export --name PPR-r > PPR-r.yml
```

First, generate or create ground truth data

- Our scripts need to be run in order
- So, we can use the boundaries between them as points where we can introduce test points
- However, our scripts suck and therefore we don't have any convenient boundaries available

```
rm(list=ls())
source("scripts/prep_modelling.R")
```

- Simplest way to run script
- We can then look at what it produces and decide what to test
- Biggest benefit here is not having to change anything

Assessing what we need

```
ls()
```

- We have a bunch of dataframes, some functions and a few intermediate results
- However, the best pinch point is ppr_train and test, as they are downstream of all our feature engineering

Add data to package

```
ppr <- readxl::read_excel("~/Dropbox/PPR/PPR-ALL.xlsx"
set.seed(49)
ppr_sample <- dplyr::sample_frac(ppr, size = 0.05)
ppr <- ppr_sample
usethis::use_data(ppr, overwrite = TRUE)
```

- This is important for running the tests
- You *can* use absolute paths and stuff, but it's pretty ugly
- CRAN strongly suggests that data be less than 5Mb, so we take a 5% sample

Create pinch point

- We need to run our script with the package data, so we can keep everything consistent
- We add all the scripts to inst/, which is where we'll put the test data also

```
readr::write_csv(ppr_train2,
                 path = "prep_modelling_output_old.csv
```

- We put this at the end of our script
- Then, we run it like so:

```
cd ppr/inst/
Rscript prep_modelling.R
```

- We then output the full file
- However, because we took a 5% sample earlier, we need to change things

Handling the sample

- We have the data in our package [5]
- We need to change our script to use the package data
- We can either rename the script, or rely on version control

```
cp prep_modelling.R prep_modelling_refactor.R
```

- Then we rename our output file

```
## readr::write_csv(ppr_train2,
## path = "prep_modelling_output_old.csv")
readr::write_csv(ppr_train2,
                 path = "prep_modelling_output_refacto
```

[5] right now, it's the *only* thing in the package

Commented out code is bad, right?

- In general commented out code is a massive anti-pattern
- But right now, we are making minimal changes to ensure that we can safely update our script

Import data from package

```
## library(tidyverse)
## library(readxl)
## ppr <- read_excel("~/Dropbox/PPR/PPR-ALL.xlsx",
##                    sheet = "PPR-ALL")
require(ppr)
data(ppr)
names(ppr)
```

Add Test Directories

```
mkdir ppr/tests
mkdir ppr/tests/testthat/
touch ppr/tests/testthat/test_first.R
touch ppr/tests/testthat/test_integration_prep_modelli
```

- You can do this with usethis, but it was borked for me when I tried, so I did it manually

Integration Test

```
  context("integration test prep_modelling")
old <- readr::read_csv("../../inst/prep_modelling_outp
new <- readr::read_csv("../../inst/prep_modelling_outp
test_that(
        "data.frame outputs are equal",
        expect_equal(old, new)
)
```

Run the script again with the changes we've made

- Commit your code before and after!
- We actually needed to make the data changes *first* as otherwise our "old" data wouldn't match the new(er) data

Create a small script to run the integration tests

- This talk is all about automated testing, so we'll write a script

```
system("Rscript prep_modelling.R")
testthat::test_file(
            "../tests/testthat/test_integration_prep_m
```

Move existing functions to file

- Move the functions we have to a file
- Change our script to load the functions from this file
- We now have a package with code in it
- Now we run our integration test again
- Next step is to add some unit tests

Write some tests

```
require(testthat)
context("load data")
dat  <- load_data("~/Dropbox/PPR/PPR-ALL.xlsx")
test_that('load_data returns a tibble',
  {expect_equal(
            class(dat)[1], "tbl_df")})
```

- The first thing we do is write a load_data function
- this allows us to later abstract how/where we load the data from without needing to change (as much code)
- Note that this function doesn't exist yet
- Let's write it

Loading Data

```
load_data <- function(path) {
  ppr <- readxl::read_excel(path, sheet = "PPR-ALL")
  return(ppr)
  }
```

- This wouldn't normally make a whole lot of sense
- But I know that I am probably going to move to SQL storage with this project
- So centralising the data loading does make sense [6]

[6] given my use-cases

Test Names

```
finished_names <-
  c("date_of_sale_dd_mm_yyyy",
    "address", "postal_code", "county",
    "price", "not_full_market_price","vat_exclusive",
    "description_of_property",
    "property_size_description") %>%
  as.character()

test_that('normalise_names works',
{expect_equal(finished_names,
             names(normalise_names(dat)))
})
```

- We just grab the output of normalise_names, and test that
- this is a crummy test; try to test the actual invariants on edge cases

Test Fix Price

```
test_that('fix price returns the correct price',
          expect_equal(3e6, fix_price("3,000,000")))
```

- All good
- However, I think I spotted a bug in fix_price

Fix Price Bug

```
fix_price <- function(x) {
        nopunct <- gsub(",|\\.", "", x = x)
        nums <- as.numeric(
                iconv(nopunct, "latin1",
                        "ASCII",
                        sub = ""
                )
        )
}
```

- Can you spot it?

Test Case for Bug

```
test_that('fix price returns the correct price',
          expect_equal(3000000.01,
                       fix_price("3,000,000.01")))
```

- We actually shouldn't strip the dots, as they are valid in numbers
- We have a failing test, so we can fix this code (and make sure it stays fixed, which is normally more valuable)

Fix Price Fix

```
fix_price <- function(x) {
        nopunct <- gsub(",", "", x = x)
        nums <- as.numeric(
                iconv(nopunct, "latin1",
                        "ASCII",
                        sub = ""
                )
        )
}
```

- We just remove the part of the gsub call that matches dots
- Now all our tests pass

Generate Bootstrap Results

- I'll leave testing this for now, as I probably need to do the feature normalisation first

Break out components

- You want to break out chunks of code into manageable (well-named) sizes
- This is the extract function refactoring (probably the most common)

```
ppr3 <- mutate(ppr2,
               price = price / 100,
               is_big = ifelse(price >= 2e6,
                               "Big",
                               "Not Big"))
```

Write test for new function

```
test_df_output  <- readr::read_csv("../../inst/mark_va
test_df_input  <- readr::read_csv("../../inst/mark_val
test_that('mark values as large works', {
  expect_equal(mark_values_as_large(test_df_input,
                                    large = 1e6),
               test_df_output)
})
```

Write the actual function

```
mark_values_as_large <- function(df, large) {
  large <- rlang::enquo(large)
  ppr3 <- dplyr::mutate(df,
                        is_big =
                          ifelse(
                            .data$price >= !!large,
                            "Big", "Not Big"))
  return(ppr3)
  }
```

- By which I mean copying code into a function and adding a return statement
- This is the extract function refactoring

Opposite but Equal

- Sometimes, we actually want the function to be inlined
- This is called inline function

Convert price to log

- Logs are great
- In general, if you have problems with a response or predictor variable, you should log it and see if that helps

```
ppr4 <- mutate(ppr3, log_price =
                       log(price, base = 10))
```

- Does this need to be a function?

Log function tests

```
test_that('log(price)< price',
{
  data(ppr)
  ppr2 <- normalise_names(ppr) %>%
    dplyr::mutate(price=fix_price(price))

  new_df <-  log_column(ppr2, price)
  # can't find a vector based expectation in testthat
  expect_lt(new_df$log_price[1],
                  ppr2$price[1])}
        )
```

- Obviously this fails again
- One of the great things about TDD is that you end up needing to use your API somewhere before you write it
- This provides both documentation and a usability check

Log Column Function

- Again, we need to do the pointless tidyverse dance because Hadley hates quotes

```
log_column <- function(df, col) {
    col <- rlang::enquo(col)
    res <- dplyr::mutate(df,
                         log_price =
                           log(!!col, base = 10))
    return(res)
    }
```

- This is beginning to annoy me a little less, from repetition

What not to functionalise

- Exploratory analysis
- There's loads of that in my scripts, for example

```
post_codes_table <-
        with(ppr4, table(postal_code, useNA = "always"
        as.data.frame() %>%
        arrange(desc(Freq))
head(post_codes_table, n = 12)
```

- This is to figure out what's going on
- Not sure it makes a lot of sense here
- If I'm doing the same stuff on a bunch of columns, a function is useful, but otherwise it doesn't make much sense

Next Steps

```
ppr5 <- mutate(ppr4,
        is_full_market_price = ifelse(
                not_full_market_price == "No",
                "Yes",
                "No"
        )
) %>%
        select(-not_full_market_price) # remove old va
```

- So this is only done once, and is pretty specific
- But it handles some ugly, ugly names
- This is a rename field refactoring (which can be incredibly impactful)

Rename Field Test

- First, we write a test

```
data(ppr)
test_that('we have is_full_market_price column', {
          ppr3 <- normalise_names(ppr) %>%
            dplyr::mutate(price=fix_price(price)) %>%
            log_column(price)
          ppr4 <- invert_field(ppr3,
                               not_full_market_price)
          expect_equal(names(ppr4)[length(ppr4)],
                       "is_full_market_price") }
          )
```

- Which obviously fails
- I did reconsider my name after the failure (the original name was fix_field_names, but that was needlessly vague)

Invert Field Names

```
invert_field <- function(df, field) {
  field  <- rlang::enquo(field)
  ppr5 <- dplyr::mutate(df,
             is_full_market_price = ifelse(
                      !!field == "No",
                      "Yes",
                      "No"
             )
             ) %>%
    dplyr::select(-not_full_market_price)
  return(ppr5)
}
```

- Remember that once we have tests on everything, we can refactor much more fearlessly

Description of property field

- When I wrote this code (for a data science bootcamp) I discovered that this field is buggy and annoying
- There's also a bunch of entries as Gaeilge, which is annoying for my purposes
- I have about 30 lines of logic here, most of which won't be reusable right away.
- The easiest thing to do with it is just dump it all into a function so we can test it

Property Description Function

```
  fix_property_description  <- function(df) {
fix_new <- mutate(df,
        description_of_property =
                ifelse(
                        grepl("Nua", x = description_o
                        "New Dwelling house /Apartment
                        ifelse(
                                grepl("Ath", x = descr
                                "Second-Hand Dwelling 
                                ## nested ifelse are p
                                description_of_propert
                        )
                )
)

remove_irish <- dplyr::mutate(fix_new, prop_descriptio
```

Testing our Property Description Logic

- Kinda painful
- Let's punt on it and just ensure that the same data is output
- This is something that's a really, really common occurence [7]

[7] especially with code you didn't write yourself

Creating base data

Alter our prep_modelling script like so:

```
readr::write_csv(ppr10,
                 path = "ppr_data_cleaning_done.csv")
```

- This will write the file into our package

Data Splitting

- At this point, we have created functions for each of our data cleaning steps
- The next section is handling test and train sets

Data Splitting Test

```
data(ppr)
ppr_for_split  <- normalise_names(ppr) %>%
  dplyr::mutate(price=fix_price(price)) %>%
  mark_values_as_large(1e6) %>%
  log_column(price) %>%
  invert_field(not_full_market_price) %>%
  fix_property_description()
test_that('split_data returns a list',
          expect_is(split_data(ppr_for_split), 'list')
```

- Start with a simple test, make them more specific as times goes on

Data Splitting Function

```
split_data <- function(df) {
  return(list())
  }
```

- Passes the test :)

Splitting Data

- Dump all the code into a function
- Passes the tests

```
split_data <- function(df) {
  set.seed(34)
  ppr_train_indices <- with(
    df,
    caret::createDataPartition(log_price,
                               times = 1,
                               p = 0.7,
                               list = FALSE
                               )
  ) %>% as.vector() #because tibble sucks
  ppr_train <- df[ppr_train_indices, ]

  ppr_not_train <- df[-ppr_train_indices, ]
  ppr_test_indices <- with(
```

Add More Tests

```
test_that('split data has test and train',
          expect_equal(names(
            split_data(ppr_for_split)),
                       c("train", "test")))

test_that('split data returns a train tibble',
          expect_is(
            split_data(ppr_for_split)$train[1],
            'tbl_df'))
```

Yet More Tests

```
  test_that('split data returns a test tibble',
            expect_is(
              split_data(ppr_for_split)$test[1],
              'tbl_df'))

test_that('split_data train has less rows than input',
          train  <- split_data(ppr_for_split)$train
          expect_gt(
            nrow(ppr_for_split),
            nrow(train))}
          )
```

Feature Engineering

- We now move on to the next script
- First thing we do is add a write_csv call to the end of our original function

```
readr::write_csv(test_vtreat_full,
                 path="feature_eng_results_old.csv")
```

Feature Engineering Integration Test

```
  context("integration test feature engineering")
old <- readr::read_csv("../../inst/feature_eng_results
new <- readr::read_csv("../../inst/feature_eng_results
test_that(
        "data.frame outputs are equal",
        expect_equal(old, new)
)
```

Analysing what the script does

- Takes the data from the previous script
- fits a simple glmnet model
- loads in geocoded register
- Loads in more data (ppr plus pobal deprivation data)
- fits a PCA
- fit a linear model on electoral district
- loads vtreat
- applies vtreat functionality to df
- outputs model matrix [8] from vtreat

[8] it's actually a df, but it is very very similar to a model matrix

Cleaning it up

- We'll focus on the pathway to the vtreat model matrix
- Ignore the PCA and the simple models for now

Create More Datasets

- We need to load in a sample of our geocoded and pobal data
- Again, this makes it way easier and more reproducible to run the tests

```
ppr_gc <- read_csv(
        "~/Dropbox/PPR/ppr_geocoded_till_oct2018.csv"
)
set.seed(49)
ppr_sample <- dplyr::sample_frac(ppr_gc, size = 0.05)
ppr_gc <- ppr_sample
usethis::use_data(ppr_gc, overwrite = TRUE)
```

Pobal Data

```
ppr_pobal <- readRDS("~/Dropbox/Code/DDS/ppr_sf_pobal2
set.seed(49)
ppr_pobal_sample <- dplyr::sample_frac(ppr_pobal, size
ppr_pobal <- ppr_pobal_sample
usethis::use_data(ppr_pobal, overwrite = TRUE)
```

Run these scripts

- Weirdly, despite these being in the package directory, nothing is done unless you execute the scripts into a running process
- So make sure you do that
- You also need to document your datasets, but we'll avoid that for now
- Note that you'll need to update the integration test after adding data to the package
- Also note that sampling breaks the integration tests, unless you use set.seed

Write Some Tests

Vtreat Pipeline

```
generate_model_matrix <- function(df, varlist,
                                  outcomename,
                                  calibration_size) {
  df_calibration <- dplyr::sample_frac(df, size=calibr
  df_train  <- dplyr::anti_join(df, df_calibration)
  tf <- designTreatmentsN(df,
                          varlist = varlist,
                          outcomename = outcomename
                          )
  result <- prepare(tf,
                    dframe = df_train
                    )
  return(result)
}
```

Subtleties and Problems

- Sampling will break lots of tests given the approach we're using
- You can either set.seed or reduce the input data outside

Different Approach to Refactoring

- Instead of abstracting out units of work, abstract out common tasks

```
sapply(ppr_pobal, n_distinct) %>%
        as.data.frame() %>%
        rownames_to_column() %>%
        arrange(desc(‘.‘))
```

- This code is unique
- But the pattern is common
- This is a lot more useful than the original
- It's also a lot more amenable to testing
- As always, we start with a characterisation test

More Extractions

```
get_max_and_min  <- select_if(as.data.frame(ppr),
                                      is.numeric) %>%
        sapply(., function(x) {
                           data.frame(
                                      min = min(x, na.rm = TRUE)
                                      max = max(x, na.rm = TRUE)
                           )
                }) %>%
        as.data.frame()
```

Becomes

```
get_max_and_min <- function(df) {
  maxmin  <- dplyr::select_if(as.data.frame(df),
                                              is.numeric) %>%
    sapply(., function(x) {
      data.frame(
        min = min(x, na.rm = TRUE),
        max = max(x, na.rm = TRUE)
      )
    }) %>%
    tibble::as_tibble() %>%
    tibble::rownames_to_column()

  max_min_df  <- tidyr::unnest(maxmin,
                                      cols=colnames(maxmin)
    as.data.frame()
  return(max_min_df)
```

Value Counts

- This is *literally* the only thing pandas does better than R
- I write code like the below far too often for my taste

```
post_codes_table <-
  with(ppr4, table(postal_code,
                   useNA = "always")) %>%
        as.data.frame() %>%
        arrange(desc(Freq))
```

Which can be functionalised like so

```
value_counts <- function(df, col) {
  result <-
    table(eval(substitute(col), envir=df),
          useNA = "always") %>%
    as.data.frame() %>%
    arrange(desc(Freq))

}
```

Proportion of Missings

- Again, I have written code like this many, many times

```
count_proportion_missing <- function(df) {
  result <- sapply(df, function(x) {
              sum(is.na(x)) / length(x)
      }) %>%
        as.data.frame() %>%
        tibble::rownames_to_column() %>%
        dplyr::arrange(desc('.')) %>%
    dplyr::mutate_if(is.numeric, round, 4)
  return(result)
  }
```

I don't actually need these functions to run the script

- But I can add the data from the originals as test data to get started
- And thus ensure that my changes don't break anything
- To be honest, the last few functions are probably the most useful thing in my package right now
- Certainly, they are the most re-usable
- They do need some tests

Lazy Man's approach to testing

- This is my philosophy
- Minimum testing for maximum output
- We add a few write_csv calls for each of our outputs
- And use that data for testing

Write a function to make this easier

- We have lots of annoying relative paths in our test data loading
- let's abstract this away from our concerns

```
load_test_data <- function(name) {
  bpath  <- "../../inst/"
  result <- readr::read_csv(paste0(bpath, name))
  return(result)
}
```

Tests for our Data Analysis Functions

```
require(ppr)
require(testthat)
count_distincts_old <- load_test_data("count_distincts
test_that('count_distinct old and new are equal',
          expect_equal(count_distinct_values(ppr_pobal
                                             count_distincts_old))
```

- And it fails [9]

[9] always make sure your tests fail when they're supposed to

More Tests

```
maxmin_old <- load_test_data("get_max_and_min_test_dat
test_that('get_max_and_min old and new are equal',
          expect_equal(get_max_and_min(ppr_pobal),
                       maxmin_old))
```

Yet More Tests

```
missings_old <- load_test_data("count_proportion_missi
test_that('count_prop_missings old and new are equal',
          expect_equal(count_proportion_missing(ppr_po
                       missings_old))
```

Further Work

- I haven't covered that much refactoring
- I haven't covered that much TDD
- I haven't covered that much working with legacy code
- But hopefully I have given an introduction to them