Dependable Development of Statistical Models

How I once wrote code and used it again!

What We Will Discuss

- A useful workflow for getting from an idea to production grade code that can be reused in the future

What We Will Discuss

- A useful workflow for getting from an idea to production grade code that can be reused in the future
- How to write code such that we can use it in the future
    - Coding style
    - Creating functions sensibly
    - Writing code that can be easily verified as correct

What We Will Discuss

- A useful workflow for getting from an idea to production grade code that can be reused in the future
- How to write code such that we can use it in the future
  - Coding style
  - Creating functions sensibly
  - Writing code that can be easily verified as correct
- Ongoing code maintenance concerns and solutions

What We Will NOT Discuss

- Specifics of R and C++ syntax

What We Will Not Discuss

- Specifics of R and C++ syntax
- Concerns with the model formulation

What We Will NOT Discuss

- Specifics of R and C++ syntax
- Concerns with the model formulation
- How to actually make money with our model

Glossary of Terms

Finance Related

- Exchange Traded Fund (ETF) - A marketable security that tracks the value of a basket of assets
- Sector - For our purposes, one of the subdivisions of member stocks in the S&P 500 Index (Financials, Energy, etc.)
- Price - for our purposes today, this will be the final price an asset traded at within a 1 minute period (we will also call it Close).
- Forward Price - a Price at some time in the future

Glossary of Terms

Finance Related (cont.)

- Return - a measure of the current Price of an asset against some historical representation of the Price of the same asset

Common Workflow

- Create a model to predict the price of the SPDR S&P 500 ETF (SPY) based on the price fluctuations of the different sector ETFs
  - Create R code to fit this model
    - Proof of concept (messy and hard to reuse)
    - Production (nice, neat, reusable for future tasks)
- Create C++ code to enact this model in production
- Validate that our research and production code are correct
- Profit!!

A short note on coding style

- Coding style is uninteresting
- Coding style is important
- CONSISTENCY is everything
- Find something you are comfortable with (or is mandated) and get used to it!

Examples of Coding Style

```
# BAD BAD BAD...inconsistent and hard to read
some_function <- function(parameterOne, p_2, sParameter3) {
  if ( parameterOne )
  {
     return(paste0("prefix-", parameterOne, p_2, sParameter3))
  }
  else {
    return(paste(parameterOne, p_2, sParameter3))
  }
}

someOtherFunction <- function(p1, p2, iParam3)
{
  symbols <- c( 'SPY', 'XLB', 'XLE', 'XLF', 'XLI', 'XLK', 'XLP', 'XLU', 'XLV',
                'XLY', 'XME' )
  data.frame(symbol=p1, price = p1, description =p2, enabled=iParam3 )
}
```

Examples of Coding Style

```
# FINE
someFunction <- function(parameterOne, parameterTwo, parameterThree) {
  if (parameterOne) {
    return(paste0("prefix-", parameterOne, parameterTwo, parameterThree))
  } else {
    return(paste0(parameterOne, parameterTwo, parameterThree))
  }
}

someOtherFunction <- function(parameterOne, parameterTwo, parameterThree) {
  symbols <- c('SPY', 'XLB', 'XLE', 'XLF', 'XLI', 'XLK', 'XLP', 'XLU', 'XLV',
               'XLY', 'XME')
  data.frame(symbol=symbols, price=p1, description=p2, enabled=iParam3)
}
```

Examples of Coding Style

```
# ALSO FINE
some_function <- function(parameter_1, parameter_2, parameter_3)
{
  if (parameter_1)
  {
    return(paste0("prefix-", parameter_1, parameter_2, parameter_3))
  }
  else
  {
    return(paste0(parameter_1, parameter_2, parameter_3))
  }
}

some_other_function <- function(parameter_1, parameter_2, parameter_3)
{
  symbols <- c('SPY', 'XLB', 'XLE', 'XLF', 'XLI', 'XLK', 'XLP', 'XLU', 'XLV',
               'XLY', 'XME')
  data.frame(symbol=symbols, price=p1, description=p2, enabled=iParam3)
}
```

A few things I've found useful

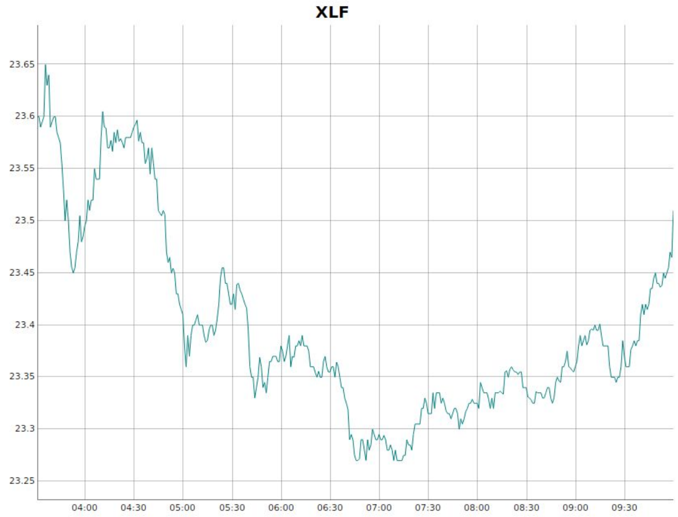
Optional (but suggested) code style guidelines

- No lines longer than 80 characters (easier to print)
- Use vertical whitespace to separate grouped code
- Name functions and variables consistently and sensibly

Now onto the fun stuff...

Stock Market Data

- All models are built upon underlying data. In this case, we will use prices from the market.
- We will focus on using data from the exchange traded funds (ETFs) related to the S&P 500.
- In particular, we will use 1 minute Open/High/Low/Close (OHLC) bars.



The Raw Data (.csv files)

```
"datetime","open","high","low","close","numEvents","volume","value"
2017-01-03 09:31:00,225.04,225.12,224.93,224.95,2712,1229453,276682272
2017-01-03 09:32:00,224.95,224.96,224.83,224.86,2175,621127,139678688
2017-01-03 09:33:00,224.86,224.92,224.84,224.87,1190,308687,69416944
2017-01-03 09:34:00,224.86,225,224.85,225,1068,255857,57551336
2017-01-03 09:35:00,225,225.15,224.99,225.11,1233,351547,79125000
2017-01-03 09:36:00,225.13,225.26,225.12,225.19,2165,634769,142956656
2017-01-03 09:37:00,225.19,225.265,225.12,225.14,1685,402474,90631768
2017-01-03 09:38:00,225.14,225.15,225.07,225.12,1079,271116,61032304
2017-01-03 09:39:00,225.115,225.12,224.86,224.89,1906,520833,117174728
2017-01-03 09:40:00,224.88,224.89,224.8,224.87,1347,337532,75891936
2017-01-03 09:41:00,224.86,224.97,224.84,224.9,1730,397894,89489328
2017-01-03 09:42:00,224.9,224.99,224.9,224.91,1302,293458,66015368
2017-01-03 09:43:00,224.92,224.92,224.77,224.86,1481,365204,82114144
2017-01-03 09:44:00,224.855,224.925,224.82,224.92,1132,301653,67831712
2017-01-03 09:45:00,224.92,224.97,224.85,224.97,1217,266945,60040516
2017-01-03 09:46:00,224.97,225.01,224.91,224.99,1024,256305,57660492
2017-01-03 09:47:00,224.99,225.05,224.97,224.99,726,170482,38359956
2017-01-03 09:48:00,224.98,224.98,224.82,224.89,862,199235,44808316
2017-01-03 09:49:00,224.89,224.905,224.76,224.87,850,263500,59243220
2017-01-03 09:50:00,224.87,224.96,224.84,224.85,529,123766,27835992
2017-01-03 09:51:00,224.84,224.85,224.74,224.76,726,195509,43945856
2017-01-03 09:52:00,224.77,224.82,224.71,224.77,939,230703,51854688
2017-01-03 09:53:00,224.77,224.85,224.77,224.82,690,180088,40485960
2017-01-03 09:54:00,224.8199,224.8999,224.7823,224.8,1004,261874,58879880
```

Outline of Proposed Model

- Simple linear model to predict the forward price of SPY (SPDR S&P 500 ETF) using returns in the sector ETFs.
- Use 1 minute Open/High/Low/Close (OHLC) bars.
- Factors will be the close price of an asset minus an exponential moving average of the close price of the same asset.
- We will try to predict the difference between the close at time $t + 10$ and the close at time t.

Basic Model Outline

```r
library(xts)

symbols <- c('SPY', 'XLB', 'XLE', 'XLF', 'XLI', 'XLK', 'XLP', 'XLU', 'XLV',
             'XLY', 'XME')
dates <- c('2017-01-03', '2017-01-04', '2017-01-05', '2017-01-06', '2017-01-09',
           '2017-01-10', '2017-01-11', '2017-01-12', '2017-01-13', '2017-01-17',
           '2017-01-18', '2017-01-19', '2017-01-20')

all_data <- NULL

# Fill all_data with some data (columns will be of the form close.SYMBOL,
# ema.SYMBOL, and forward.SYMBOL).

...
```

Basic Model Outline

```r
# Now we have all the data we need

formula <- 'I(forward.SPY - close.SPY) ~ '

for (symbol in symbols[1:length(symbols)]) {
  formula <- paste0(formula, 'I(close.', symbol, ' - ema.', symbol, ') + ')
}

formula <- paste0(formula, ' - 1')

result <- lm(formula, data=all_data)
```

A First Pass

We will:

- Figure out how to get the data we need
- Calculate the factors we want
- Fit our model
- Determine if there is any validity to the model

A First Pass

```
# Loop over the dates
for (d in dates) {
  data <- NULL

  # Loop over the symbols
  for (symbol in symbols) {

    # Read the raw data
    filename <- paste0("~/Downloads/data/", symbol, "/", d, ".csv")
    a <- read.csv(filename, header=TRUE)
    alpha <- .05
```

A First Pass

```
# Sample the data
a$datetime <- as.POSIXct(a$datetime, tz="America/NewYork")
a <- xts(a[, 'close'], order.by=a$datetime)
start <- as.POSIXct(paste0(d, ' ', '09:31:00'), tz="America/NewYork")
end <- as.POSIXct(paste0(d, ' ', '16:00:00'), tz="America/NewYork")
idx <- seq(from=start, to=end, by=60)
idx <- xts(rep(NA, length(idx)), order.by=idx)
a <- na.locf(merge(idx, a))[index(idx)]
colnames(a) <- c('dummy', 'close')
a <- cbind(a, NA)
ema_col <- paste0('ema.', symbol)
colnames(a)[ncol(a)] <- ema_col
```

A First Pass

```
# Calculate the EMAs
ema <- NULL

for (i in 1:dim(a)[1]) {
  if (is.null(ema)) {
    ema <- as.numeric(a[i, 'close'])
  } else {
    ema <- as.numeric((1 - alpha) * ema + alpha * a[i, 'close'])
  }
  a[i, ema_col] <- ema
}
```

A First Pass

```r
# Calculate the forwards
a <- cbind(a, NA)
forward_col <- paste0('forward.', symbol)
colnames(a)[ncol(a)] <- forward_col

for (i in 1:dim(a)[1]) {
  if (i + 10 <= dim(a)[1]) {
    a[i, forward_col] <- as.numeric(a[i + 10, 'close'])
  }
}
```

A First Pass

```
    # Add the data columns for this symbol
    if (is.null(data)) {
      data <- a[, -1]
      colnames(data) <- c(paste0('close.', symbol), ema_col, forward_col)
    } else {
      colnames(a) <- c('dummy', paste0('close.', symbol), ema_col, forward_col)
      data <- merge(data, a[, -1])
    }
  } # end loop over symbols
```

A First Pass

```
  # Append the day's data to all_data
  if (is.null(all_data)) {
    all_data <- data
  } else {
    all_data <- rbind(all_data, data)
  }
} # end loop over dates
```

A First Pass (results)

```
Coefficients:
                              Estimate Std. Error t value Pr(>|t|)
I(close.SPY - close.SPY.ema) -0.301167   0.163513  -1.842 0.065556 .
I(close.XLB - close.XLB.ema) -0.001551   0.071220  -0.022 0.982632
I(close.XLE - close.XLE.ema)  0.158330   0.047023   3.367 0.000766 ***
I(close.XLF - close.XLF.ema)  0.118678   0.249504   0.476 0.634342
I(close.XLI - close.XLI.ema)  0.292684   0.089616   3.266 0.001098 **
I(close.XLK - close.XLK.ema)  0.149698   0.190574   0.786 0.432193
I(close.XLP - close.XLP.ema) -0.524456   0.108113  -4.851 1.27e-06 ***
I(close.XLU - close.XLU.ema)  0.209653   0.057194   3.666 0.000249 ***
I(close.XLV - close.XLV.ema)  0.076743   0.077826   0.986 0.324141
I(close.XLY - close.XLY.ema)  0.142541   0.077298   1.844 0.065238 .
I(close.XME - close.XME.ema) -0.159269   0.034794  -4.577 4.82e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Multiple R-squared:  0.02583,    Adjusted R-squared:  0.02365
F-statistic: 11.88 on 11 and 4929 DF,  p-value: < 2.2e-16
```

A First Pass

What we did:

- Lots of scratch work
- No optimization or design
- JUST MADE IT WORK

A First Pass

What we got:

- Sloppy code
- Nearly impossible to modify code
- Proof of concept

A Second Pass

This time we will:

- Make some sensible functions
- Name our variables reasonably
- Get shorter “main” code

A Second Pass

```
# Gets the raw data and adds EMAs for a single symbol
get_symbol_data <- function(symbol, date) { … }

# Gets the raw data and EMAs for a single date for multiple symbols.  Calls
# get_symbol_data() in a loop.
get_day_data <- function(date, symbols) { … }

# Gets all the data we need to fit our model for all the dates in question.
# Calls get_day_data() in a loop.
get_all_data <- function(dates, symbols) { … }

# Gets all the data (using get_all_data()), creates the formula, and emits the
# fit.
fit_model <- function(symbol, driver_symbols, dates) { … }
```

A Second Pass

```
symbols <- c('SPY', 'XLB', 'XLE', 'XLF', 'XLI', 'XLK', 'XLP', 'XLU', 'XLV',
             'XLY', 'XME')
dates <- c('2017-01-03', '2017-01-04', '2017-01-05', '2017-01-06', '2017-01-09',
           '2017-01-10', '2017-01-11', '2017-01-12', '2017-01-13', '2017-01-17',
           '2017-01-18', '2017-01-19', '2017-01-20')

result <- fit_model('SPY', symbols, dates)
```

A Second Pass

What we did:

- Created functions
- Reduced the complexity of the “main” code we need to write
- Created a reusable design (maybe?)

A Second Pass

What we got:

- Cleaner code
- Reasonable functions
- Ability to reuse for identical models on different symbol sets
- Really simple “main” code

What We Really Want

Desires:

- Access to our model data
- A framework for building data sets and adding new features
- A way to fit different models easily

Implementation:

- Separate data acquisition from feature calculation
- Create a suite of feature generators
- Create formulas from data sets and fit them however we want

A Third Pass (this time for sure!)

This time we will:

- Think about separating concerns
- Make more generalized functions
- Think about the big picture (new features, different types of fits)
- Come out with a solution that we can use in the future

A Third Pass

```
# Retrieves data from the specified directory for the symbol and date given.
# indicates which column we want to use as our timestamp and which data columns
# we want to see.
get_raw_symbol_data <- function(symbol, date, timestamp_col, data_cols,
  base_dir) { … }

# Creates minute samples of an arbitrary data set.
sample_equity_data_minute <- function(data, date) { … }

# Calculates an EMA of the column given assuming that it is decayed at every
# event (our sampled frequency) by the given alpha and adds a column to the
# data set containing that EMA (called [col].ema)
calculate_ema <- function(data, col, alpha) { … }
```

A Third Pass

```
# Calculates a forward of the column given.  The forward value is taken n
# observations forward from the current row and stored in the data set as
# [col].forward
calculate_forward <- function(data, col, n) { ... }

# Gets the data and calculates the features (EMA) and forwards for each symbol
# over the date set.  This is specific to my model.
my_model_get_data <- function(symbols, dates, alpha, lag, data_dir) { ... }

# Given data generated by my_model_get_data(), run the appropriate fit for the
# given symbol and driver_symbols.
my_model_fit <- function(symbol, driver_symbols, data) { ... }
```

A Third Pass

```r
symbols <- c('SPY', 'XLB', 'XLE', 'XLF', 'XLI', 'XLK', 'XLP', 'XLU', 'XLV',
             'XLY', 'XME')
dates <- c('2017-01-03', '2017-01-04', '2017-01-05', '2017-01-06', '2017-01-09',
           '2017-01-10', '2017-01-11', '2017-01-12', '2017-01-13', '2017-01-17',
           '2017-01-18', '2017-01-19', '2017-01-20')
alpha <- .05
n <- 10

data <- my_model_get_data(symbols, dates, alpha, n, '~/Downloads/data/')
result <- my_model_fit('SPY', symbols, data)
result_2 <- my_model_fit('XLF', symbols, data)
```

A Third Pass

What we did:

- Created functions that retrieve data
- Created functions that create features
- Separated data acquisition and feature generation from the actual fit

What we got:

- Slightly more complex “main” code
- Access to our data
- Ability to generate new kinds of features easily
- Reusable code (data acquisition and features)

From Here...

- Move the general purpose (data and feature) functions into their own R packages for even better reuse.
- Move the model specific functions (prefixed by my_model) into their own package.
- Generate short, and readable production scripts (the “main” sections) that can operate using these packages.
- Conceive new models based on these packages, and when you’re done with your proof of concept work, put them in their own packages!

We’re Done, Right?

NO! Research is typically only one half of the battle in the real world. We also need to put our model to work.

- This is often not the concern of the researcher, however, it pays to understand the process and make things work together well.

Interfacing with Production

We will need to:

- Be sure the production implementation is identical to our research implementation
- Be able to make changes to either research or production and reflect them in the other
- Be nice to the production developers!

Production Implementations

A few simple rules:

- When building predictors, make them as simple as possible
  - The model is only math...make sure the implementation reflects that
- Make sure that the predictor can be easily configured without external dependencies
- BUILD UNIT TESTS!

Production Implementations

If we've followed the simple rules above we can do the following:

- Unit test (that's a freebie)
- Use a tool like Rcpp to build an R interface for the production predictor
- Build a comparison between the production predictor and our research predictor into the research process

And we never have to ask ourselves...is there a mathematical difference between my research and the production implementation.

Production Implementation

```
class Predictor
{
public:
    Predictor(double *betas, int count, double alpha);
    void update_prices(double *prices); // Updates prices and EMAs
    double predict(double current_price); // Predicts the forward price

private:
    double *betas_;
    double *emas_;
    double *prices_;
    int count_;
    double alpha_;
};
```

As a Package Using Rcpp

```
#include <Rcpp.h>
#include <vector>
#include "predictor.hpp"

// [[Rcpp::export]]
Rcpp::NumericVector prod_predict(Rcpp::NumericMatrix prices)
{
  std::vector< double > predictions;
  double betas[11] = {-0.3011671973112251544, -0.001550537424065 6672,
                      0.1583297052708066144, 0.1186776199122092090, 0.2926841040343490241,
                      0.1496979859345836938, -0.5244559054532954567, 0.2096527161095741720,
                      0.0767429708349566392, 0.1425406631949422132, -0.1592688290777747364};

  Predictor predictor(betas, 11, .05);
```

As a Package Using Rcpp

```
  for (int i = 0; i < prices.nrow(); ++i)
  {
    double cprices[11];

    for (int j = 0; j < 11; ++j)
    {
      cprices[j] = prices(i, j);
    }

    predictor.update_prices(cprices);
    predictions.push_back(predictor.predict(cprices[0]));
  }

  return Rcpp::wrap(predictions);
}
```

Proving It All Works

```r
library(stat385data)
library(stat385tester)

my_model_get_data <- function(symbols, dates, alpha, lag, data_dir) { … } # returns an xts

my_model_fit <- function(symbol, driver_symbols, data) { … } # Returns an lm object

symbols <- c('SPY', 'XLB', 'XLE', 'XLF', 'XLI', 'XLK', 'XLP', 'XLU', 'XLV', 'XLY', 'XME')
dates <- c('2017-01-03', '2017-01-04', '2017-01-05', '2017-01-06', '2017-01-09', '2017-01-10', '2017-01-11', '2017-01-12',
'2017-01-13', '2017-01-17', '2017-01-18', '2017-01-19', '2017-01-20')
alpha <- .05
lag <- 10

data <- my_model_get_data(symbols, dates, alpha, lag, '../data/')
result <- my_model_fit('SPY', symbols, data)

r_prediction <- unname(predict(result, data["2017-01-03", ]) + as.numeric(data["2017-01-03", "close.SPY"]))
c_prediction <- stat385tester::prod_predict(coredata(data["2017-01-03", 1:11]))
all.equal(r_prediction, c_prediction)
```

Conclusion

- Scratch work (first pass) is always necessary, but if we think about the big picture, we can skip the second pass and get to truly reusable code significantly faster.
- If we think about separating concerns properly, we will be able to create a code base that allows us to easily acquire data (often the hardest part), add features, and create models effectively and with minimal duplicated code.
- Research is only half the battle. Think big picture all the way through to production.
- Making money is left as an exercise for the reader.

It’s All in the Details

- All of the code as well as some sample data is available at https://github.com/dcdillon/dependable-development
    - Example code for a production implementation in C++
    - Rcpp based package linking the C++ predictor back into R

Some Interesting (or not) Links

Personal

- https://www.linkedin.com/in/danielcdillon
- https://github.com/dcdillon

Business - Sun Trading

- https://suntradingllc.com