Rcpp

Henry Bourne

2023-02-08

Rcpp

Rcpp intro

We will start by giving a brief introduction on how to use Rcpp. Here is the c++ version, that can be called and used by Rcpp, of our moving_avg file we made last time in C:

```
cat moving_avg.cpp
```

```
## #include <Rcpp.h>
## using namespace Rcpp;
##
## // [[Rcpp::export(name = "moving_avg")]]
## NumericVector moving_avg(const NumericVector y, const int lag)
## {
##   int ni = y.size();
##   NumericVector ys(ni);
##
##   // Define sum to store curent sum of window we are computing moving average for
##   double sum = 0;
##
##   // We now compute the moving average
##   for(int i = 1; i < ni; i++){
##       // Add newest sample
##       sum += y[i] ;
##
##       if( i >= lag ){
##           // Subtract oldest sample
##           sum -= y[i - lag] ;
##           ys[i] = sum / lag ;
##       }
##       else{
##           // Just let value be 0 if we aren't passed lag yet
##           ys[i] = 0.0;
##       }
##   }
##   return ys;
## }
```

Note that this file is much simpler than our previous one (in the last portfolio). The first thing to notice is that SEXP is mising and we don't need to use *PROTECT()* or *UNPROTECT()*, for example the type of the function is now *NumericVector* instead of SEXP. This new type wraps an R object of type SEXP and protects it from R's garbage collector whist in scope. So all we need to do now is when passing R to a C++ function is wrap all the SEXP inputs (using an appropriate wrapper) and then we can use them as normal in

C++ without having to worry about memory management. Some examples of these types are: IntegerVector, NumericVector, LogicalVector, CharacterVector, int, double, bool, String, IntegerMatrix, NumericMatrix, LogicalMatrix, CharacterMatrix. Let's now load in some data so we can test out our new version of moving average:

```
library(tidyquant)
```

```
## Loading required package: lubridate
```

```
##
## Attaching package: 'lubridate'
```

```
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
```

```
## Loading required package: PerformanceAnalytics
```

```
## Loading required package: xts
```

```
## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'
```

```
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
```

```
##
## Attaching package: 'PerformanceAnalytics'
```

```
## The following object is masked from 'package:graphics':
##
##     legend
```

```
## Loading required package: quantmod
```

```
## Loading required package: TTR
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
library(tidyr)
stock <- c("BTC-USD")
prices <- tq_get(stock, from = as.Date("2018-01-02"))
prices <- prices[,c("symbol", "date", "adjusted")]
prices <- prices %>% pivot_wider(names_from = "symbol", values_from = "adjusted")
colnames(prices)[colnames(prices) == stock] = gsub(x = stock, pattern = "-", replacement = "_")
prices <- as.data.frame(prices)
rownames(prices) <- prices$date
prices <- prices[-1]
head(prices)
```

```
##            BTC_USD
## 2018-01-02 14982.1
## 2018-01-03 15201.0
## 2018-01-04 15599.2
## 2018-01-05 17429.5
## 2018-01-06 17527.0
```

```
## 2018-01-07 16477.6
```

Now we have our data let's calculate the moving average:

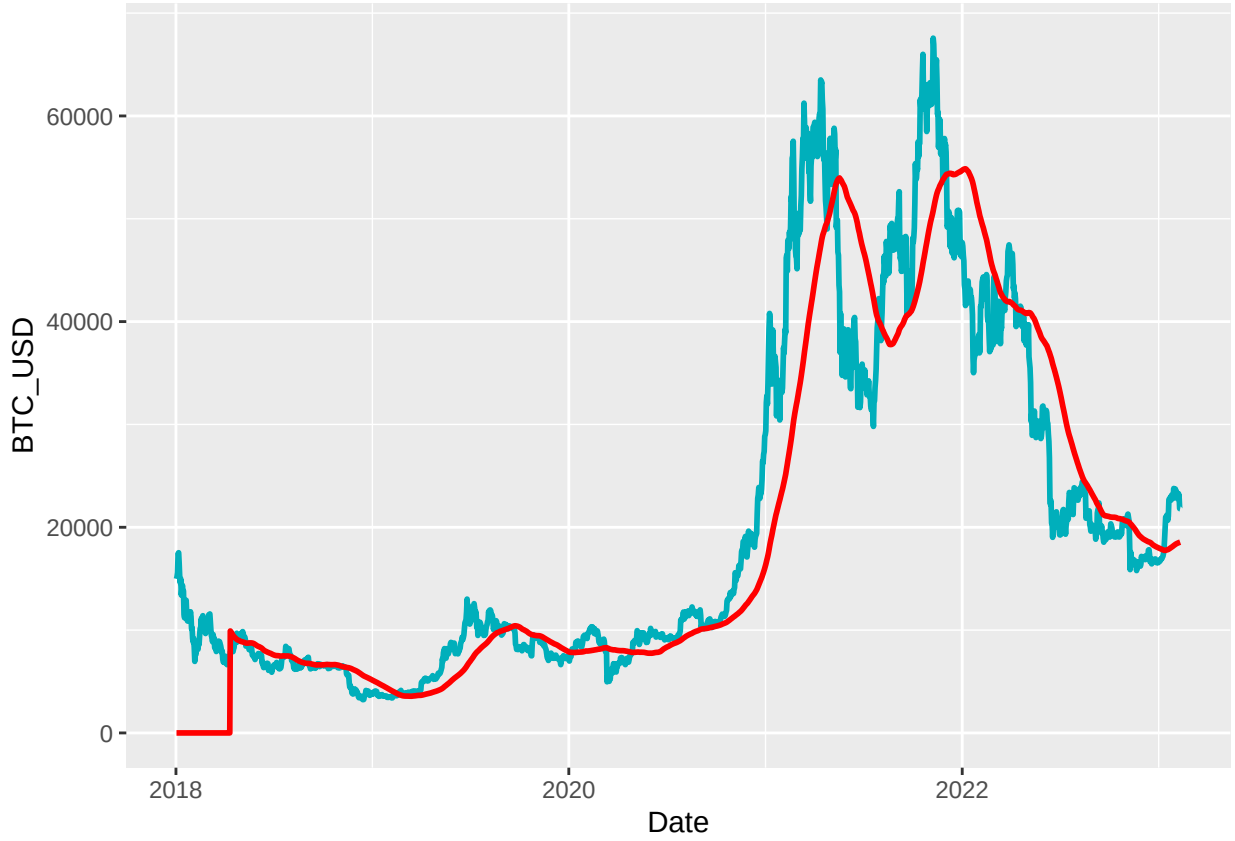
```
sourceCpp("moving_avg.cpp")
mavg_prices <- moving_avg(as.vector(prices["BTC_USD"])[[1]], 100)
prices["BTC_USD_mavg100"] <- mavg_prices
```

And let's plot our results:

```
library(SVMForecast)
library(ggplot2)
# We use a function from one of my own libraries to create a column for the data frame with the dates
prices <- long_format(prices)

# We plot the BTC_USD price data along with the moving average obtained from moving_avg()
ggplot(data = prices, aes(x = Date, y = BTC_USD))+
  geom_line(color = "#00AFBB", size = 1) +
  geom_line(aes(x=Date, y=BTC_USD_mavg100), color = "red", size = 1)
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
```



This looks exactly like what we got when we used *.Call()* with our C version! But using this method it is much simpler to write the function in C++ and it is easier to then use that function in R. One thing to note is that what *sourceCpp()* does is compile the C++ file, load the corresponding dyamic library in R (using *dyn.load()*) and creates an R wrapper with the name described by the comment in the C++ file (in ours was: "// [[Rcpp::export(name = "moving_avg")]]").

Now let's move onto inline Rcpp. Generally it's preferable to use Rcpp as above (ie. having C++ code stored

in its own .cpp files), however, it's also possible to define and execute C++ functions within an R script. We could define our moving average function in an R script by writing the following:

```
sourceCpp(code = '
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export(name = "moving_avg2")]]
NumericVector moving_avg(const NumericVector y, const int lag)
{
  int ni = y.size();
  NumericVector ys(ni);

  // Define sum to store curent sum of window we are computing moving average for
  double sum = 0;

  // We now compute the moving average
  for(int i = 1; i < ni; i++){
      // Add newest sample
      sum += y[i] ;

      if( i >= lag ){
          // Subtract oldest sample
          sum -= y[i - lag] ;
          ys[i] = sum / lag ;
      }
      else{
          // Just let value be 0 if we arent passed lag yet
          ys[i] = 0.0;
      }
  }
  return ys;
}')
```

We could then use it straight away as follows:

```
mavg_prices <- moving_avg2(as.vector(prices["BTC_USD"])[[1]], 100)
prices["BTC_USD_mavg100"] <- mavg_prices
```

Alternatively we could also write:

```
cppFunction( '
NumericVector moving_avg3(const NumericVector y, const int lag)
{
  int ni = y.size();
  NumericVector ys(ni);

  // Define sum to store curent sum of window we are computing moving average for
  double sum = 0;

  // We now compute the moving average
  for(int i = 1; i < ni; i++){
      // Add newest sample
      sum += y[i] ;

      if( i >= lag ){
```

```
          // Subtract oldest sample
          sum -= y[i - lag] ;
          ys[i] = sum / lag ;
      }
      else{
          // Just let value be 0 if we arent passed lag yet
          ys[i] = 0.0;
      }
  }
  return ys;
}')
```

Which is slightly more compact. Again we can use it right away:

```
mavg_prices <- moving_avg3(as.vector(prices["BTC_USD"])[[1]], 100)
prices["BTC_USD_mavg100"] <- mavg_prices
```

We can also immediately carry out C++ code as follows:

```
evalCpp('1.0 + 1.0', verbose = TRUE)
```

```
##
## Generated code for function definition:
## --------------------------------------------------------
##
## #include <Rcpp.h>
##
## using namespace Rcpp;
##
## // [[Rcpp::export]]
## SEXP get_value(){ return wrap( 1.0 + 1.0 ) ; }
##
## Generated extern "C" functions
## --------------------------------------------------------
##
##
## #include <Rcpp.h>
## #ifdef RCPP_USE_GLOBAL_ROSTREAM
## Rcpp::Rostream<true>&  Rcpp::Rcout = Rcpp::Rcpp_cout_get();
## Rcpp::Rostream<false>& Rcpp::Rcerr = Rcpp::Rcpp_cerr_get();
## #endif
##
## // get_value
## SEXP get_value();
## RcppExport SEXP sourceCpp_7_get_value() {
## BEGIN_RCPP
##     Rcpp::RObject rcpp_result_gen;
##     Rcpp::RNGScope rcpp_rngScope_gen;
##     rcpp_result_gen = Rcpp::wrap(get_value());
##     return rcpp_result_gen;
## END_RCPP
## }
##
## Generated R functions
## -------------------------------------------------------
```

```
##
## `.sourceCpp_7_DLLInfo` <- dyn.load('/tmp/RtmpBkKK5a/sourceCpp-x86_64-pc-linux-gnu-1.0.10/sourcecpp_3
##
## get_value <- Rcpp:::sourceCppFunction(function() {}, FALSE, `.sourceCpp_7_DLLInfo`, 'sourceCpp_7_get
##
## rm(`.sourceCpp_7_DLLInfo`)
##
## Building shared library
## --------------------------------------------------------
##
## DIR: /tmp/RtmpBkKK5a/sourceCpp-x86_64-pc-linux-gnu-1.0.10/sourcecpp_36c240beca27
##
## /usr/lib/R/bin/R CMD SHLIB -o 'sourceCpp_8.so' 'file36c2187a4814.cpp'

## [1] 2
```

Let's now move onto some more complex stuff!

Rcpp sugar

Another nice thing we can do in Rcpp is use R like stuff. For example it is possible to carry out vectorized operations if your variables are from a Rcpp vector class, eg. if variables a and b are of type *NumericVector* then $a + b$ will perform vectorized addition. Here are some examples of operations that are vectorized in Rcpp: +, -, *, /, >, <, >=, <=, ==, &, | and !. There is also Rcpp sugar for many R functions such as ifelse, mean, median, sqrt, colSums, diag, dnorm and pgamma. A full list is avaliable here: "https://thecoatlessprofessor.com/programming/cpp/unofficial-rcpp-api-documentation/#sugar".

One important thing to note is that there is a cost to calling C++ code from R. So although the C++ code its self might be faster it may not always be faster to call C++ code from R, oftentimes it may be faster to write the code straight in R.

A second thing to note is about random number generation. Just above I mentioned that you can use functions such as dnorm and pgamma in C++ with Rcpp, when using these kinds of functions they make use of the R random number generator. If using *cppFunction()* it will automatically handle everything and will use the current seed to perform the random number generation in C++. If we are working manually then things get a bit more involved and refer to this website "https://gallery.rcpp.org/articles/random-number-generation/index.html".

Rcpp Armadillo

RcppArmadillo provides an interface for the *Armadillo* C++ numerical linear algebra library. One important thing to note is that Rcpp doesn't do recycling (eg. (1, 2, 3) + 1 = (2, 2, 3) and 1 + (1, 2, 3) = 2) and that Rcpp sugar doesn't really work for matrices (at least do what we might expect), most importantly its missing the "%*%" operator. This is where Armadillo comes in. Let's write some C++ using RcppArmadillo:

```
sourceCpp(code = '
 // [[Rcpp::depends(RcppArmadillo)]]
 // ^ an RCPP attribute telling us that the code depends on the RcppArmadillo package

 #include <RcppArmadillo.h>
 // ^ includes the code from RcppArmadillo
 using namespace Rcpp;

 // [[Rcpp::export(name = "MMv")]]
 // arma::vec defines a vector in arma namespace of the the Armadillo library
 // similarly arma::mat defines a matrix
 arma::vec MMv(arma::mat& A, arma::mat& B, arma::vec& y) {
```

```
  // We then perform matrix multiplication
  return A * B * y;
}', verbose = TRUE)
```

```
##
## Generated extern "C" functions
## ----------------------------------------------------------
##
##
## #include <Rcpp.h>
## #ifdef RCPP_USE_GLOBAL_ROSTREAM
## Rcpp::Rostream<true>&  Rcpp::Rcout = Rcpp::Rcpp_cout_get();
## Rcpp::Rostream<false>& Rcpp::Rcerr = Rcpp::Rcpp_cerr_get();
## #endif
##
## // MMv
## // arma::vec defines a vector in arma namespace of the the Armadillo library  // similarly arma::mat
## RcppExport SEXP sourceCpp_9_MMv(SEXP ASEXP, SEXP BSEXP, SEXP ySEXP) {
## BEGIN_RCPP
##     Rcpp::RObject rcpp_result_gen;
##     Rcpp::RNGScope rcpp_rngScope_gen;
##     Rcpp::traits::input_parameter< arma::mat& >::type A(ASEXP);
##     Rcpp::traits::input_parameter< arma::mat& >::type B(BSEXP);
##     Rcpp::traits::input_parameter< arma::vec& >::type y(ySEXP);
##     rcpp_result_gen = Rcpp::wrap(MMv(A, B, y));
##     return rcpp_result_gen;
## END_RCPP
## }
##
## Generated R functions
## ----------------------------------------------------------
##
## `.sourceCpp_9_DLLInfo` <- dyn.load('/tmp/RtmpBkKK5a/sourceCpp-x86_64-pc-linux-gnu-1.0.10/sourcecpp_3
##
## MMv <- Rcpp:::sourceCppFunction(function(A, B, y) {}, FALSE, `.sourceCpp_9_DLLInfo`, 'sourceCpp_9_MM
##
## rm(`.sourceCpp_9_DLLInfo`)
##
## Building shared library
## ----------------------------------------------------------
##
## DIR: /tmp/RtmpBkKK5a/sourceCpp-x86_64-pc-linux-gnu-1.0.10/sourcecpp_36c22c8141c9
##
## /usr/lib/R/bin/R CMD SHLIB -o 'sourceCpp_10.so' 'file36c21e90fdcc.cpp'
```

Refer to the comments in the C++ code for information on what everything means. The main thing to note is that we declare variables as types in the armadillo namespace and then this lets us perform matrix multiplication simply using the multiplication operator. In the output given by the above chunk is included the C++ code generated by *sourceCpp()*. We note that in the generated C++ it handles turning SEXP objects into Armadillo ones and then converts them back into SEXP before returning. Let's try out our new function:

```
A <- matrix(1:4, 2, 2)
B <- matrix(5:8, 2, 2)
v <- c(1,1)
```

```
MMv(A, B, v)
```

```
##      [,1]
## [1,]   54
## [2,]   80
```

The reason Armadillo is so quick is becasue it evaluates expressions in a way that minimizes computation, eg. in the above case it evaluates $A \cdot (B \cdot y)$ as opposed to $(A \cdot B) \cdot y$ avoiding a $O(d^3)$ computation. Armadillo can also combine several operations into one and reduce the need for temporary objects, via delayed evaluation.

Into practice

Let's now put all the above into practice by writing a C++ function using Rcpp that carries out a forward pass for a neural network. We will create our matrices in R and then feed them to the C++ function to compute the forward pass. Let's say we have a fully connected feedforward neural network with two hidden layers and weights between -1 and 1:

```
n.input <- 10
n.layer1 <- 100
n.layer2 <- 50
n.out <- 1

W_1 <- matrix(runif(n.input*n.layer1,-1,1), n.layer1, n.input)
W_2 <- matrix(runif(n.layer1*n.layer2,-1,1), n.layer2, n.layer1)
W_3 <- matrix(runif(n.layer2*n.out,-1,1), n.out, n.layer2)
```

Let's now right a function in Rcpp to calculate the forward pass for a network of this form and some input:

```
sourceCpp(code = '
 // [[Rcpp::depends(RcppArmadillo)]]

 #include <RcppArmadillo.h>
 using namespace Rcpp;


arma::vec binary_step(arma::vec v) {
  // Get size of input vector and create vector to store outputs
  int n = v.size();
  arma::vec out(n);

  // Perform binary step function for each element in vector
  for(int i = 0; i < n; i++){
      if (v[i] < 0){
        out[i] = 0;
      }
      else{
        out[i] = 1;
      }
  }
  // Return result
  return out;
}


// [[Rcpp::export(name = "forwardPass")]]
arma::vec forwardPass(arma::mat& W_1, arma::mat& W_2, arma::mat& W_3, arma::vec& input, std::string act
```

```
  // We retrieve the activation function
  arma::vec (*f)(arma::vec v);
  if(act_fun == "binary_step"){
    f = &binary_step;
  }
  else{
    throw std::invalid_argument( "Didnt give valid act_fun name" );
  }

  // We perform the forward pass
  arma::mat l1 = W_1 * input;
  return  f(W_3 * f(W_2 *  f(W_1 * input)));
}')
```

Note that we have written the above code such that we can add activation functions and use them in *forwardPass()* in a quick and easy way should we choose. Let's now create our input and perform a forward pass:

```
input <- c(0,1,1,1,0,0,1,0,0,1)
forwardPass(W_1, W_2, W_3, input)
```

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

It works! above is the output of our fixed neural network!