Language Interoperation
Statistics 650/750
Week 10 Thursday

Alex Reinhart and Christopher Genovese

01 Nov 2018

Announcements

- Remember the challenge deadlines:
  - For **650**, Part 2 final revisions must be done by **Tuesday, Nov 13**
  - For **750**, Part 2 final revisions must be done by **Tuesday, Nov 6**

reticulate

The reticulate package for R provides a bridge between R and Python: it allows R code to call Python functions and load Python packages. You can even use Python code in an RMarkdown document in RStudio.

Calling Python code in R is a bit tricky. If I make an R data frame and want to give it to a Python function, how can the Python function manipulate the data frame? If a Python function returns a tuple, how does the R code access a tuple if tuples are not an R data type?

`reticulate` solves these problems with automatic conversions. An R data frame is given to Python code as a Pandas data frame; a named list (like `list(a=4, b=2, c=3)` is passed to Python as a dictionary; an R matrix is turned into a Numpy array. The conversions work in the other direction as well: when Python returns a tuple, it's turned into an R list; a Pandas data frame is turned back into an R data frame.

The introductory vignette gives plenty of examples. Let's try a couple.

Basic Usage

With `reticulate`, we can import ordinary Python packages and do things with them:

```r
library(reticulate)

os <- import("os")
os$listdir("/Users/alexreinhart/Desktop")
```

This is equivalent to:

```python
import os

os.listdir("/Users/alexreinhart/Desktop")
```

We can do fancier things like

```r
library(reticulate)

np <- import("numpy")

a <- np$arange(4)
a * 2
```

You can run entire Python scripts or bits of code and then access their results:

```r
library(reticulate)

py_run_file("some_script.py")

py_run_string("x = 10")

py$x  #=> 10
```

RMarkdown

You can seamlessly use Python and R inside the **same** RMarkdown document. For example, consider this file:

The vignette has more examples like this.

(You might need to update your knitr package to the latest version for this to work.)

Resources

- The reticulate package has documentation vignettes introducing it, showing how to use packages, and showing how to use Python in RMarkdown.

rpy2

Python is a great general-purpose language with libraries to do just about anything – connect to databases, use web services, make graphical interfaces, whatever. But R has CRAN, with every statistical method imaginable implemented in a package somewhere. If you're writing Python code, it can be frustrating to find a CRAN package with no Python equivalent.

rpy2 is a Python library that lets you call R directly from inside Python. The documentation is incomplete and unclear, but the basics are straightforward.

Examples

We can load R packages (like the built-in `base` and `stats` packages) easily:

```python
import rpy2.robjects as robjects
from rpy2.robjects.packages import importr
base = importr("base")
stats = importr("stats")
```

We can run code with the `robjects.r` method and get its results:

```
result = robjects.r("""
foo <- function(x, y) { x + y}
foo(3, 4)""")
```

If we want to create R data, we can do so explicitly:

```
xs = robjects.FloatVector([1, 2, 3, 4, 5])
```

but it's easier to do so with R functions:

```
xs = robjects.r("1:5")
```

And you can call R functions directly:

```
robjects.r['sort'](xs, decreasing=True)
```

which will return an R vector.

There's a lot more – you can use rpy2 to generate R graphics, you can access R objects and classes, work with data frames and types, use R packages, and so on. Check the documentation for more details.

Cython

Cython is an optimizing compiler for Python. It turns Python code into C code which can be compiled into highly efficient native code, provided you do a tiny bit of extra work to annotate variable types.

Cython also makes it easy to call C or C++ libraries, so if you need Python to call an external package, Cython may be the way to go. (cffi is a simpler way, if you just need to call a few C functions.)

Cython is useful when you've done extensive profiling to find bottlenecks in your code. Intensive loops and calculations can be factored out into Cython and easily made fast.

Examples

Here's some real Python code from my project:

```
def intensity_grid(xs, ys, xy, Ms, ts, alpha, theta, omega, sigma2, eta2,
                   T, t, min_dist2, min_t):
    tents = np.empty((xs.shape[0], ys.shape[0]))

    for ii in range(xs.shape[0]):
        for jj in range(ys.shape[0]):
            tents[ii, jj] = intensity_at(xs[ii], ys[jj], xy, Ms, ts, alpha,
                                         theta, omega, sigma2, eta2, T, t,
                                         min_dist2, min_t)

    return tents
```

The grid is often very big, and every `intensity_at` call requires a sum over every crime – so this is a very slow and very expensive function. I'd like to speed it up, and parallelize it if possible. We can move it to a separate file, `intensity.pyx`, and start by annotating the variable types:

```
import numpy as np

def intensity_grid(double[:] xs, double[:] ys, double[:,:] xy, long[:] Ms,
                   double[:] ts, double[:] alpha, double[:] theta, double omega,
                   double sigma2, double eta2, double T, double t,
                   double min_dist2, double min_t):
    cdef int ii, jj
    cdef double[:,:] tents = np.empty((xs.shape[0], ys.shape[0]))

    for ii in range(xs.shape[0]):
        for jj in range(ys.shape[0]):
            tents[ii, jj] = intensity_at(xs[ii], ys[jj], xy, Ms, ts, alpha,
                                         theta, omega, sigma2, eta2, T, t,
                                         min_dist2, min_t)

    return np.asarray(tents)
```

(I'll assume `intensity_at` has been moved to the same file and is getting the same sort of treatment.)

Now the generated C code doesn't need all sorts of expensive type-checking operations – it checks the variable types at the beginning of the function and then generates highly efficient code for the rest.

We're doing a lot of array accesses. Python checks the bounds on every access to make sure we don't access out of bounds. But we know we're not going out of bounds, so we can annotate:

```
@cython.boundscheck(False)
@cython.initializedcheck(False)
@cython.wraparound(False)
def intensity_grid(double[:] xs, double[:] ys, double[:,:] xy, long[:] Ms,
                   ...):
```

This tells Cython not to check array bounds, not to check if the array is initialized before we use it, and not to do wraparound indexing (i.e. `A[-1]` gets the last element of the array). This generates yet more efficient code.

One last bonus – Cython supports OpenMP, a framework for parallelization of code. This function is a prime candidate for parallelization, since each pass through the inner loop is separate from the other passes. It's embarrassingly parallel. Let's write the full parallel version:

```
@cython.boundscheck(False)
@cython.initializedcheck(False)
@cython.wraparound(False)
def intensity_grid(double[:] xs, double[:] ys, double[:,:] xy, long[:] Ms,
                   double[:] ts, double[:] alpha, double[:] theta, double omega,
                   double sigma2, double eta2, double T, double t,
                   double min_dist2, double min_t):
```

```
    cdef int ii, jj
    cdef double[:,:] tents = np.empty((xs.shape[0], ys.shape[0]))

    for ii in prange(xs.shape[0], nogil=True, schedule='static'):
        for jj in range(ys.shape[0]):
            tents[ii, jj] = intensity_at(xs[ii], ys[jj], xy, Ms, ts, alpha,
                                         theta, omega, sigma2, eta2, T, t,
                                         min_dist2, min_t)

    return np.asarray(tents)
```

`prange` is a Cython build-in function which acts like `range`, but evaluates in parallel. The `nogil` option tells Cython that I'm not going to use Python's Global Interpreter Lock, which prevents multiple threads from accessing the Python interpreter simultaneously – here I'm promising to only call Cython code, so it can be called in parallel.

The `schedule` option chooses how work will be assigned to threads (on different CPU cores). I chose to have the work just evenly divided, since each iteration should take about the same amount of time; other schemes split dynamically based on how long each iteration is taking and which threads are free.

Now my code executes in parallel. A task which would have taken over an hour, largely inside `intensity_at`, now takes fifteen or twenty minutes (on a quad-core machine).

Notice I didn't have to tell Cython that `tents` should be shared between threads; it deduced this automatically. It can also handle "reduction variables":

```
def sum(double[:] big_array):
    cdef double s = 0.0

    for ii in prange(big_array.shape[0], nogil=True, schedule='static'):
        s += big_array[ii]

    return s
```

Here each thread adds up its portion of `big_array` and the results are automatically summed together at the end, producing a nice parallel sum. This only works for simple operations, like `+`, which Cython can automatically figure out how to reduce.

Parallelization won't work for variables that have to be shared for reading and writing by multiple threads simultaneously – that's a much more difficult task, and one we'll talk about more later.

Analyzing Cython performance

The only way Cython code can be fast is if it understands your data types well enough to generate efficient C. Cython provides tools for inspecting the generated C code with `cython -a example.pyx`.

Building Cython code

Cython code has to be compiled before you can use it. You can manually convert it to C and compile it with your favorite C compiler, like GCC:

```
cython example.pyx

```

```
gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing \
    -I/usr/include/python2.7 -o example.so example.c
```

This produces an `example.so` file, a shared library you can import into Python code with the normal `import example` statement.

Alternately, you can use distutils, Python's tool for building packages. You first create a file named `setup.py` that specifies what has to be compiled:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
    name = "An example module",
    ext_modules = cythonize('example.pyx')
)
```

Then you can just run `python setup.py build_ext --inplace` and your module will be Cythonized and compiled automatically for you. There are various options you can add if you need to call other libraries, like OpenMP; see the manual for details.

You can also build this step into a Makefile to automatically compile your Cython whenever necessary.

rcpp

Rcpp Basics

Wrapping

`evalCpp()` evaluate short C++ code snippets, given as string

`cppFunction()` defines an R function from a C++ function given as a string

`sourceCpp()` compiles and links a C++ source file and exports tagged functions into R

Example:

```
f <- cppFunction('double weightedMean(NumericVector x, NumericVector w) {
  int n = x.size();
  double numerator = 0.0;
  double denominator = 0.0;
  for ( int i = 0 ; i < n ; ++i ) {
      numerator += x[i] * w[i];
      denominator += w[i];
  }
  // No error checking or assertions in this example, see below
  return numerator/denominator;
}')
f(1:4, rep(1,4)  # => 2.5
```

Note the structure of the `for` loop:

```
for ( initializers; condition; updater ) { BODY }
```

where the initializer can contain declaration of variables that are then **only visible** inside the loop.
For error checking we might include:

```
try {
    if ( is_true(any(w < 0.0)) || denominator <= 0.0 ) {
        throw std::domain_error("Invalid weights");
    }
    return numerator/denominator;
} catch(std::exception &ex) {
    forward_exception_to_r(ex);
} catch(...) {
    ::Rf_error("c++ exception (unknown reason)");
}
```

`sourceCpp()` reads its input from a file and creates a shared, dynamically linked library. (It can really also take a string with the `code` argument, which is how the other two work.)

```
// File slow-fib.cpp
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
int fibonacci(const int x) {
    if (x == 0) return(0);
    if (x == 1) return(1);
    return fibonacci(x - 1) + fibonacci(x - 2);
}
```

Note the export comment tag (the space matters), which marks the function for export to R.

```
sourceCpp("slow-fib.cpp")
fibonacci(10) # => 55
```

C++ Features

Unlike R, C++ is a compiled, statically typed language.

Each variable must be given a specific type, and each function must be declared with the types of its arguments and of its return value.

Static typing lets the compiler optimize effectively, but it puts more constraints on the developer.

C++ is a large, complex language with many features. A few things are worth remembering:

- Standard C is also legal C++.

- Syntax has similarities with R, but (non-compound) statements must all be terminated with a `;`.
- C++ (like C) is zero-indexed, not one-indexed like R. Beware.
- There is no `<-` operator: use = for assignment.
- Scalars and vectors (or other aggregate types) are not interchangeable (though a spoonful of Sugar helps).
- Functions must explicitly `return` their value.
- You can use C libraries and functions directly (note: externs).
- The Standard Template Library (or STL) exposes a wide variety of rich and well-tested data structures and algorithms.
- The Boost library is a powerful third-party library that goes above and beyond the STL.
- C++ has evolved, modern versions: C++11 and C++14 offer many nice new features. You may have to configure specially to use those features with Rcpp.

Scalar Types

The common "scalar" types are bool, int, double, and String. (All but the last of these are C++ primitive types.)

```cpp
double trim(double x, double threshold) {
    if ( x > threshold ) {
        return threshold;
    } else if ( x < -threshold ) {
        return -threshold;
    } else {
        return x;
    }
}
```

Exercise: Write a function `signum()` that takes an integer and returns -1, 0, or 1 if that integer is negative, zero, or positive.

```cpp
int signum(int x) {
  if (x > 0) {
    return 1;
  }
  if (x == 0) {
    return 0;
  }
  return -1;
}
```

Vector Types

Rcpp defines several classes to handle R vectors. These have a nice range of methods and work well with "sugar" as we'll see below.

NumericVector, IntegerVector, CharacterVector, LogicalVector

For instance, you use the `.size()` method to get the length of the vector, as illustrated above.

Several ways to create vectors:

```cpp
SEXP x;
std::vector<double> y(10);

NumericVector xx(x);       // create from a SEXP
NumericVector xx(10);      // of a given size (filled with 0)
NumericVector xx(10, 2.0); // ... with a default for all values
NumericVector xx( y.begin(), y.end() ); // range constructor

// using create
NumericVector xx = NumericVector::create(
    1.0, 2.0, 3.0, 4.0 );
// with names attribute
NumericVector yy = NumericVector::create(
    Named["foo"] = 1.0,
    _["bar"] = 2.0 ); // short for Named
```

Extracting and assigning values:

```cpp
double u = xx[0];
double v = xx(1);
double z = yy["foo"] + yy["bar"];

xx[0] = 1.618;
xx(1) = -1.0;
yy["foo"] = 10.0;

yy["foobar"] = 1;  // grow the vector
```

These vectors support some nice R-like operations:

```cpp
// [[Rcpp::export]]
NumericVector positives(NumericVector x) {
    return x[x > 0];
}

// [[Rcpp::export]]
NumericVector in_range(NumericVector x, double low, double high) {
    return x[x > low & x < high];
}

```

```
// [[Rcpp::export]]
NumericVector no_na(NumericVector x) {
    return x[ !is_na(x) ];
}

// [[Rcpp::export]]
List first_three(List x) {
    IntegerVector idx = IntegerVector::create(0, 1, 2);
    return x[idx];
}

// [[Rcpp::export]]
List with_names(List x, CharacterVector y) {
    return x[y];
}
```

Returning new vectors

```
pdistR <- function(x, ys) {
  sqrt((x - ys)^ 2)
}
```

```
NumericVector pdistCpp(double x, NumericVector ys) {
  int n = ys.size();
  NumericVector out(n);  // <- note constructor

  for(int i = 0; i < n; ++i) {
    out[i] = sqrt(pow(ys[i] - x, 2.0));
  }
  return out;
}
```

Matrix Types

Rcpp supplies various matrix types as well: NumericMatrix, IntegerMatrix, CharacterMatrix, LogicalMatrix

- Use `.nrow()` and `.ncol()` methods to get dimensions
- Use `()` not `[]` for indexing

```
NumericVector rowSumsCpp(NumericMatrix x) {
  int nrow = x.nrow();
  int ncol = x.ncol();
  NumericVector out(nrow);

  for (int i = 0; i < nrow; i++) {
    double total = 0;
    for (int j = 0; j < ncol; j++) {
      total += x(i, j);
    }
    out[i] = total;
  }
  return out;
}
```

Functions

You can pass, use, and return R functions from within C++. Note the `_[]` construction for named arguments.

```
Function rnorm("rnorm");

rnorm(100, _["mean"]=10.2, _["sd"]=3.2);
```

Other Useful Classes

List, DataFrame, Environment are often directly useful, analogously to how we use them in R.

(Note: DataFrames are not easy to use as input because of static typing.)

There are other specialized classes in the library that are less commonly used but are valuable when you need them: SEXP, DottedPair, ...

STL Interface

One of the big advantages of C++ is a fantastic and well-tuned run-time library. The STL is the center of this. Rcpp plays nicely with the STL.

An important type in the STL is the *iterator* over some collection.

```
double iteratorSum(NumericVector x) {
    double total = 0;
    NumericVector::iterator it;
    for(it = x.begin(); it != x.end(); ++it) {
      total += *it;
    }
    return total;
}
```

Note operations

`.begin()` iterator pointing to beginning of collection

`.end()` iterator pointing just past the end

`=` or `!` equality checks (cf. distance)

`++` advance (also `--` for bidirectional iterators)

`*` dereferencing.

Algorithms:

```cpp
// sum a vector from beginning to end
double s = std::accumulate(x.begin(), x.end(), 0.0);

// prod of elements from beginning to end
int p = std::accumulate(vec.begin(),
                        vec.end(), 1, std::multiplies<int>());

// inner_product to compute sum of squares
double s2 = std::inner_product(res.begin(), res.end(),
                               res.begin(), 0.0);
```

Another example:

```cpp
IntegerVector findInterval(NumericVector x, NumericVector breaks) {
    IntegerVector out(x.size());

    NumericVector::iterator it, pos;
    IntegerVector::iterator out_it;

    for(it = x.begin(), out_it = out.begin(); it != x.end();
        ++it, ++out_it) {
      pos = std::upper_bound(breaks.begin(), breaks.end(), *it);
      *out_it = std::distance(breaks.begin(), pos);
    }

    return out;
}
```

Rcpp Syntactic Sugar

Rcpp provides R-like "syntactic sugar" for operating on vectors in a concise way.

Commonly types of functions:

- Math functions: abs(), ceil(), sin(), cos(), ...
- Scalar summaries: min(), max(), sum(), ...
- Vector summaries: cumsum(), diff(), pmin(), pmax()
- Search: match(), which$_{\text{max}}$(), duplicates(), unique(), ...
- Distribution functions (d, q, p, and r versions)
- Vector views: head(), tail(), rev(), seq$_{\text{along}}$(), seq$_{\text{len}}$(), rep$_{\text{each}}$(), rep$_{\text{len}}$()

And More

There are many additional deep features in Rcpp that are useful in practice. Check the resources.

There are also many plugins and packages that are easy to include and use:

- Fast matrix computations (Armadillo)
- Eigenvalue Problems (Eigen)
- Optimization
- Monte Carlo Simulation
- Numpy interface
- Boost interfaces

See `http://rcpp.org/` for links to these packages.