RCaller 3.0: An Easy Tool for Abstraction of Java and R Connectivity

Mehmet Hakan Satman and Paul Curcean

July 9, 2016

Contents

Abstract

In this paper, version 3.0 of the open source library RCaller is introduced. RCaller is a software library which simplifies performing data analysis and statistical calculations in *Java* using R. The details are hidden from the user including transferring data between platforms, function calls, and retrieving results. Addition to the previous revisions, RCaller 3.0 implements the scripting API of Java in which the R function calls and data transfers are performed in a standard way as in the way of calling other scripting languages in Java. Besides implementation of new features, RCaller has many performance improvements in the new release of 3.0. A simulation study shows that the new release is two times faster than the previously reported and published one, especially, in process of transferring large matrices. The results of the simulation study also show that the library can be used on performing calculations with moderate size of data in reasonable times.

1 Introduction

R is an open source programming language and a programming environment for statistical and data analysis [9] which is an implementation of the S language [1]. Having an *REPL* (Read-Eval-Print-Loop) interface as a consequence of being an interpreted language; including both procedural and object-oriented as well as functional programming paradigms; having a syntax similar to ALGOL-family languages, and being developed under an open source license make R very popular in many fields of research, especially for those based on data. R has also a successful foreing language interface, namely .Call interface, which simplifies calling legacy Fortran, C, and C++ libraries. The .Call interface is also frequently used by the package developers for the code in which more performance is required for critical calculations. Since the .Call interface is very transparent and does not hide the R internals from the package developers, it is more error prone and hard to use in some cases. Rcpp is an R package which wraps the R internals using C++ templates and reduces the time required for interfacing C++ and R interoperability [3, 2] in the developing stage. Rcpp mainly wraps SEXP data structures with XXXVector, XXXMatrix, List, etc. in a way like writing code in R.

A huge amount of R code is written in C, C++, Fortran, and R itself. In addition to this, Java is another mainstream programming language which is used in many areas of software as a general purpose language. Beside being a language, Java is the name of the JVM (*Java Virtual Machine*) and an ecosystem in which many languages are compiled and translated to. As the use of Java platform increases in statistics based applications, the need of interfacing the Java language with R increases. Firstly, the Java ecosystem does not have a statistical library as comprehensive as R. Secondly, Java has some mature and legacy code such as WEKA [6, 7, 15] among others which gains productivity and efficiency when it is used with R. For interfacing Java with R and vice versa, many software packages and libraries were developed. rJava [13, 14] is a software library that uses JNI [5] (*Java Native Interface*) for both calling R from Java and vice versa. It is known that JNI is the nautral way of interfacing C and Java, and many software including the Java core libraries are implemented through JNI. This implementation makes *function callbacks* possible between the languages. Since JNI translates data types between inter-language calls, it is not too much effective to perform frequent function calls, for instance using a loop, even the operations are performed on the computers memory. In addition to this, the external native library must be compiled for the client machine and must be hosted in the java_library_path. However, rJava is one of the most efficient and scalable library for interfacing R with Java.

Rserve [12] is an other option for interfacing R and Java. Rserve opens a server socket on a specific TCP port and listens for incoming connections, possibly sent from the Java side. Whenever a connection request is received, the data protocol is activated and all of the results that are obtained in the R side are sent to Java through the TCP connection. Since Rserve can create multiple R instances in the background, more than one clients can be handled by the library. Rserve provides an unified and platform independent method, that is, the server and the client can be hosted on different kind of operating systems as well as different networks.

RCaller [10] is a LGLP'ed Java library for calling R from within Java. RCaller creates R processes, translates Java data structures to R code, sends these codes and R function calls to R, receives the results, and translates the results back to Java objects. Despite being a slower option, RCaller provides a painless method as it can be integrated by adding a single Maven entry into the project's pom.xml file. RCaller is pure Java and can be run on any machine that has Java and R are installed.

In this paper, we introduce the version 3.0 of the software library RCaller which is first introduced in [10] for the 2.X version branch. In Section 2 we give a brief guidance for adding and using RCaller as a dependency in Java projects. In Section 3, we give a brief introduction for using RCaller in simple calculations. In Section 4, we mention transferring data including vectors, matrices, data frames and Java objects between platforms. In Section 5, we demonstrate gaining performance improvements in the case of using a single R process. In Section 6, we introduce some new features of 3.0 version of RCaller. In Section 7, we replicate the simulation study that is performed in the previously published work and compare results by means of time efficiency. Finally in Section 8, we conclude.

2 Setup and Installation

RCaller is hosted on Github (https://github.com/jbytecode/rcaller) and the source code can be downloaded and compiled manually. Using git, the source tree can be downloaded using the command prompt as shown below:

Listing 1: Downloading source code

```
$ git clone https://github.com/jbytecode/rcaller.git
```

The most straightforward way of including RCaller in a project as a dependency is to add RCaller as a Maven repository into the pom.xlm file. Listing 2 shows the XML code which can be directly copied between XML tags <dependencies> and </dependencies>. By using this way, all of the dependencies including the libraries that RCaller imports are downloaded and packaged with the host application.

Listing 2: Maven dependency code

```
<dependency>
  <groupId>com.github.jbytecode</groupId>
  <artifactId>RCaller</artifactId>
  <version>3.0</version>
  <classifier>jar-with-dependencies</classifier>
</dependency>
```

In projects that are not defined as Maven structures, precompiled binaries can be downloaded from the project page[11] and added to the classpath manually.

3 RCaller Basics

RCaller basically wraps all of the details and performs type conversions between the platforms. RCaller serves three different ways of calling R from Java. In the first case, the data is transferred, a vector of results is calculated, and the result is handled in Java. In the background, an external process for Rscript executable is created which is located in the bin directory of R installation in Windows systems and /usr/bin/Rscript in Linux systems. This way of calling R is useful in the situtations that fit the one-time send-calculate-return pattern.

In the second way, an external R process is created and the process is kept alive on the memory. Commands, function calls, and data are sent to R, the results are sent back to Java, and the created process is kept alive for the later computations. This way of calling R is convenient for the situtations that fit the loop of send-calculate-return pattern and interactive computation is required.

In the third way, the process of calling R from Java is more abstract and wrapped by the scripting API which is defined by *JSR 223: Scripting for the Java^{TM}Platform* [4]. Scripting API serves a

[11] https://github.com/jbytecode/rcaller/releases

standard way for integrating a scripting language such as `javascript` with *Java* in which the function calls and the data transfers are masked. This feature is presented in version 3.0 and introduced in great detail in Section 6.1.

Most of the main classes of `RCaller` are located in the package `com.github.rcaller.rstuff`. The class `RCaller` includes methods for creating external processes and transferring data while `RCode` includes methods for converting data types and creating `R` code. Now suppose that the mean of a double vector containing values 1.0, 2.0, and 3.0 is calculated throughout `RCaller`. A simple call is given in Listing 3.

Listing 3: Simple call

```
1  RCaller caller = RCaller.create();
2  RCode code = RCode.create();
3
4  double[] arr = new double[]{1.0, 2.0, 3.0};
5  code.addDoubleArray("myarr", arr);
6
7  code.addRCode("avg <- mean(myarr)");
8  caller.setRCode(code);
9
10 caller.runAndReturnResult("avg");
11
12 double[] result = caller.getParser().getAsDoubleArray("avg");
13 System.out.println(result[0]);
```

In Listing 3, the mean of an vector is calculated on `R` side. In line 1 and line 2, instances of `RCaller` and `RCode` are created using the corresponding factory methods. In line 4, a double array is created and stored within the variable name `arr`. In line 5, the `Java` variable `arr` is converted to an `R` vector and its name is set to `myarr`. In line 7, a literal `R` code is added for calculating the mean. The code from line 10 performs the most time consuming jobs, which are: an `Rscript` process is created, the generated `R` code is sent, the calculations are performed, and the results are sent back in `XML` format. In line 12, the generated context is handled as a double array. Since the result contains a single value, length of the array is 1. Finally, the result is printed as 2.0.

4 Transferring Data

`RCaller` simplifies transferring data between `Java` and `R` plotforms. Data sent from the `Java` side is encoded into the `R` code and the calculations are performed on the `R` side. Whenever a result is requested in `Java`, the result is converted into a valid `XML` code and then parsed on the `Java` side. `RCaller` performs data transformations between platforms using the `RCode` class. The class `RCode` provides the following methods for transferring scaler types from `Java` to `R`:

- addLogical (String, boolean)
- addDouble (String, double)
- addFloat (String, float)
- addInt (String, int)
- addLong (String, long)
- addShort (String, short)
- addString (String, String)

where the arguments for variable names in type of `String` and values in corresponding type. Listing 4 shows an example of passing different types of data to `R` and getting back the results in `Java`.

Listing 4: Passing Data

```
RCode code = RCode.create();
RCaller caller = RCaller.create();

code.addBoolean("a", true);
code.addDouble("e", Math.exp(1.0));
code.addInt("i", 1);

code.addRCode("d <- a + e + i");
caller.setRCode(code);

caller.runAndReturnResult("d");

double[] result = caller.getParser().getAsDoubleArray("d");
System.out.println(result[0]);
```

4.1 Vectors and Matrices

Besides the scalar types, the class `RCode` also provides some utility functions for converting `Java` array types to `R` code. As the methods listed in Section 4, array-passing function names follow the pattern `addXXXArray(name, value)` as follows:

- addLogicalArray (String, boolean[])
- addDoubleArray (String, double[])
- addFloatArray (String, float[])
- addIntArray (String, int[])
- addLongArray (String, long[])
- addShortArray (String, short[])
- addStringArray (String, String[])

In addition to array-passing functions, the function `addDoubleMatrix` can be used for transferring matrices from `Java` to `R`. In `RCaller 3.0`, process of transferring matrices is performed in a more efficient way compared with the previous version as shown in Section 7.1 [10]. Listing 5 shows an example for performing *Singular Value Decomposition (SVD)* on a matrix defined in `Java`. The matrix is transferred from `Java` to `R`, the *svd* function is called with the given matrix as argument and the result is handled back on the `Java` side. Since two matrices `u` and `v` are produced after a `svd` call, the `getAsDoubleMatrix()` method is called twice and the result is handled in `Java` in type of `double[][]`.

Listing 5: Transferring Matrices

```
RCaller caller = RCaller.create();
RCode code = RCode.create();

double[][] d = new double[][]{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
code.addDoubleMatrix("d", d);
code.addRCode("result <- svd(d)");

caller.setRCode(code);
caller.runAndReturnResult("result");

double[][] u = caller.getParser().getAsDoubleMatrix("u");
double[][] v = caller.getParser().getAsDoubleMatrix("v");
```

4.2 Data Frames

`Data Frames` are mostly used in `R` for data representation and 3.0 version of `RCaller` has a minimum support for the `data.frame` object. A `DataFrame` object in `RCaller` contains a matrix of objects in type of `Object[][]` and an array of strings for names. The length of the string array must be equal to the number of columns of the objects matrix. There are two methods implemented for creating a data frame in `RCaller`. In Listing 6 the factory method `create` takes two arguments for the dimension of data and returns an empty `DataFrame` object with default names.

Listing 6: Default DataFrame Creator

```java
public static DataFrame create(int n, int m) {
    return new DataFrame(
    DataFrameUtil.createEmptyObjectsMatrix(n,m),
    DataFrameUtil.createDefaultNamesArray(n)
    );
}
```

Since only the dimension is given, the data matrix is filled with `null` values and the names are set to the default strings `var0`, `var1`, `...`, and `varn-1` where `n` is the number of columns. In Listing 7, a static method for creating a `DataFrame` object is shown. If the number of columns and the length of the string array are not equal, an `IllegalArgumentException` will be thrown, otherwise a `DataFrame` object containing the passed parameters is created and returned.

Listing 7: Custom DataFrame Creator

```java
public static DataFrame create(Object[][] objects, String[] names) {
    if (objects.length != names.length) {
        throw new IllegalArgumentException("...");
    }

    return new DataFrame(objects, names);
}
```

Using the factory method a `DataFrame` object can be created and sent from `Java` to `R` using the `addDataFrame` method declared in the class `RCode`.

In order to obtain the best performance and avoid any data loss, `RCaller` exports the data contained in the `DataFrame` object as a `csv` file. The path of the file will be transfered to `R` and it will be imported using the function `read.csv` on the `R` side. Since a data conversation process is not performed, `DataFrames` objects are transferred in an efficient way. Note that the current version of `RCaller` supports only the one-way transferring of `DataFrame` objects and retrieving a `data.frame` from `R` to `Java` will be implemented in further revisions.

4.3 Plain Java Objects

A `Java` object can be passed to `R` and calculations can be performed on the fields that are declared using `public` keyword on the `R` side. Any `Java` object can be passed as an `R` `List` object with elements corresponding to the public fields of the `Java` object. In Listing 8, a `Java` class is declared with three public fields.

Listing 8: A Plain Java Object Class

```java
public class PlainJavaObject {
    public double[] d = new double[]{1.0, 2.0, 3.0};
    public int[] i = new int[]{1, 2, 3};
    public String s = "Test String";
}
```

The class RCode defines the method addJavaObject which takes only one argument in type of com.github.rcaller.JavaObject. The constructor of the JavaObject class takes two arguments that define the name of object and the object itself. The object then will be converted to an R List with the given name. In Listing 9, an instance of the PlainJavaObject is passed from Java to R. The declared fields d, i, and s are accessable in R using the $ operator like myobject$d, myobject$i, and myobject$s, respectively.

Listing 9: Passing Java Objects

```
RCaller caller = RCaller.create();
RCode code = RCode.create();

code.addJavaObject(new JavaObject("myobject", new PlainJavaObject()));
code.addRCode("myobject$d <- c(9,8,7)");

caller.setRCode(code);
caller.runAndReturnResult("myobject");

System.out.println(caller.getParser().getNames());
```

The output is [d, i, s]. Since the element d of List myobject is altered, it will be returned as [9.0, 8.0, 7.0], rather than [1.0, 2.0, 3.0].

5 Single R Process for Multiple Calculations

The method runAndReturnResult that is mentioned in previous sections creates an external process for the Rscript executable, transfers data to R, performs calculations, and gets back the results in Java. Each time the method runAndReturnResult is called, a new operating system level process is created. However, this way of calling R is not much efficient. The method runAndResultOnline creates an external process for R executable and keeps it alive during the calculations. After handling the results on the Java side, the process is kept alive and waits for the next calculations. During the calculations in a session, variables and data objects are shared between the sequent calls. Since the executable file is activated once, the elapsed time during the calculations includes the time consumed by transferring data and the time consumed by R. As a result of this, only the first call consumes much time, due to the creation of the external process. The rest of the calls are performed efficiently. The code shown in Listing 10 is an example of online calling of R using a single process.

Listing 10: RCaller Online

```
RCaller caller = RCaller.create();
RCode code = RCode.create();
caller.setRCode(code);

code.addDoubleArray("d", new double[]{1.0, 2.0, 3.0});
code.addRCode("mymean <- mean(d)");
caller.runAndReturnResultOnline("mymean");
System.out.println(
    caller.getParser().getAsDoubleArray("mymean")[0]
);

code.clearOnline();
code.addRCode("myvar <- var(d)");
caller.runAndReturnResultOnline("myvar");
System.out.println(
    caller.getParser().getAsDoubleArray("myvar")[0]
);

code.clearOnline();
code.addRCode("mymed <- median(d)");
caller.runAndReturnResultOnline("mymed");
System.out.println(
    caller.getParser().getAsDoubleArray("mymed")[0]
);

caller.StopRCallerOnline();
```

In Listing 10, an array in type of `double[]` is sent to `R`. Using the same external process, the mean, the variance and the median of variable `d` are calculated and sent to `Java` sequentially. Since the process is kept alive during the calculations, it is terminated by calling the function `StopRCallerOnline` of object `caller` in type of `RCaller`.

6 Features in Version 3.0

6.1 Java Scripting Interface for RCaller

Java Scripting API provides a standard interface for wrapping all of the details of the function calls and the object transfers between the `Java` language and other scripting languages which are possibly implemented in `Java` or compiled to binary and used throught `JNI`.

Listing 11 demonstrates a simple example of calling `JavaScript` in `Java`. In this example, an instance of `ScriptEngineManager` class is created. The `getEngineByName` method returns a `ScriptEngine` object for a given language name. The `engine` object mainly implements three methods for sending objects, retrieving objects, and evaluating foreign language codes. The argument passed to `eval` method is native to the called language.

Listing 11: Calling JavaScript

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");

engine.eval("var a = 3;");
engine.put("b", 7.0);
engine.eval("var c = a + b;");

System.out.println(engine.get("c"));
```

The code shown in Listing 12 presents a simple example for interfacing `R` in `Java` throughout *Java Scripting Interface*. An instance of `ScriptEngine` is created by calling the `getEngineByName` method with argument `RCaller`. The code passed to `eval` method hosts native `R` code for creating the variable `a` in line 4. In line 5, the value of 7.0 in `Java` is sent to `R` and saved as variable `b`. In line 6, sum of `a` and `b` is assigned to the variable `mysum`. Finally, by using the `get` method, the calculated value of `mysum` variable is retrieved on the `Java` side. Differently, `mysum` is in type of `vector` rather than `scaler`, the object `result` in `Java` is in type of `double[]`. Since length of the vector is unit, the retrieved result is handled using `result[0]`.

Listing 12: Calling R

```
 1 ScriptEngineManager manager = new ScriptEngineManager();
 2 ScriptEngine engine = manager.getEngineByName("RCaller");
 3 
 4 engine.eval("a <- 3");
 5 engine.put("b", 7.0);
 6 engine.eval("mysum <- a + b");
 7 
 8 double[] result = (double[])engine.get("mysum");
 9 System.out.println(result[0]);
10 
11 ((RCallerScriptEngine)engine).close();
```

Listing 13 demonstrates an other simple example of sending an array from `Java` to `R`, performing a sorting operation on the `R` side, and retrieving the sorted array from `R` to `Java`. The `Java` array `a` is of type `double[]` and it is sent using the method `put()`. Since the code `b <- sort(a)` is native to `R`, it is evaluated using the `eval()` method. Finally, the sorted array `b` is retrieved using the method `get()`. These three methods are defined in the scripting API and the behaivour is in the same way of calling other scripting languages in `Java`.

Listing 13: Passing Java Arrays

```
double[] a = new double[]{19.0, 17.0, 23.0};
engine.put("a", a);
engine.eval("b <- sort(a)");
double[] result = (double[]) engine.get("b");
```

Function calls can be performed using the `eval()` method as shown in the previous examples. In addition to this way, the `Java Scripting API` provides the `Invocable` interface that hosts some functions for calling external functions in `Java`. The `invokeFunction` function defined in `Invocable` interface takes variable number of arguments in which the first one is function name and the others are function arguments that will then passed to the function. Listing 14 demonstrates calling `runif` function which is defined in `R` with parameters `n`, `min`, and `max` for expressing the number of generated random numbers, lower bound of the random numbers, and upper bound of the random numbers, respectively. Since the function arguments can be passed to functions with their names, the arguments in `invokeFunction` are specified with `Named()` function in which the argument names and their values are defined. In the example, 5 random numbers are drawn using the Uniform(0,100) distribution. The result calculated in `R` and retrieved in `Java` is of type `ArrayList<NamedArgument>`, because the output is supposed to be vector or a list in which the array of results are possibly stored with their names. Each result in the array has properties of `Obj` and `Name`. The values are handled with the `getObj()` method whereas the names are handled using the `getName()` method.

Listing 14: Invoking runif

```
Invocable invocable = (Invocable) engine;
Object result = invocable.invokeFunction(
                    "runif",
                    Named("n", 5),
                    Named("min", 0),
                    Named("max", 100)
                  );
ArrayList<NamedArgument> allresults =
                     (ArrayList<NamedArgument>) result;

double[] dresult = (double[]) allresults.get(0).getObj();
```

The example given in Listing 15 is similar with the one given in Listing 14. The `sqrt` function defined in `R` takes the argument `x` and returns the square root of `x`. The function call can be performed using `sqrt(x = 5.0)` as well as using `sqrt(5.0)`. If the orders of arguments are not important, the `Named` function can be used without argument names. In Listing 15, the `sqrt` is called with value of 25.0 without an argument name. Since the function returns a scaler rather than an array, it is stored in the first element of the return list and handled by object `dresult[0]`. Note that the `invokeFunction` method can call any `R` function including the user-defined functions. Listing 16 shows an example of calling a user-defined `R` function which is defined on the `Java` side. Function `f` takes only one argument `a` and returns the power of 2.

Listing 15: Invoking sqrt

```
Invocable invocable = (Invocable) engine;
ArrayList<NamedArgument> allresults =
      (ArrayList<NamedArgument>) invocable.invokeFunction(
                                  "sqrt",
                                  Named("", 25.0)
                                );

double[] dresult = (double[]) allresults.get(0).getObj();
```

Listing 16: Invoking user-defined functions

```
Invocable invocable = (Invocable) engine;
double[] x = new double[]{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};

engine.eval("f <- function(a){return(a^2)}");

ArrayList<NamedArgument> allresults =
            (ArrayList<NamedArgument>) invocable.invokeFunction(
                                           "f",
                                           Named("a", x)
                                         );

double[] dresult = (double[]) allresults.get(0).getObj();
```

6.2 R Start-up Options

Since the `R` executable can be run with several arguments which can be set in the command prompt, `RCaller` can create an external process of `R` by setting these variables in version 3.0. By default, `RCaller` starts an `R` process with the option `--vanilla`. By using this option, the process is not started with an existing enviroment, the variable pool is not saved and the process does not read the `RProfile.site`. Shortly, in each creation of the external process, a *clean* environment is started without a history. The `--vanilla` option of processes is the default one in the current version of `RCaller` for performance issues. However, starting a clean process is not always the best option, because the sources obtained in the previous sessions would be usable in the current one.

For a performance improvement, some implemented methods can be stored in `RProfile.site` and possibly be compiled into the `bytecode` using `cmpfun` from package `compiler` [8, 11]. But in order to use the methods that are exported in this file, `RCaller` should not be started with the default option `--vanilla` or the option `--no-site-file`.

The static method `RProcessStartUpOptions` is encapsulated in the `RCallerOptions` class and it is called with default values in instantiation of a new `RCaller` object. Creating a `RStartUpOtions` object can be performed by either with the default values of arguments or with the user-defined values. Listing 17 shows the factory method for creating an `RPocessStartUpOptions` object with default arguments and the argument for the option `--vanilla` is set to `true`.

Listing 17: Default RProcessStartUpOptions Creator

```
 public static RProcessStartUpOptions create() {
        return new RProcessStartUpOptions(
            false, false, false, false, false,
            false, false, false, false, true, false, false,
            false, false, false, false, null, null,
            null, null, null, null, null, null, null);
    }
}
```

Listing 18: Custom RProcessStartUpOptions Creator

```
public static RProcessStartUpOptions create(
      boolean save, boolean noSave, boolean noEnviron,
      boolean noSiteFile, boolean noInitFile, boolean restore,
      boolean noRestoreData, boolean noRestoreHistory,
      boolean noRestore, boolean vanilla, boolean noReadLine,
      boolean quiet, boolean silent, boolean slave,
      boolean interactive, boolean verbose,
      Integer maxPPSize, Integer minNSize, Integer minVSize,
      String debugger, String debuggerArgs, String gui,
      String arch, String args, String file) {

        return new RProcessStartUpOptions(
               save, noSave, noEnviron, noSiteFile, noInitFile,
               restore, noRestoreData, noRestoreHistory, noRestore,
               vanilla, noReadLine, quiet, silent, slave, interactive,
               verbose, maxPPSize, minNSize, minVSize, debugger,
               debuggerArgs, gui, arch, args, file);
    }
}
```

Listing 18 shows the factory method `create` for creating a configuration for starting up options. Invoking the `create` method with the arguments that are given in Listing 19 produces the option string that is passed to external `R` process as in shown in Listing 20.

Listing 19: Custom RProcessStartUpOptions Creator

```
RProcessStartUpOptions rProcessStartUpOptions =
        RProcessStartUpOptions.create(
               true, false, false, false, true, true,
               false, false, false, false, false, true,
               true, true, false, false, null, null, 10,
               null, null, "test", null, null, null
    );
```

Listing 20: Generated Options

```
--save --no-init-file --restore --quiet --silent ⏎
--slave --min-vsize=10 --gui=test
```

7 Performance Issues

7.1 Performance Improvements

A simulation study is reported in the recent work [10] to reveal the performance of `RCaller` for comparative aims. In the original simulation study, a randomly created `double` vector of size 1000 is created on the `Java` side and sent to `R` to calculate a new `double` vector which is squared of the original. Finally the squared vector is retrieved on the `Java` side. We replicate the same simulation with the version 3.0 and the results of the previous simulations and new ones in milliseconds are shown in Table 1[12].

[21]The simulations are performed on the same computer with same configuration but new versions of `R` and `Linux` are installed.

Table 1: Performance improvements in RCaller 3.0

	RCaller 2.2		RCaller 3.0	
Statistic	RCaller	RCallerOnline	RCaller	RCallerOnline
min	557	257	247	103
mean	569	266.96	255.70	111.52
median	565	263	255	111
max	643	296	518	366
std.dev.	14.92	9.63	10.31	9.68
mad	4.45	5.93	2.96	2.96

In Table 1, it is shown that the new version is more than two times faster than the older version in average. Standard deviations are also reduced, that is, the new version differs less between calculations. It is also shown in Table 1 that median absolute deviations (*mad*) are reduced, but the reduction on *mad* values are larger than the reduction on standard deviations. The main reason of this situation is that the first attempt of creating `RCallerOnline` is more time consuming and the sequent calls are faster. The simulation results without the initialization process are reported in Table 2. Since the median and mad statistics are more robust when they are compared to summation based counterparts, they stay the same. As a result of this, median and mad are more useful for comparing the performances of versions. It is shown in Table 1 and Table 2 that the volatility of elapsed computation time as well as the computation time itself are reduced in the version 3.0.

Table 2: Performance improvements without initialization

Statistic	RCaller	RCallerOnline
min	247	103
mean	255.43	111.26
median	255	111
max	380	179
std.dev.	6.11	5.37
mad	2.96	2.96

7.2 Time Consumed on Transferring Matrices

Since data are transferred as text and `XML` formats, `RCaller` consumes more time than a method that transfers the data in binary. We perform a simulation study to reveal the time performance of `RCaller` on sending a matrix from `Java` to `R`, performing a small operation on this matrix on the `R` side, and finally retrieve the matrix from `R` to `Java`. Specifically, we create a matrix with dimensions of $n \times n$ in `Java`, we send it and calculate the transpose of the matrix, and retrieve the result in `Java` where $n = 2, 3, ..., 150$. Since calculating a transpose of a matrix is not a time consuming operation, transferring square matrices are prominent by means of both sending and receiving the data. The results of the simulation is shown in Figure 1.

In Figure 1, it is shown that the time consumed by transferring data between two platforms increases as the dimension of the square matrix increases. Increase of the consumed time is upward-concave, that is, increase of the computation time also increases as the dimension gets larger and larger. In Listing 21, it is shown that data fits the model

$$t = \beta_0 + \beta_1 n + \beta_2 n^2 + \epsilon \tag{1}$$

well. Since parameters are significant, we can conclude that the time curve is upward-concave.

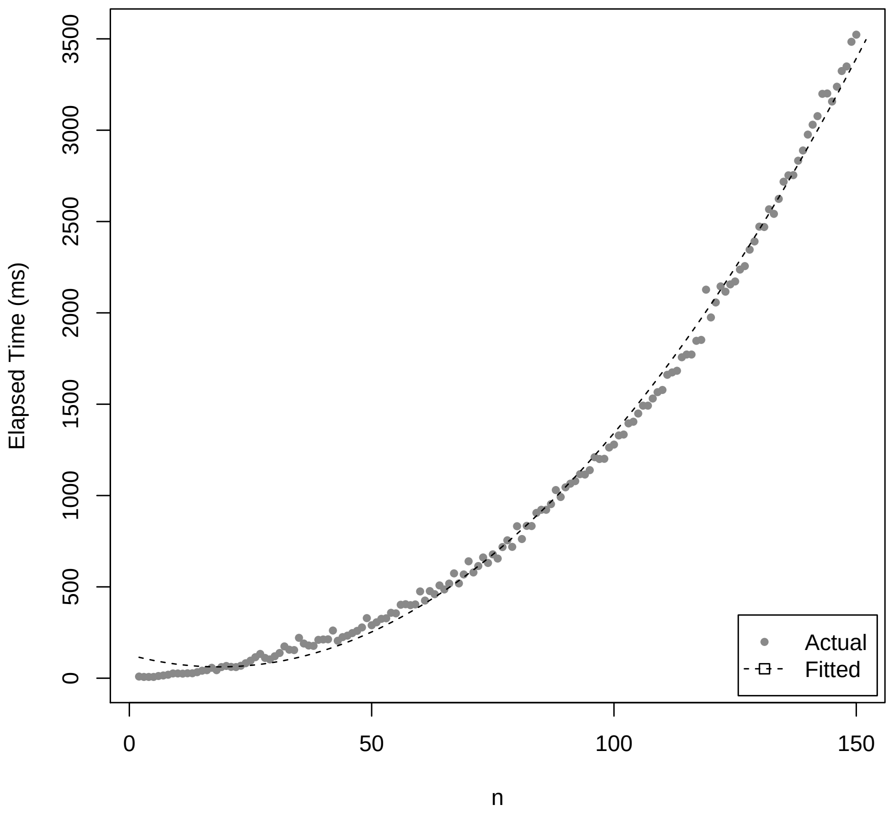

Figure 1: Time consuming by transferring matrices in milliseconds

Listing 21: Fitted Curve

```
Coefficients:
                   Estimate  Std. Error  t value  Pr(>|t|)
(Intercept)     127.984946   14.646752    8.738  5.12e-15 ***
n                -7.127311    0.444450  -16.036   < 2e-16 ***
n^2               0.192632    0.002835   67.956   < 2e-16 ***
---
Signif. codes:  0    ***    0.001    **    0.01    *    0.05

Residual standard error: 57.25 on 146 degrees of freedom
Multiple R-squared:  0.9968,    Adjusted R-squared:  0.9968
F-statistic: 2.295e+04 on 2 and 146 DF,  p-value: < 2.2e-16
```

The estimated regression equation can be written using the results given in Listing 21 as

$$\hat{t} = 127.984946 - 7.127311n + 0.192632n^2 \tag{2}$$

and the derivative $\frac{d\hat{t}}{dn}$

$$\frac{d\hat{t}}{dn} = -7.127311 + 0.385264n \tag{3}$$

can be obtained. Using these estimates, it can be concluded that increasing the matrix dimensions from 100×100 to 101×101 increases the consumed time by 31.39909 milliseconds, approximately. Similarly, increasing the dimensions from 150×150 to 151×151 increases the computation time by 50.66229 milliseconds, approximately. However, both of the operations take 290ms for a 50×50 matrix and 1279ms for a 100×100 matrix. Consequently, the method can be used for small and moderate size matrices. If the operation takes more time on the `R` side, in other terms, the time consumed by the operation is larger than the time consumed by the transferring data, the method can be useful in the data of larger sizes.

8 CONCLUSIONS

`RCaller` is a simple to use `Java` library for `Java` and `R` interoperability. Basically, `RCaller` is based on creating an external process of executable file `Rscript.exe`. The `R` code and the data created in `Java` are converted to `R` code and sent to `R` throughout streams, computations are performed in `R`, and finally the result is sent back to `Java` in `XML` file format. Besides creating an external process for each calculation, `RCaller` provides an `online` method for performing more than one operations sequentially using a single `R` process which is kept alive along the application lifetime. This use of `RCaller` is more efficient than the `RScript.exe` based one as shown in the simulation study of the previous study. In this paper, the simulation study is replicated for the version `3.0` and the results of the simulation study show that the performance is improved 100% by means of time efficiency. The new version also implements the `scripting API` in `Java` and the process of calling `R` is more like calling a scripting language such as `javascript` in `Java`. By using this `API`, calling `R` from `Java` is oversimplified using four basic methods `get`, `eval`, `put`, and `invokeFunction`.

We also perform a stress test for measuring the time consumed by matrix operations and the results of this test show that the increasing the dimension of a squared matrix from 150×150 to 151×151 increases the computation time 50 milliseconds, whereas, increasing the dimension of a squared matrix from 200×200 to 201×201 increases the computation time 70 milliseconds, approximately. Since only the transpose of a matrix is calculated on the `R` side, the time consumed by sending and receiving matrices are prominent.

Time improvements, new features implemented and being easy to use, make `RCaller` an elegant option in `jvm` based projects which need statistical calculations in the background. `RCaller` hides the details of interactions of two seperate platforms and the users are more able to focus on the development as they are performing all of the calculations on the `Java` side only.

References

[1] Richard A Becker, John M Chambers, and Allan R Wilks. The new s language. *Pacific Grove, Ca.: Wadsworth & Brooks, 1988*, 1, 1988.

[2] Dirk Eddelbuettel. *Seamless R and C++ integration with Rcpp.* Springer, 2013.

[3] Dirk Eddelbuettel, Romain François, J Allaire, John Chambers, Douglas Bates, and Kevin Ushey. Rcpp: Seamless r and c++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011.

[4] Jeff Friesen. *Beginning Java^{TM} SE 6 Platform: From Novice to Professional*, chapter Scripting, pages 281–344. Apress, Berkeley, CA, 2007.

[5] Rob Gordon. *Essential JNI: Java Native Interface.* Prentice-Hall, Inc., 1998.

[6] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[7] Kurt Hornik, Christian Buchta, and Achim Zeileis. Open-source machine learning: R meets Weka. *Computational Statistics*, 24(2):225–232, 2009.

[8] Aloysius Lim and William Tjhi. *R High Performance Programming.* Packt Publishing Ltd, 2015.

[9] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2016.

[10] M Hakan Satman. Rcaller: A software library for calling r from java. *British Journal of Mathematics & Computer Science*, 4(15):2188, 2014.

[11] Luke Tierney. Compiling r: A preliminary report. In *Proceedings of DSC*, volume 2, page 2, 2001.

[12] Simon Urbanek. A fast way to provide r functionality to applications. In *Proceedings of DSC*, volume 2. Citeseer, 2003.

[13] Simon Urbanek. How to talk to strangers: ways to leverage connectivity between r, java and objective c. *Computational statistics*, 24(2):303–311, 2009.

[14] Simon Urbanek. *rJava: Low-Level R to Java Interface*, 2016. R package version 0.9-8.

[15] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, San Francisco, 2nd edition, 2005.

Acknowledgement

Please cite `RCaller` using the *Bibtex* entry:

```
@article{satman2014rcaller,
  title={RCaller: A software library for calling R from Java},
  author={Satman, M Hakan},
  journal={British Journal of Mathematics \& Computer Science},
  volume={4},
  number={15},
  pages={2188},
  year={2014},
  publisher={SCIENCEDOMAIN International}
}
```

for LaTeX,

```
Satman, M. Hakan. "RCaller: A software library for calling R from Java."
British Journal of Mathematics & Computer Science 4.15 (2014): 2188.
```

for other text editors.