D2-1 Optimisation of R support vector machine on HECToR with SPRINT

Project Title	ESPRC dCSE "Bootstrapping and support vector machines with R and SPRINT"
Document Title	D2-1 Optimisation of R support vector machine on HECToR with SPRINT
Authorship	Luis Cebamanos
Document Filename	SPRINT-dCSE3-svm_D2-1_report-v0.2
Document Version	0.2
Document Number	
Distribution Classification	Public following PI permission

Document History

Personnel	Date	Comment	Version
T. Sloan	15/01/2013	Renamed file and added frontpage for submission to NAG	0.2
L. Cebamanos	29/06/2012	Original draft	0.1

Contents

Introduction

They key objective of this dCSE project was to implement a parallel version of the support vector machine method, a learning method that analyzes data and recognizes patterns used for classification and regression analysis. A user requirements survey conducted at the beginning of the SPRINT project, highlighted the support vector machine (SVM) function as one of the candidates for parallelisation.

The Simple Parallel R Interface (SPRINT) [1], developed by the Division of Pathway Medicine and EPCC at The University of Edinburgh, allows R users to exploit the HECToR supercomputing facility and other High Performance Computing (HPC) systems without expert knowledge of such systems. R is a freely available language and environment for statistical computing (http://www.r-project.org/).

The SVM function

The standard SVM takes a set of input data and predicts, for each given input, which of two possible classes forms the input, making the SVM a non-probabilistic binary linear classifier. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

A classification task usually involves separating data into training and testing sets. Each sample in the training set contains one target value (i.e the class labels) and multiple attributes (i.e. the observed variables). The SVM method will generate a model based on the training data provided which predicts new target values, provided through the test data, given only the test data attributes.

The R package e1071 offers an interface to the C++ implementation by Chih-Chung Chang and Chih-Jen Lin [2], libsvm featuring:

- C- and ν-classification
- One class classification
- ϵ- and ν-regression

also includes:

- linear, polynomial, radial and sigmoidal kernels
- formula interface
- *k*-fold cross validation

The SVM implementation in R detects if *svm* is used for classification or regression depending on the dependant variable *y*: if *y* is a factor, the engine switches to classification, otherwise it works as a regression machine.

An example of how to use the *svm* function from package e1071 it is explained here using the

Golub data set. To install the Golub data set, in R type:

```
>source("http://bioconductor.org/biocLite.R")
>biocLite("golubEsets")
```

Then load the libraries needed to run svm:

```
>library(class)
>library(e1071)
>library(golubEsets)
```

The original Golub study consisted of both the training and the test samples. We load the objects to run and test the *svm* function.

```
>data(Golub_Train)
>data(Golub_Test)
```

Extract the microarray data from each sample

```
>train <- exprs(Golub_Train)
>test  <- exprs(Golub_Test)
```

Get the vector that contains the known leukimia type for each pacient

```
>samplelabels <- Golub_Train@phenoData@data$ALL.AML
```

We train the SVM classifier with the chosen parameters. We have chosen a radial kernel and a value of cross validation of 10.

```
>svmmodel1 <- svm(x=t(train), y=samplelabels, type='C', kernel="radial", cross=10)
```

The trained classifier is now applied to samples of unknown leukemia type, the Golub_Test data set

```
>predmodel1 <- predict(svmmodel1, t(test), decision.values = TRUE)
```

We can now compare the predicted classes with the real one:

```
>table(predmodel1, Golub_Test@phenoData@data$ALL.AML)
predmodel1 ALL AML
     ALL  20  12
     AML   0   2
```

```
>classAgreement(table(predmodel1, Golub_Test@phenoData@data$ALL.AML))
$diag
[1] 0.6470588
$kappa
[1] 0.1639344
$rand
[1] 0.5294118
$crand
[1] 0.05752685
```

A perfect class prediction should be 0 in the table result and 100% in the class agreement.

The SPRINT parallel interface is called *psvm* and takes the same parameters as the sequential implementation. It requires the loading of the sprint library and termination of MPI after the computation is finished. Below is an example of how to call *psvm*:

```
#load libraries
>library(sprint)
>library(golubEsets)

#load data
>data(Golub_Train)

# get microarray expression
>train <- exprs(Golub_Train)

#Get vector with classes
>samplelabels <- Golub_Train@phenoData@data$ALL.AML

#call psvm
>psvmmodel <- psvm(x=t(train), y=samplelabels, type='C', kernel="radial", cross=10)

#close mpi communication and quit R
> pterminate()
> quit()
```

Profiling

Support Vector Machine (SVM) was highlighted as one the candidates for parallelisation by a user requirements survey conducted at the beginning of the SPRINT project.

The Golub data set [3] was selected for profiling purposes as it contains training and test data samples that fit perfectly for SVM classification. This data set contains over 7000 genes from 38 patient samples. In order to make the data set more complex, i.e. larger, it was replicated in number of genes and patients.

For profiling purposes we used Rprof since it was not possible to use gprof to profile the C code.

The results obtained from calling *svm* for classification using a radial kernel can be seen on Figure 1.

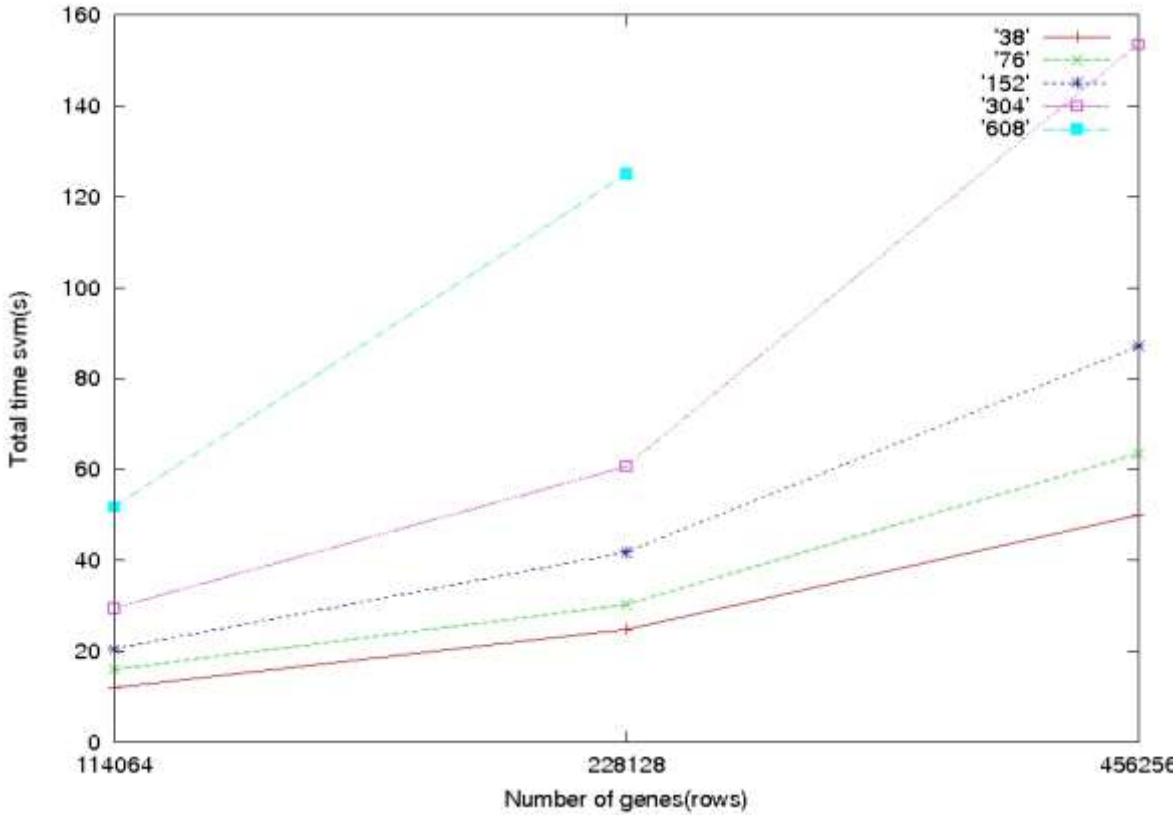

Figure 1: Time consumed in svm depending on the number of genes and the patient samples.

Figure 1 shows the time invested in the *svm* call depending on the number of genes. The line's code colour shows the number of columns or patient samples. It is possible to see how the time invested in *svm* increases with the number of genes, however there is a stronger influence from the number of patient samples. Although the *svm* time always increases with the number of patient samples, it is possible to group the lines when this number is not large. Up to 152 patient samples we see a very similar behaviour, however the *svm* time dramatically increases when using more than 200 patient samples. The blue line shows how the *svm* time with 608 patient samples could not be plotted for more than 228,128 genes due to run out of memory in a machine with 8 GB of RAM memory.

While it is not common to find data sets larger than 200,000 genes/rows and 1,000 samples/columns in a biology study, nevertheless the time consumed in the *svm* calculation shows that it will not be larger than 130 seconds.

The *svm* function contains a large amount of input parameters. An important parameter widely used in statistics is cross validation. If no value for cross validation is passed, *svm* uses 0 by default. The uses of positive values, *k*, for cross validation will run the training algorithm immerse in *svm k* times, on different subsets of the training set. By using multiple runs, each not using all the samples, it is possible to get some estimates for how accurate the trained classifier is. Figure 2 shows the *svm* time with cross validation value of 10.

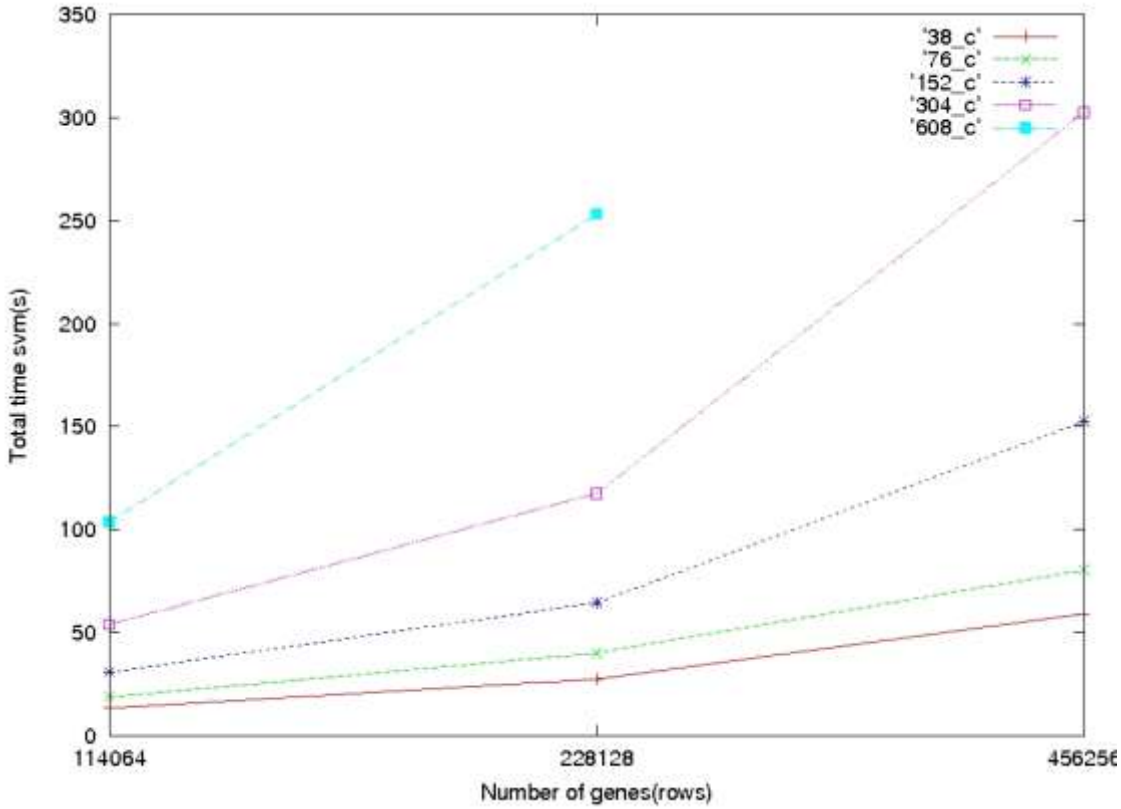

Figure 2: Time consumed in svm depending on the number of genes and patient samples with cross validation 10.

As seen in Figure 1, we see similar behaviour in Figure 2 for the gene/sample runs with the only difference of being the total time. The reason for this is the time consumed by using multiple runs.

As mentioned at the beginning of this report, *svm* was chosen by R users as a candidate for parallelisation. After consultation with users at the Division of Pathway Medicine, the approach taken was to produce an implementation of SVM in SPRINT that allows multiple instances of the R implementation of *svm* to be run in parallel and to allow cross-validation runs also to be run in parallel.

Implementation

The project has produced a full integration of the *svm* function using the R package e1071 and libsvm library. The integration of libsvm library into SPRINT also means the addition of C++ code to SPRINT. Previously SPRINT only used C code, however following this project, SPRINT is now able to compile and run C++ code adding more functionality to SPRINT since there are lots of interesting libraries using C++ code.

In order to properly run and compile C++ within SPRINT some changes were needed to make in the Makefile file:

```
sprint.so: $(INTERFACE_OBJS) $(SHARED_ALGORITHM_OBJS) $(SHLIB_OBJS) $(IMPLEMENTATION_OBJS)
ifeq (mpicc,$(findstring mpicc,$(MPICC)))
        $(MPICXX) -shared -o $@ $^
else
```

```
	$(CC) -shared -o $@ $^ $(MPILIBS)
endif

.cpp.o:
	$(MPICXX) $(LOCAL_CFLAGS) -I$(R_INCLUDE_DIR) -c -o $@ $<

.c.o:
	$(MPICC) $(LOCAL_CFLAGS) -I$(R_INCLUDE_DIR) -c -o $@ $<
```

Makevars file:

```
MPICC    = mpicc -std=gnu99
MPICXX   = mpicxx
MPILIBS  =
```

and to Makevars.in file

```
MPICC    = @MPICC@
MPICXX   = @MPICXX@
MPILIBS  = @MPILIBS@
```

In addition to all these changes, the C++ code has to be called from a C wrapper which is called from the R interpreter. SPRINT uses a C interface to call the C implementation, which is also responsible for calling the C++ classes and other structures included in C++ code.

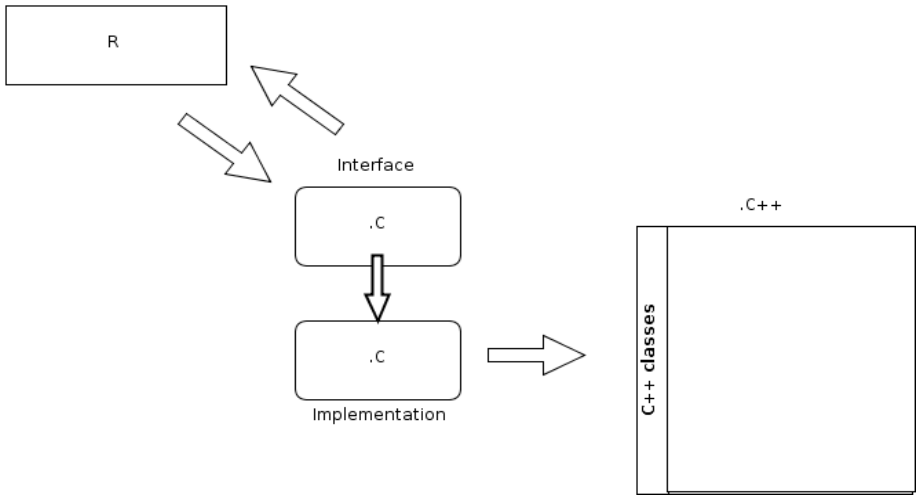

Figure 1 shows the communication process in a function that uses c++ code, *psvm* for instance, where the C code, integrated in the implementation file, is able to call a pool of C++ classes and other structures.

The R interface of *psvm* is almost identical to the *svm* implemented in package e1071 therefore there are no big differences in the fashion that they are called by the user. All the parameters not used are defined by default as was the case in the *svm* function provided by package e1071.

psvm is able to run totally independent instantiations of *svm* on different processors through MPI. Given the parameters to an *svm* function, *psvm* will run the *svm* function on a number N of processors indicated by the user providing similar results.

Furthermore, if positive values of cross validation are passed as arguments, *psvm* is able to run those multiple runs in parallel, where each run will be running independently on different processors and will gather the results back to the master.

Future work

Our effort to track the bottleneck in SVM points to the *svmtrain* function. Since the standard R profiler does not give much information about the C code, timing areas were used for that purpose. That allowed us to track that *svmtrain* was the function where SVM spent most of its time. However, the SVM function calls many other functions which depend on the arguments of the SVM function. It will be worthwhile profiling the C code to understand its behaviour depending on the arguments. In this project SVM classification was investigated and this indicated the sequential minimal optimization (SMO) algorithm included in the class *Solver* is a bottleneck. Future effort at parallelizing SVM should focus on this algorithm which is thought to be the core of the application.

Conclusions

We have presented an R implementation of SVM integrated with the SPRINT package and successfully run this on HECToR. Furthermore, this implementation of SVM is able to run some sections in parallel, such as the cross validation section. We have identified areas to tackle in a full parallelization of the R *svm* function Our current implementation has an identical interface to the *svm* included in e1071 package thus making it very easy to adapt existing R scripts to use it. Furthermore, we have been able to adapt SPRINT package to run C++ code which gives more functionality to SPRINT for future implementations.

Acknowledgements

The SPRINT project is funded by a Wellcome Trust grant [086696/Z/08/Z]. This project was funded under the HECToR distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR – A Research Councils UK High End Computing Service – is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE

References

[1] J. Hill, M. Hambley, T. Forster, M. Mewissen, T. Sloan, F. Scharinger, A. Trew, and P. Ghazal. SPRINT: A new parallel framework for R. BMC Bioinformatics, 9(1):558, 2008.

[2] Chang, C.-C. & Lin, C.-J., LIBSVM: a library for support vector machines.,

[3] T.R. Golub, D.K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J.P. Mesirov, H. Coller, M. Loh, J.R. Downing, M.A. Caligiuri, C.D. Bloomfield, and E.S. Lander, Molecular Classification of Cancer: Class Discovery and Class Prediction by Gene Expression,