Rgtsvm package

Rgtsvm version:0.55
Date: 02/10/2018

Abstract

Rgtsvm provides a GPU based solution for support vector machine (SVM) and support vector regression (SVR) implementation in the R computing environment. It offers a compatible interface with the svm function in *e1071* [1], but at a much faster computing speed. *Rgtsvm* supports three mostly commonly used machine learning tasks: binary classification, multiclass classification and epsilon-regression. Kernels provided by the *e1071* package are all available in *Rgtsvm*, which include linear, polynomial, sigmoidal, and radial basis function. k-fold cross validation and the tuning of hyper-parameters are also supported. To facilitate learning on exceptionally large dataset, *Rgtsvm* provides a pointer-based solution to circumvent the need to physically copy the data matrix.

1. Installation

The *Rgtsvm* is an add-on package for the R system which supports GPU architecture. Currently it only works on Linux or compatible Linux, and no Windows version is available. The package depends on the CUDA library, the Boost library, and *bit64* package in R. *Rgtsvm* requires that paths to the *CUDA* and *Boost* libraries are specified by the user during installation, either by an R command or "*install.packages*" in the R console.

a) Download the source codes from GitHub

```
$ git clone https://github.com/Danko-Lab/Rgtsvm.git
$ cd Rgtsvm
```

b) Install the package using 'R CMD INSTALL'

```
$ export CUDA_HOME=/usr/local/cuda-8.0
$ export BOOST_HOME=/usr/boost/1.55.0
$ R CMD INSTALL --configure-args="--with-cuda-home=$CUDA_HOME --with-
boost-home=$BOOST_HOME" Rgtsvm
```

If you have installed the package *devtools*, you can install *Rgtsvm* directly from Github without downloading source codes manually. It still requires the paths of *CUDA* and *Boost*.

```
> library(devtools)
> install_github("Danko-lab/Rgtsvm/Rgtsvm", args="--configure-args='--
with-cuda-home=YOUR_CUDA_PATH --with-boost-home=YOUR_BOOST_PATH'" )
```

In order to use CUDA and BOOST, some clusters require users to load modules into the system environment. E.g., users load the CUDA and BOOST on the Stampede server (XSEDE), and then users can check the home folder using the grep command as follows:

```
$ module load cuda
$ module load boost/1.55.0

$ printenv | grep CUDA
TACC_CUDA_BIN=/opt/apps/cuda/6.5/bin/
TACC_CUDA_LIB=/opt/apps/cuda/6.5/lib64/
TACC_CUDA_INC=/opt/apps/cuda/6.5/include
TACC_CUDA_DIR=/opt/apps/cuda/6.5/

$ printenv | grep BOOST

TACC_BOOST_INC=/opt/apps/gcc4_9/boost/1.55.0/x86_64/include
TACC_BOOST_LIB=/opt/apps/gcc4_9/boost/1.55.0/x86_64/lib
TACC_BOOST_DIR=/opt/apps/gcc4_9/boost/1.55.0/x86_64

$ export CUDA_HOME=/opt/apps/cuda/6.5
$ export BOOST_HOME=/opt/apps/gcc4_9/boost/1.55.0/x86_64
```

2. SVM functions

Rgtsvm provides three SVM functions as follows:

2.1 Binary classification

Under the binary classification setting, *Rgtsvm* finds the hyperplane that maximizes the margin between two classes. The optimal hyperplane is represented by support vectors and their associated coefficients in the dual space. This procedure is implemented for the users in *Rgtsvm.*

2.2 Multi-class classification

Since SVMs can only solve binary classification problems, multi-class classification is implemented as one against the rest [2] by learning hyperplane for sub-classifiers. In the test time, a voting mechanism is used to determine the most likely class.

2.3 ε-regression

In the ε-regression setting, the response is continuously valued. ε-regression is unique in that the response is regressed in a way that errors within the hyper parameter-control margin are ignored. To our knowledge, *Rgtsvm* is the only open sourced GPU-based SVR.

3. Training workflow

Training and prediction are basic operations in working with SVMs. When implementing a machine learning task, the performance is complicated by the dependence on many factors, such as training samples, feature vectors, training parameters or even the machine learning problem itself. In order to get better model performance, a training workflow summarized in the following figure is usually adopted. The training starts from the splitting of dataset, after which the training dataset is used to obtain a model with optimized parameters while the test dataset serves as an independent evaluation for the trained model. In this section, we will show a demonstration of how to use *Rgtsvm* at each step in figure 1.

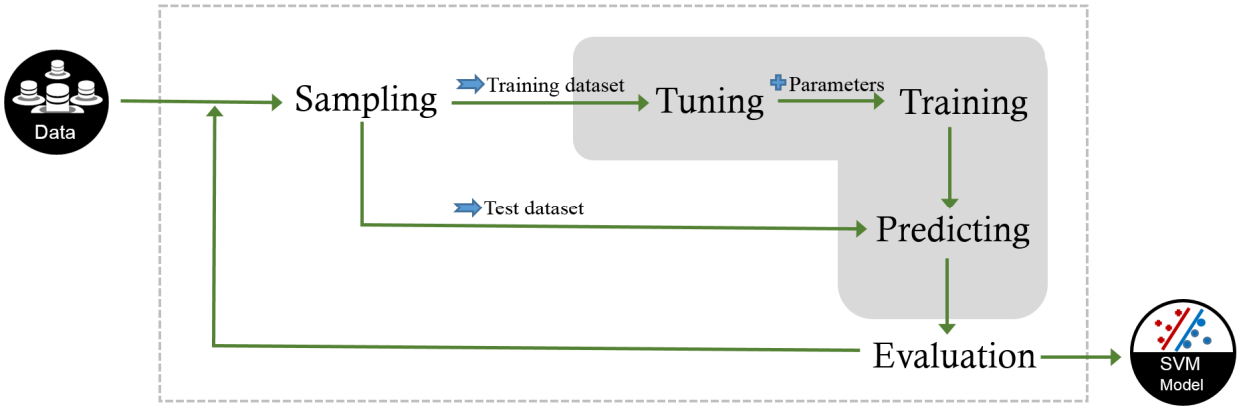

Figure 1: Training workflow

3.1 Data preparation:

We demonstrate the use of *Rgtsvm* using Epsilon [3], a synthetic classification task from the 2008 PASCAL Large Scale Learning Challenge. The training data and test data have 400K and 100K elements with 2000 feature vectors respectively. Because the training data is excessively large to run in the tuning process, we select the test data to for the sake of demonstration. The function `'load.svmlight'` in *Rgtsvm* is used to read the SVM light-formatted file into memory and return a sparse matrix in which the first column indicates label and the feature vectors start from 3rd column.

```
# Downloading epsilon data from
#
https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/epsilon_normalized.t.bz2
# and
# https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/epsilon_normalized.bz2
# and then decompress data using the command 'bunzip2'

# Loading data in the SVM light format.
# Notice: the original training data is too large as a demonstration,
#   therefore we use the test dataset for this example.
dat <- load.svmlight("epsilon_normalized.t")
```

3.2 Sampling

For the balanced data, we randomly sample 60% for training and the remaining 40% for testing. But for the imbalanced data we optimized the ratio of different labels.

```
# Shuffling the indexes and using 60K as training dataset and
#   the reset as test dataset
idx <- sample(1:NROW(dat))
i.train <- idx[1:60000];
i.test <- idx[-c(1:60000)];
```

3.3 Tuning

Rgtsvm provides a tuning function to optimize the parameters in kernel functions, including `coef0`, `gamma`, `degree`, or SVM functions, including `cost`, `epsilon` in regression and `tolerance` of the termination criterion. It does a grid search to find the best parameter combination. Each grid search involves k-fold cross-validation, during which the dataset is divided into k parts and *Rgtsvm* takes each part as test data to evaluate the model trained by the remaining parts. A complete evaluation requires training and prediction *k* times. In *Rgtsvm*, users can speed up the computation with a rough estimation by doing this only on a subset of k parts. This is done by specifying the `rough.cross` parameter.

The following code, illustrates how to optimize the `gamma` parameter and the `cost` parameter within a given range. Noticed that the `cross` parameter and `sampling` parameter is specified inside the `tune.control` function, rather than in the `tune.svm` function.

```
# tuning parameters including gamma and cost
# Notice: the 'cross', cross number and other parameters should be assigned
```

```
#    into 'tune.control' structure.
gt.tune <- tune.svm( dat[i.test,-c(1,2)], dat[i.test,1],
  gamma = 2^seq(-11, -1, 2),
  cost = 10^(-1:1),
  tunecontrol=tune.control( sampling = "cross", cross=8, rough.cross=3),
  scale=F );
```

It might take a while for this tuning process to finish. If this is successful, we get a tuning result which includes two important parts, the best parameters and the best model. The model and parameters are shown as follows:

```
# Printing the tuning results a). best model and b). best parameters
> show(gt.tune$best.model);

Call:
best.svm(x = dat[i.test, -c(1, 2)], y = dat[i.test, 1], gamma = 2^seq(-10,
    -2, 2), cost = 10^(-1:1), tunecontrol = tune.control(sampling = "cross",
    cross = 8, rough.cross = 3), scale = F)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  10
      gamma:  0.0625
    tolerance:  0.001
 time elapsed:  108.16


Number of Support Vectors:  15862

> show(gt.tune$best.parameters);
    gamma cost
14 0.0625   10
```

We may also export the PDF figure (Figure 2) to check the performance using the following codes:

```
# Drawing the figure of tuning results.
pdf("svm-tune.pdf");
plot( gt.tune, transform.x = log2, transform.y = log2)
plot( gt.tune, type = "perspective", theta = 120, phi = 45)
dev.off();
```

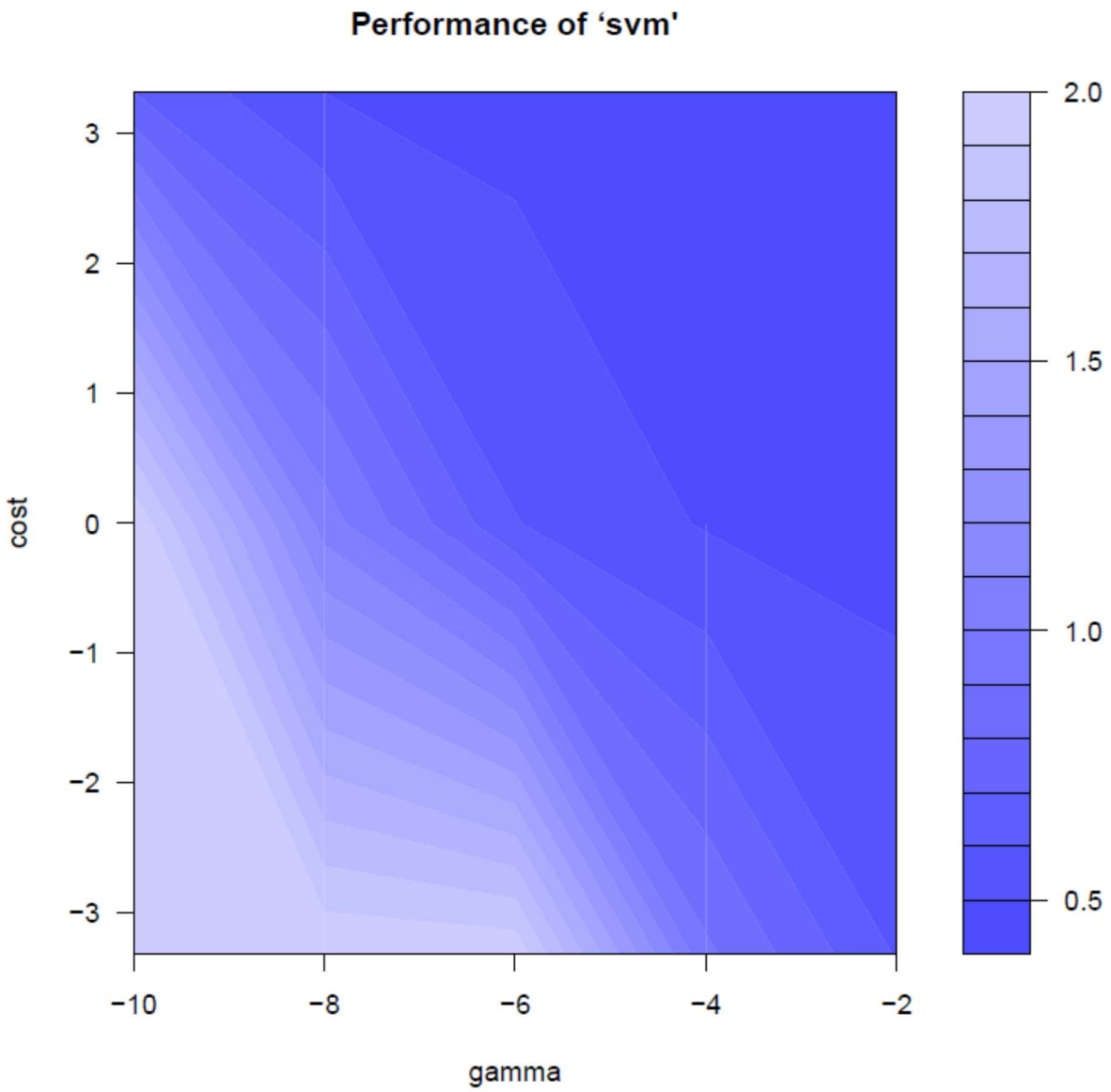

Figure 2: Tuning performance in the example

3.4 Training

The best model in `tune.svm` is tuned by using the k-1 parts of data with the optimized parameters. In order to achieve higher accuracy, we still need to train model using all training data with the optimized parameters as follows:

```
# Training the model using the best parameters.
gt.svm <- svm( dat[i.train,-c(1,2)], dat[i.train,1],
  gamma=as.numeric(gt.tune$best.parameters[1]),
  cost=as.numeric(gt.tune$best.parameters[2]), scale=F );
```

To check the model quality and parameters after training:

```
# Printing the trained model and its accuracy
> show(gt.svm);

Call:
svm.default(x = dat[i.train, -c(1, 2)], y = dat[i.train, 1], gamma =
as.numeric(gt.tune$best.parameters[1]),
    cost = as.numeric(gt.tune$best.parameters[2]))


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  10
      gamma:  0.0625
    tolerance:  0.001
 time elapsed:  195.818


Number of Support Vectors:  22221

> cat("Accuracy for the training data=", gt.svm$fitted.accuracy, "\n");
Accuracy for the training data= 0.90785
```

3.5 Prediction

Prediction is performed on the holdout test dataset. *Rgtsvm* can not only output the classification labels, but also the decision values to facilitate the computation of PR (Precision-Recall) or ROC (Receiver operating characteristic) curves. This can be easily done by specifying the parameter `decision.values` into the prediction call.

```
# Predicting the test dataset, using 'decision.values' to
#   get the decision values
y.pred <- predict( gt.svm, dat[i.test,-c(1,2)], decision.values=TRUE )
```

3.6 Evaluation

Check the accuracy of classification labels with the following code:

```
# Printing the accuracy for the test dataset
> table(y.pred==dat[i.test,1]);

FALSE  TRUE
 4343 35657
> cat("Accuracy for the test data=",
+    length(which( y.pred == dat[i.test,1] ) )/length(y.pred), "\n" );
Accuracy for the test data= 0.891425
```

In general, the SVM package takes 0 as the boundary to decide the binary classification labels, label 1 is returned if the decision value is greater than 0, and -1 otherwise. For an imbalanced dataset or low performance model, we may need to adjust the decision criteria. In these cases, PR curve or ROC curve is helpful to evaluate the optimal decision boundary.

To illustrate the use of PR/ROC curve as a performance metric, we used the PR/ROC library (containing 3 functions). PR and ROC curves (Figure 3) are plotted, with the AUC (Area under the curve) shown as the figure title. The trained model performed well, with AUC greater than 0.95.

```
# Download the script to draw PR or ROC curve to verify the model quality
source("https://raw.githubusercontent.com/andybega/auc-pr/master/auc-pr.r");

# Using the decision values to draw the PR and ROC curve.
str(y.pred);
# Obtaining the decision values from the attribute.
pred<-attr(y.pred, "decision.values");

pdf("svm-eval.pdf");
# Drawing PR curve and calculate the AUC value
xy <- rocdf(pred, obs=dat[i.test,1], type="pr")
AUC <- auc_pr( obs=dat[i.test,1], pred)
plot(xy[, 1], xy[, 2], xlab="Recall", ylab="Precision",
   main=paste("PR AUC=", round(AUC,3)));

# Drawing ROC curve and calculate the AUC value
xy <- rocdf(pred, obs=dat[i.test,1], type="roc")
AUC <- auc_roc( obs=dat[i.test,1], pred)
plot(xy[, 1], xy[, 2], xlab="False Positive Rates",
   ylab="True Positive rates", main=paste("ROC AUC=", round(AUC,3)));

dev.off();
```

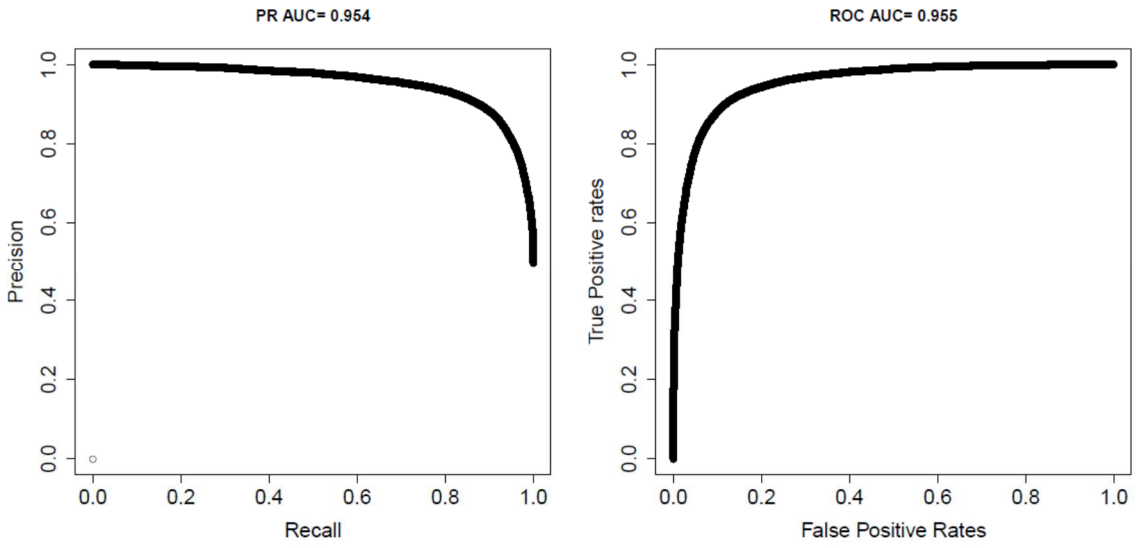

Figure 3: PR and ROC curve in the example

4. Simulation tests

The *Rgtsvm* interface is the compatible with *e1071*. Users may use the same arguments for the "`svm`" function in *e1071* to call *Rgtsvm*. The output of *Rgtsvm* follows a similar layout as the "`svm`" function in *e1071*, greatly facilitating the reuse of scripts written for *e1071*. In the following 3 simulated examples, we compare the performance and accuracy with the package *e1071*.

In the binary classification example, we observe roughly 170 fold speed up in training and 160 fold speed up in predicting. In the regression example, we observe roughly 40 fold speed up in training and 120 fold speed up in predicting.

4.1 Binary classification

The predictor for binary classification example is simulated from two 100-dimensional multivariate normally distributed random variables, each of 50000 examples. Two random variables have the same covariance matrix but differ at their mean value. The response is labeled as 0 and 1 for each class. The dataset for training and testing are divided into 80% and 20% respectively. The accuracy was roughly 90% on the test set.

```
library(Rgtsvm);
library(MASS);
set.seed(1);
size=50000;
dimension=100;

covar.mat <- matrix( runif(dimension*dimension),nrow=dimension);
covar.mat <- t(covar.mat)%*% covar.mat;

zero<- mvrnorm(size,mu=c(1:dimension),Sigma=covar.mat);
one <- mvrnorm(size,mu=c(1:dimension)-5,Sigma=covar.mat);

x <- rbind(zero,one);
y <- c(rep(0,nrow(zero)),rep(1,nrow(one)));

i.all <- 1:(2*size);
i.training <- sample(i.all, length(i.all)*0.8);
i.test <- i.all [! i.all %in% i.training];

model.gpu <- svm(x[i.training,],y[i.training],type="C-
classification");
y.pred <-predict( model.gpu, x[i.test,] );
cat("accuracy", sum(y.pred == y[i.test])/length(i.test),"\n");

#accuracy=0.8997
```

4.2 Multi-class classification

The predictor for multi-class classification example is simulated from three 2-dimensional multivariate normally distributed random variables, each of 3000 examples. Each random variable has its own covariance matrix and mean value. The response is labeled as 0, 1 and 2 correspondingly. The dataset for training and testing are divided into 80% and 20% respectively. The accuracy was roughly 90% on the test set.

```
library(Rgtsvm);
library(MASS);

size=3000
dimension=2

covar.mat0<-matrix(runif(dimension*dimension),nrow=dimension)
covar.mat0<-t(covar.mat0)%*% covar.mat0
covar.mat1<-matrix(runif(dimension*dimension),nrow=dimension)
covar.mat1<-t(covar.mat1)%*% covar.mat1
covar.mat2<-matrix(runif(dimension* dimension),nrow= dimension)
covar.mat2<-t(covar.mat2)%*% covar.mat2

zero<-mvrnorm(size,mu=c(1: dimension),Sigma= covar.mat0)
one<-mvrnorm(size,mu=c(1: dimension)-100,Sigma= covar.mat1)
two<-mvrnorm(size,mu=c(1: dimension)-200,Sigma= covar.mat2)

x<-rbind(zero,one,two)
y<-c(rep(0,nrow(zero)),rep(1,nrow(one)),rep(2,nrow(two)))

all.idx<-1:(3*size)
training.idx<-sample(all.idx, length(all.idx)*0.8)
test.idx<-all.idx[! all.idx %in% training.idx]


model.gpu<-svm(x[training.idx,],y[training.idx],type="C-
classification")
predicted.y<-predict(model.gpu,x[test.idx,])
cat("accuracy", sum(predicted.y==y[test.idx])/length(test.idx),"\n")
#accuracy 1
table(model.gpu$fitted==y[training.idx])
# TRUE
# 7200
```

4.3 ε-regression

The predictor for epsilon regression example is simulated using two 100-dimensional multivariate normally distributed random variables, each of 20000 examples. Two random variables have different covariance matrix and mean value. The response is constructed by injecting normally distributed noise to the Euclidian distance from the predictor to the origin. The dataset for training and testing are divided into 80% and 20% respectively. The Pearson correlation between predicted value and the response was 0.89 on the test set.

```
library(Rgtsvm);
library(MASS);
set.seed(1);
size=20000;
dimension=100;

covar.mat0 <- matrix(runif(dimension*dimension),nrow=dimension);
covar.mat0 <- t(covar.mat0)%*% covar.mat0;

covar.mat1 <- matrix(runif(dimension*dimension),nrow=dimension);
covar.mat1 <- t(covar.mat1)%*% covar.mat1;

zero <- mvrnorm(size,mu=c(1:dimension),Sigma= covar.mat0);
one  <- mvrnorm(size,mu=c(1:dimension)-10,Sigma= covar.mat1);

zero.d <- (zero-matrix(rep(c(1:100), size),nrow=size,byrow=T));
zero.d <-
apply(zero.d,1,FUN=function(x)sum(x^2))+rnorm(n=size,sd=5000);

one.d <- (one-matrix(rep(c(1:100), size),nrow=size,byrow=T));
one.d <- apply(one.d,1,FUN=function(x)sum(x^2))+rnorm(n=size,sd=5000);

x <- rbind(zero,one);
y <- c(zero.d, one.d);

i.all <- 1:(2*size);
i.training <- sample(i.all, length(i.all)*0.8);
i.test <- i.all [! i.all %in% i.training ];

model.gpu <- svm(x[i.training,],y[ i.training ],type="eps-
regression");
y.pred <- predict( model.gpu, x[i.test,] );
cat("correlation=", cor( y.pred, y[i.test]),"\n");
# correlation= 0.8801636
```

5. Batch prediction

Generally, each calling of predict involves initialization of kernel space which can take a long time for a big SVM model. If the predict using same model is called multiple times, the same initialization will be performed repeatedly, which consumes too much GPU time. To reduce the computation time for multiple prediction callings using same big model, *Rgtsvm* introduce new functions based on the current framework. Especially for this case, users firstly run the initialization for the subsequent prediction, and then release the GPU resource and memory after all predictions are finished. The following codes show a very simple application scenario.

```
load( "test/gt.tune.rdata" )
model <- gt.tune$best.model;
m.init <- predict.load(model, 1, verbose=T)
```

```
#Allow prediction is called by multiple times
pred <- predict.run( m.init, model$SV, decision.values = TRUE )

predict.unload( m.init );
```

This feature can be enhanced by using **multiple GPU cards**. In the current version, multiple GPU cards are only supported by the `predict.load` and `predict.run` function. The second parameter of `predict.load` can be assigned to the maximum GPU cards in the computing node, but users should be very cautious about the memory usage.

6. Supporting multiple GPU cards

Rgtsvm supports multiple GPU cards using the following strategies:

1) The training function can be performed only on one GPU card, but the GPU device can be specified by the parameter `gpu.id` or calling the function `selectGPUdevice`.
2) The original prediction function only supports one GPU card, but the GPU device can be specified by the parameter `gpu.id` or calling the function `selectGPUdevice`. It is same as the training function.

```
# Select the 2nd GPU card to run the following codes
# (training or prediction)
selectGPUdevice( 1 );
svm(x, y)

# --------- OR ---------
# Specify the 2nd GPU device (gpu.id=1) in the function svm or predict
svm(x, y, gpu.id=1);
predict(x, y, gpu.id=1);
```

3) The power of batch prediction can be increased by using multiple GPU cards described in section 5.

7. Data structure of trained model

The returned object from "`svm`" function is of class "`gtsvm`". It contains a fitted model. In addition to variables reported in the "`svm`" object returned by *e1071*, "`gtsvm`" also contains several extra variables. Important variables included in class "`gtsvm`" are listed below, with those that are unique to "`gtsvm`" marked by an asterisk.

Items	Description
`SV`	The resulting support vectors (possibly scaled if the raw input is scaled)
`index`	The index of the resulting support vectors in the preprocessed data matrix (after the possible effect of '`na.omit`' and '`subset`')
`coefs`	The corresponding coefficients times the training labels.

*t.elapsed	the user, system and total elapsed time used in GPU computing
*fitted.accuracy	training accuracy in classification task (type="C-classification")
*fitted.MSE	the mean squared error of training (type="eps-regression")
*fitted.r2	equivalent to the squared Pearson correlation coefficient (type="eps-regression")

8. Big training data

To facilitate the training of large datasets, *Rgtsvm* enables the use of the reference class in R to encapsulate the predictor matrix in the form of a pointer. This prevents the training data from being copied multiple times and consuming unnecessary memory space. As a result, the only practical limitation on the number of training examples is the amount of GPU memory.

```
library(Rgtsvm)
library(MASS);

set.seed(1);
size=500000;
dimension=100;

covar.mat <- matrix(runif(dimension*dimension),nrow=dimension);
covar.mat <- t(covar.mat)%*% covar.mat;

zero <- mvrnorm(size,mu=c(1:dimension),Sigma= covar.mat);
one  <- mvrnorm(size,mu=c(1:dimension)-5,Sigma= covar.mat);

x <- rbind(zero,one);
y <- c(rep(0,nrow(zero)),rep(1,nrow(one)));

i.all <- 1:(2*size);
i.training <- sample(i.all, length(i.all)*0.8);
i.test <- i.all [! i.all %in% i.training ];

bigm.x <- attach.bigmatrix( data =x[ i.training,]);
model.gpu <- svm(bigm.x,y[ i.training ],type="C-classification");

y.pred <- predict(model.gpu,x[i.test,]);
cat("accuracy", sum(y.pred==y[i.test])/length(i.test),"\n");
# accuracy 0.929205
```

9. Compatibility with *e1071*

Models returned from *Rgtsvm* are backward compatible with *e1071* and vice versa. The class name for the returned model is called "gtsvm" and "svm" for *Rgtsvm* and *e1071* respectively. Therefore, users

can run prediction on GPU using models trained by *e1071*, simply by changing the class name from "svm" to "gtsvm", e.g.

```
#Example 1, e_svm trained by e1071 is used in Rgtsvm

class(e_svm) <- "gtsvm";
library(Rgtsvm);
y.pred <- Rgtsvm::predict(e_svm, x_test);


#Example 2, gt_svm trained by Rgtsvm is used in e1071

class(gt_svm) <- "svm";
library(e1071);
y.pred <- e1071::predict(gt_svm, x_test);
```

10. Reference

[1] Meyer, D., & Wien, F. T. (2015). Support vector machines. *The Interface to libsvm in package e1071*.

[2] Crammer, K., & Singer, Y. (2001). On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of machine learning research*, *2*(Dec), 265-292.

[3] http://largescale.ml.tu-berlin.de/instructions/