CEGEG076: Spatio-temporal Analysis and Data Mining

Tutorial 6: Support Vector Machines for Regression

11am, DMS Watson G15 - Public Cluster, 29nd February 2016

1. Introduction

In this second tutorial on support vector machines you will learn how to use SVMs for regression, so called support vector regression (SVR). In the same vein as the SVM, SVR enables a form of linear regression to be carried out in feature space, resulting in a nonlinear solution in the input space. The solution is sparse, depending only on the points that lie outside the ε tube, which are the support vectors. The tutorial starts by revisiting the Gauss1 data from the previous tutorial in order to illustrate the way in which the method works. Then, you will the UK temperature data you have used in previous tutorials to build a model to forecast temperatures. You will use the *caret* package to streamline the process of training your model and comparing it with a benchmark model.

2. Revisiting the Gauss1 data

To illustrate the way in which SVR works, we will briefly revisit the Gauss1 dataset that was introduced in the first practical session. The data are shown in Figure 1. First, load the workspace called *RegressionWS*, which contains the required datasets.

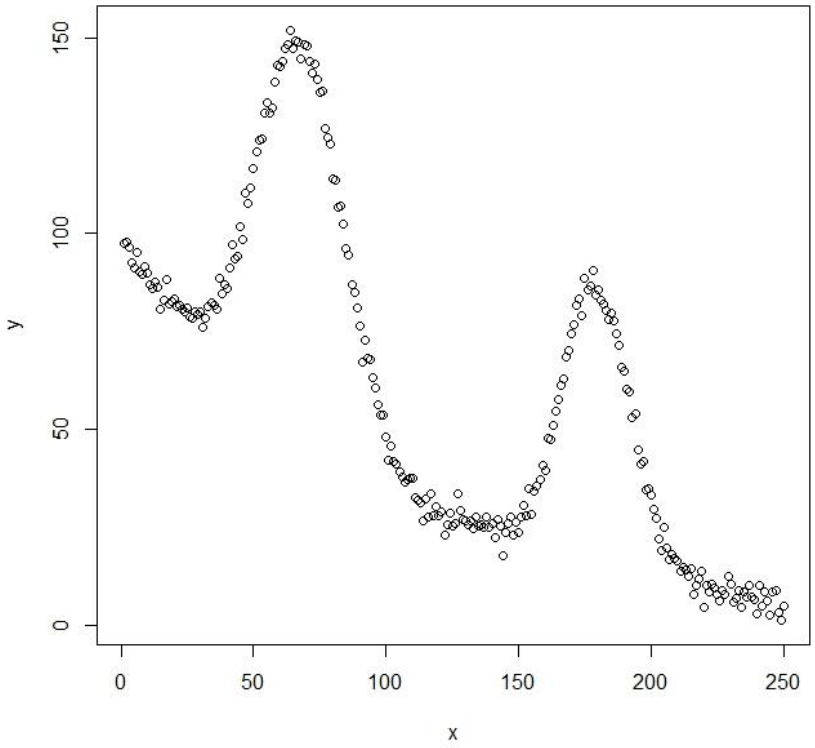

Figure 1 – Gauss1 dataset

Dividde the data into $\mathbf{X}$ and $\mathbf{y}$. We will also set a value of C and epsilon:

```
X <- gauss1[,2] # Create X matrix, including an intercept
y <- gauss1[,1]
C <- 1
epsilon=0.1
```

Next we will fit an SVR model to the data using a linear kernel, called `vanilladot` in kernlab. The linear kernel is simply the Gram matrix $\mathbf{X}\mathbf{X}^{\mathrm{T}}$, so the algorithm will produce a linear solution.

```
library(kernlab)
gauss1Lin <- ksvm(X, y, type="eps-svr", kernel="vanilladot", C=C,
epsilon=epsilon)
gauss1LinPred <- predict(gauss1Lin, X)

plot(gauss1[,c("x","y")], xlab="x", ylab="y", bg="white", cex.axis=2,
cex.lab=2)
lines(gauss1LinPred, col="blue", lwd=4)
```

We use the same command `ksvm` to carry out regression as we do for classification, but change the `type` to `"eps-svr"`. As well as `C`, there is an additional parameter to tune, which is the width of the tube `epsilon`. `C` determines the amount of allowable error in the model, controlling the trade-off between model complexity and error. `epsilon` determines the width of the tube in which we wish to contain the data.

We will now use an RBF kernel and fit a model with the same values of C and epsilon, with automatic sigma estimation:

```
gauss1RBF <- ksvm(X, y, type="eps-svr", kernel="rbfdot", C=C,
epsilon=epsilon)
gauss1RBFPred <- predict(gauss1RBF, X)
lines(gauss1RBFPred, col="red", lwd=4)

legend("topright",legend=c("Linear SVR", "RBF SVR"),
col=c("blue", "red"), lty=1, cex=1.5)
points(gauss1[gauss1RBF@alphaindex,c("x","y")], cex=2, pch=21, col="black",
bg="black")
```

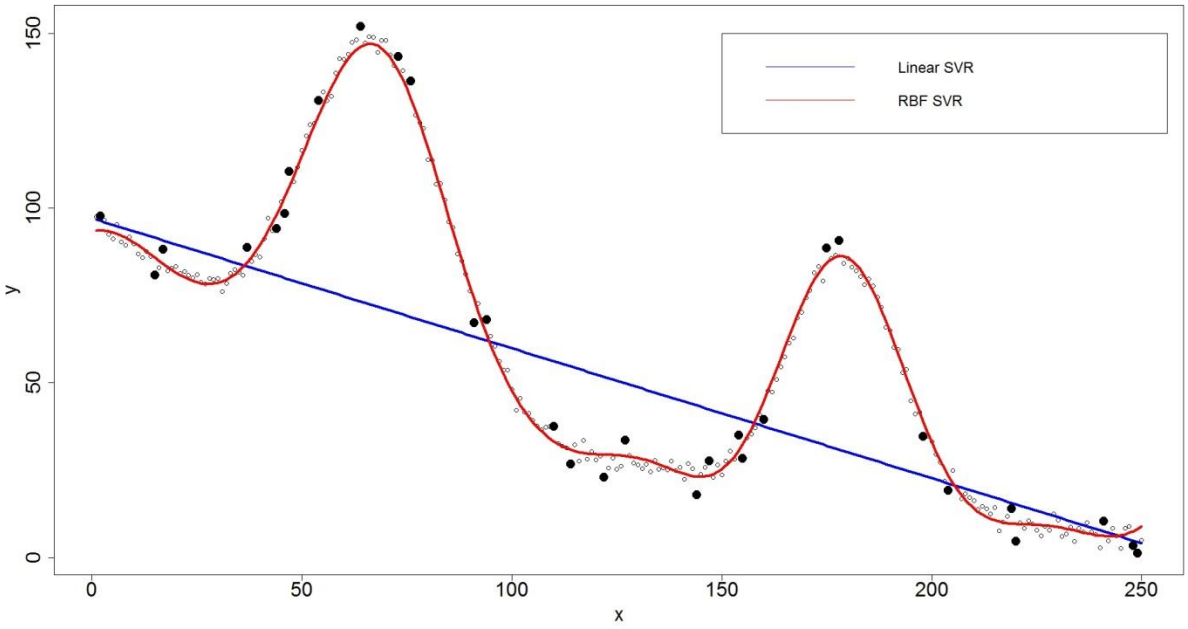

Figure 2 – Linear and nonlinear SVR fits to the Gauss1 dataset

The SVs are shown as dark circles in figure 2. If you check the number of SVs in the nonlinear solution, you will find that only 30 are required, just 12% of the data. This is in contrast to the KRR solution, which requires all of the data points to compute the solution.

3. SVR for forecasting UK temperature

In this part of the tutorial, you will learn how to build a time series SVR model for forecasting of UK temperatures. At this point, we will begin to use a new library called *caret,* which is an incredibly useful library containing a set of functions for streamlining the process of creating predictive models. *caret* inherits from a number of other R libraries including *kernlab, e1701* and *klaR*, allowing their functions to be used in a common framework. Although we focus on SVR in this tutorial, we will use the *caret* library to illustrate the process of testing and evaluating a range of predictive models of different types. The home page of the caret package is here: http://topepo.github.io/caret/index.html

4. The data

The data you will use are the UK temperature data from the earlier tutorials. They are stored in the workspace called *RegressionWS*, which you should already have downloaded from Moodle. By now, you should be familiar with the data.

5. Train an SVR model

We will now build a time series SVR model. To build the model, we will use a temporal autoregressive structure. This means we will forecast the future temperature at a single location (`EngSE`) as a function of a subset of previous temperature observations. We will use the `embed` function to rearrange the series in this way:

```
# embed the data

m <- 3
data <- embed(x = UKTemp[,"EngSE"], dimension = m+1)
```

Next, we will divide the embedded data into training and testing sets. In non-temporal data the training and testing sets are usually divided randomly. However, with temporal data it is important to consider the ordering of time. In this case, we will use the first 80% of the months to train the model, and the remaining 20% of months for testing.

```
# divide data into training and testing sets

n <- nrow(data)
trainYears <- ceiling(n*0.8/12)
split <- 12*trainYears

yTrain <- data[1:split,1]
XTrain <- data[1:split,-1]
yTest <- data[(split+1):nrow(data),1]
XTest <- data[(split+1):nrow(data),-1]
```

To train the model we will use *k*-fold cross validation, with *k* set to 5. In *k*-fold cross validation, the data are randomly partitioned into 5 folds. Each fold is left out in turn and the remaining k-1 folds are used to train a model and predict its values. The selected model is the one with the best average performance across the 5 folds. The procedure prevents overfitting to a particular subset of the

training data. In this case, we do not enforce the ordering of time in the training data because we assume that we have observed the entire training period and those observations of temperature in a particular year are broadly independent of those in other years (note that this may not be the case if there are long term trends). We will use the `caret` package to Initialise the cross validation:

```
library(caret)

ctrl <- trainControl(method = "cv", number=5)
```

Create a grid of parameters to test (at present, caret only allows `sigma` and `C` to be trained in this way so we cannot optimise over `epsilon` automatically) and train the model:

```
SVRGridCoarse <- expand.grid(.sigma=c(0.001, 0.01, 0.1), .C=c(10,100,1000))
SVRFitCoarse      <-      train(XTrain,      yTrain,      method="svmRadial",
tuneGrid=SVRGridCoarse
, trControl=ctrl, type="eps-svr")
```

Here, the method `method="svmRadial"` argument specifies that we wish to build an SVM model with an RBF kernel. Printing the `SVRFitCoarse` object to the console returns a summary of the fitting procedure:

```
SVRFitCoarse

Support Vector Machines with Radial Basis Function Kernel

984 samples
  3 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 787, 786, 789, 787, 787
Resampling results across tuning parameters:

  sigma  C     RMSE      Rsquared   RMSE SD     Rsquared SD
  0.001    10  1.929628  0.8370670  0.06994068  0.01468750
  0.001   100  1.904022  0.8403263  0.06598712  0.01514779
  0.001  1000  1.887274  0.8431277  0.05396167  0.01313413
  0.010    10  1.863295  0.8470090  0.05526972  0.01336407
  0.010   100  1.834758  0.8517749  0.04705413  0.01143042
  0.010  1000  1.771537  0.8617942  0.05516890  0.01161515
  0.100    10  1.688152  0.8741137  0.07130869  0.01290526
  0.100   100  1.687310  0.8738326  0.07267955  0.01277163
  0.100  1000  1.706251  0.8708839  0.08385032  0.01401683

RMSE was used to select the optimal model using  the smallest value.
The final values used for the model were sigma = 0.1 and C = 100.
```

Results are given for each of the parameter combinations in terms of the root mean squared error (RMSE) and the R squared indices, including their standard deviations. We can view the results visually using the `plot` function (figure 3):

```
plot(SVRFitCoarse)
```

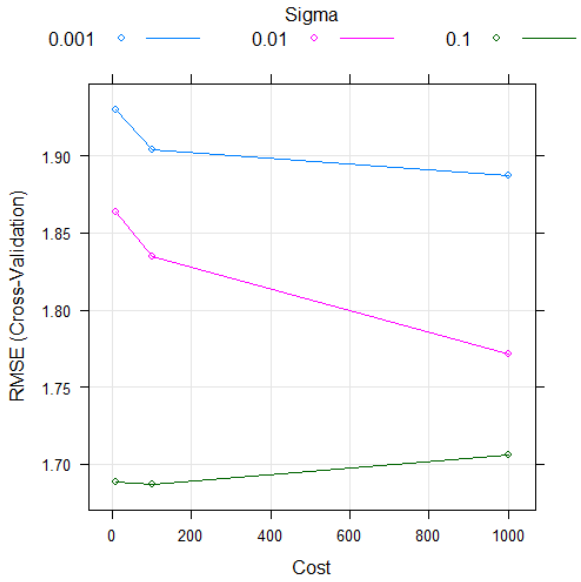

Figure 4

3 – Training errors for the SVR model with different parameter combinations

From this, we can see that the value of `sigma=0.1` and `C=100` provide the best model performance. We can refine the gird to see if we can gain further improvements in performance:

```
SVRGridFine <- expand.grid(.sigma=c(0.05, 0.1, 0.15), .C=c(1,10,50))
system.time(SVRFitFine <- train(XTrain, yTrain, method="svmRadial",
tuneGrid=SVRGridFine,
trControl=ctrl, type="eps-svr"))
plot(SVRFitFine)
```

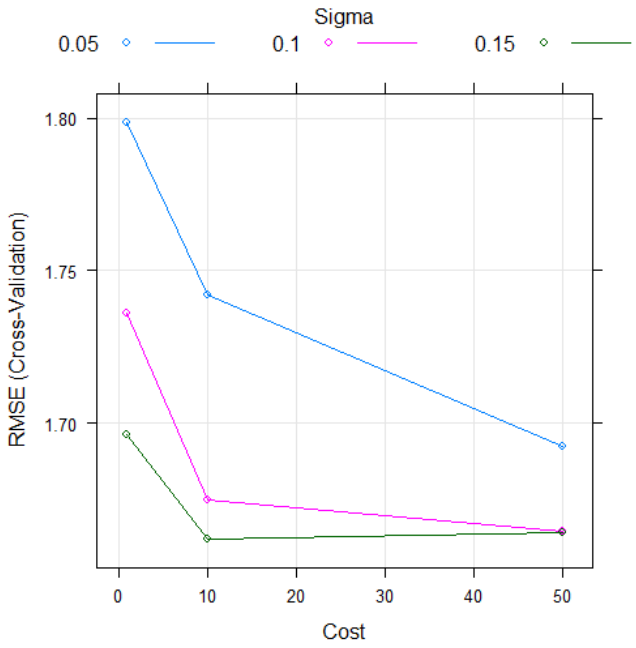

Figure 4 – Training errors for the SVR model: further investigation of C

Examining figure 4, we can see that the value of `C=10` provides the best model performance. The best value of sigma is, `sigma=0.15`. It can be seen that the lowest error obtained lies within the range of the parameters we have tested, so we can stop here. In practice, we would most likely have more time and would start with a larger grid of parameters. Examine the properties of the fitted model:

```
names(SVRFitFine)
```

```
[1] "method"       "modelInfo"    "modelType"    "results"      "pred"
[6] "bestTune"     "call"         "dots"         "metric"       "control"
[11] "finalModel"   "preProcess"   "trainingData" "resample"
"resampledCM"
[16] "perfNames"    "maximize"     "yLimits"      "times"
```

The model object for the best model can be accessed by:

```
SVRFitFine$finalModel
```

```
Support Vector Machine object of class "ksvm"

SV type: eps-svr  (regression)
 parameter : epsilon = 0.1  cost C = 10

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.15

Number of Support Vectors : 759

Objective Function Value : -1759.657
Training error : 0.116715
```

Notice that although we trained the model using methods from the *caret* package, the returned model is a `ksvm` object from *kernlab*. Therefore we can access all its elements as before:

```
SVRFitFine$finalModel@nSV
2219
```

Approximately 77% of the points are used as support vectors. If you have time, try producing a plot that shows the position of the support vectors in the series like those you produced for the classification models. We can use the model for one-step-ahead prediction and plot the results:

```
yPred <- predict(SVRFitFine, XTest)
plot(yTest, type="l", xaxt="n", xlab="Date", ylab="UKTemp")
points(yPred, col="blue", pch=21, bg="blue")
#axis(1, at=seq(90, (180*5)+90, 180),
labels=unique(dates[(split+1):nrow(data),1]))
axis(1, at=seq(6, (12*19)+6, 12),
labels=format(dates[(trainYears+1):length(dates)], "%Y"))
legend(90, 3, legend=c("Observed"), lty=1, bty="n")
legend(150, 3, legend=c("Predicted"), pch=21, col="blue", bg="blue",
bty="n")
```

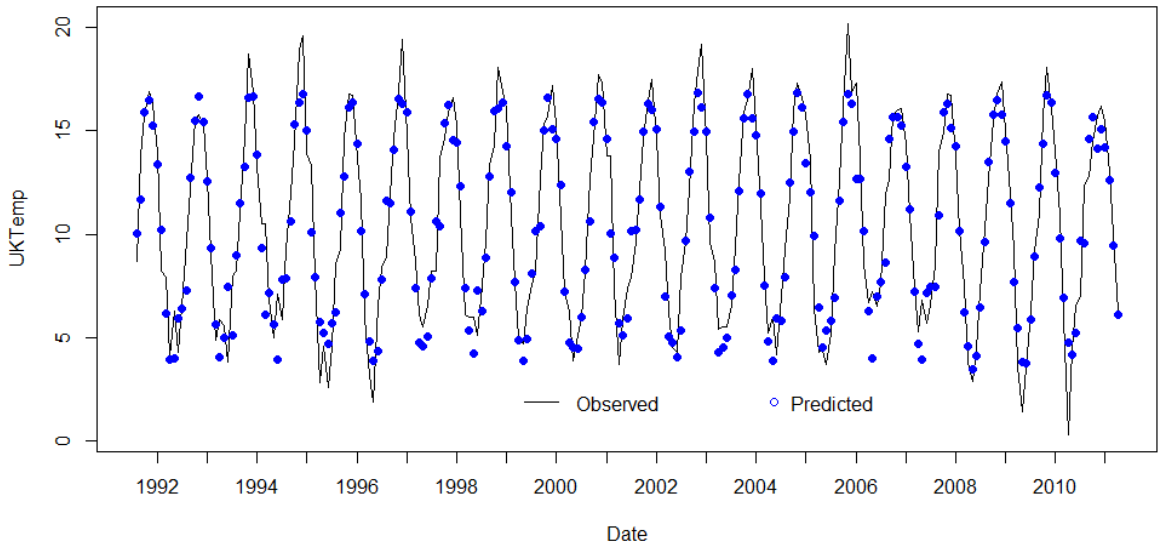

Figure 5 – Observed vs predicted values for the SVR model

Figure 5 shows the observed versus predicted values for the SVR model. It can be seen that the model forecasts the temperatures reasonably well and is able to follow the seasonal trend. It is good practice to also check whether any autocorrelation remains in the residuals of the model:

```
SVRResidual <- yTest-yPred
plot(SVRResidual)
acf(SVRResidual)
```

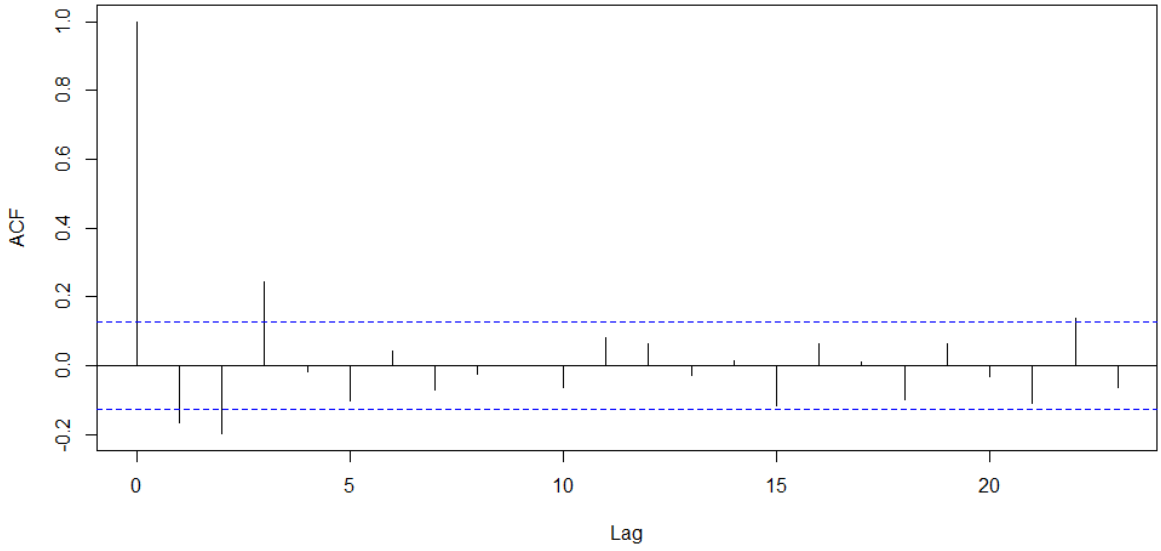

Figure 6 – Residual autocorrelation for SVR model

It can be seen from figure 6 that the residual autocorrelation is close to the 95% confidence limit (blue dashed lines), but some significant autocorrelation remains. One of the drawbacks of machine learning algorithms is that they are not designed to deal with autocorrelation directly.

6. Creating a space-time model

The previous example used time series only and was essentially a temporal autoregressive model like those introduced in the time series modelling content of the course. In this example, we will create a space-time model using the spatial weight matrix that you defined for the temperature data previously. We will use a function called `st_embed`, which is in the workspace you have loaded. `st_embed` embeds the series of a particular location along with its adjacent neighbours:

```
stseries <- st_embed(data= UKTemp, m=3, col=1, W=W, ii=TRUE)
```

The first two arguments are the same as the embed function. The `col` argument specifies the column of the data matrix we wish to model. `W` is the spatial weight matrix. `ii` specifies whether we want to include the data of the location `col` in the series. In this case, we want to model the location as a function of its own previous values and those of surrounding regions, so we set `ii` to be `true`. We can then create the training and testing data.

```
stXtr <- stseries$X[1:1104,]
stXts <- stseries$X[1105:nrow(stseries$X),]
stytr <- stseries$y[1:1104]
styts <- stseries$y[1105:nrow(stseries$y)]
```

We can now train a space-time SVR model with a single kernel based on these training and testing data:

```
stmodel <- ksvm(x=stXtr,y=stytr,type="eps-svr",kernel="rbfdot",
kpar=list(sigma=0.1), C=10, epsilon=0.1, cross=5)
```

Now we can predict the testing data using this model:

```
stforecast <- predict(stmodel, stXts)
```

We will calculate the errors directly by defining our own rmse function. You can define any error measure you may wish to use in the same way:

```
rmse <- function(obs, pred, na.rm=TRUE)
{
   sqrt(mean((obs-pred)^2, na.rm=na.rm))
}

errorSTs <- rmse(stforecast, styts)
errorSTr <- rmse(stytr, predict(stmodel, stXtr))
```

How do the errors from the space-time model compare with the error of the temporal autoregressive model? Is this a more accurate model? (Note: At this stage, it is important to bear in mind that we have only tested one parameter combination for each of these models). How do the results compare to those of the models you have tested in previous weeks? Try using `caret` to find the best ST model. Also, think about how you might make a model for all the links at once.

7. Summary

In this tutorial you have learnt the basics of how to train SVR models for forecasting space-time series using the *caret* library. *caret* makes the process of training machine learning models simple and facilitates easy comparison between different methods. However, because *caret* is a general

purpose package, it does not automatically do everything you may want it to. For example, the `"svmRadial"` option does not allow epsilon to be added to the `trainGrid` object. Furthermore, in time series forecasting one may want to vary the embedding dimension, which *caret* does not do automatically. If you have time, try some of the following:

- Use caret to train a model to classify the crop yield or landslide susceptibility data from the classification tutorial.
- Experiment with different values of the embedding dimension and epsilon.
- Train a model for some more regions.