

gretl + SVM

Allin Cottrell

April 4, 2024

1 Introduction

This is documentation for a `gretl` function named `svm`, which offers an interface to the machine-learning functionality provided by `libsvm` (SVM = Support Vector Machine). We assume that the reader knows at least a little about machine learning and how it relates to econometrics. If this assumption does not hold please take a look at [Mullainathan and Spiess \(2017a\)](#), or google the topic if you prefer, and come back when you're ready.¹

`Libsvm` is an open-source library (see <https://www.csie.ntu.edu.tw/~cjlin/libsvm/>) and `gretl` (as we suppose you know) is an open-source, cross-platform econometrics package (<http://gretl.sourceforge.net/>). Support for `libsvm` in `gretl` was added in the 2017d release (2017-11-07). Since then, however, some improvements have been made and at present you'll be better off using `gretl` 2019a or higher to work with this.

In section 2 we outline the `gretl svm` function and illustrate its usage via a few example scenarios; section 3 goes into more detail on the supported options. Section 4 discusses cross-validation; section 5 gives a classification example; section 6 discusses probability estimation; and section 7 provides some details on the implementation of `libsvm` in `gretl`.

2 The `svm` function

The signature of `svm` is as follows:

```
series svm(list L, bundle bparms, bundle *bmod[null], bundle *bprob[null])
```

That is, this function returns a series (predictions) and it has two required arguments, a list (of series) and a bundle containing zero or more parameter specifications. The third and fourth arguments take the form of “pointers to bundle”; they are optional, and may be set to `null` or omitted altogether; we illustrate their use below but for now we just note that they offer means of ferrying additional information from `svm` to the user or vice versa.

The list argument to `svm` works like the list argument to regression commands in `gretl`: it should contain the dependent variable first, followed by the independent variables. Note that there's no point in including an intercept (the series `const` or 0) in `L`; it will just be dropped by `libsvm`.

The `bparms` bundle may contain quite a number of parameter specifications and other options but most of these have default values which may be acceptable, in which case you can get by with a minimal set. A listing is given in Table 1 and usage is discussed in section 3.

The workflow for supervised machine learning goes like this:

1. Gather suitable data and divide them into a training set and a testing or “holdout” set.

¹For a good account of the technical background on SVMs, see [Smola and Schölkopf \(2004\)](#).

2. Build a model using the training data and assess its fit.
3. Using this model, generate predictions for the testing data and assess the fit.

Then if the out-of-sample performance is deemed satisfactory, the model may be used to generate “live” predictions for additional data.

Gretl’s `svm` is set up to perform steps 2 and 3 above with ease (though it can also do other things). We’ll walk through this sort of basic usage in the first scenario below.

2.1 First scenario

Suppose you have an “integrated” dataset, comprising n training observations followed by m testing observations, and this dataset is loaded in gretl. You construct a list `L` as described above to pass as the first argument to `svm`. In the simplest case, the only parameter you need include in the bundle to pass as the second argument is `n_train`, which sets the number of observations to use for training (assumed to be at the start of the dataset); for example, if you have 5000 training observations,

```
bundle b = defbundle("n_train", 5000)
series yhat = svm(L, b)
```

Here’s what happens by default.

1. Gretl examines your dependent variable (the first member of `L`): if this is a binary dummy, or its “coded” attribute is set, the C -SVC SVM type is selected (classification), otherwise ϵ -SVR (regression) is selected. In both cases the default kernel is RBF (radial basis function).
2. Gretl finds the maxima and minima of the independent variables in the sample range 1 to `n_train` and scales them onto $[-1, 1]$.
3. The `libsvm` training function is called, with all parameters at their default values as shown in Table 1. The resulting model is stored in memory.
4. The `libsvm` prediction function is called on the training range, and the degree of fit is shown (for classification, the count and percentage of correct predictions; for regression the MSE and R^2).
5. If all has gone OK, gretl checks if you have enough observations for a testing phase—that is, somewhat arbitrarily, there are at least 10 observations between `n_train` and the end of the current sample range. (One would generally expect there to be a good deal more testing observations.) If so, the independent variables in the test set are scaled using the ranges saved from the training data, a second prediction call is made, and the out-of-sample fit is shown.

We’ll give a real example, replication of the Mullainathan and Spiess study referenced in the Introduction. It’s not an exact replication since the authors used different machine-learning software (a collection of R tools), but we can replicate their baseline OLS results and our `libsvm` results are comparable to those shown in their article for machine learning.

A `hansl` script for this purpose is shown in Listing 1. We discuss it below.

In BLOCK 1 of the Mullainathan–Spiess script we open a data file named `ahs_jep.gdtb`. This is a native gretl version of the dataset used by the authors, which they created from “raw” American Housing Survey data with the help of R. Our version is a translation of the R data; it is available for download as

http://gretl.sourceforge.net/svm/ahs_jep.gdtb

Listing 1: Mullainathan and Spiess replication script

http://gretl.sourceforge.net/svm/MS_simple.inp

```
set verbose off

# helper function: get R^2 and MSE
function matrix get_stats (const series y, const series yhat)
    scalar SSR = sum((y - yhat)^2)
    return {1 - SSR/sst(y), SSR / $nobs}
end function

# BLOCK 1: open and arrange data
open ahs_jep.gdtb -q
rename LOGVALUE y
dataset sortby holdout
list X = dataset
list drop = y folds8 holdout
X -= drop
X = cdummify(X)
ntrain = 10000
test1 = ntrain + 1

# BLOCK 2: OLS baseline
smpl 1 ntrain
X = dropcoll(X)
ols y 0 X --quiet
printf "OLS: training R^2 = %.3f, MSE = %.3f\n", $rsq, $ess/$nobs
smpl test1 $tmax
yhat = lincomb($xlist, $coeff)
m = get_stats(y, yhat)
printf "OLS: holdout R^2 = %.3f, MSE = %.3f\n", m[1], m[2]

# BLOCK 3: SVM training and testing
smpl full
bundle parms = defbundle("n_train", ntrain, "quiet", 1)
list L = y X
series svmpred = svm(L, parms)
smpl 1 ntrain
m = get_stats(y, svmpred)
printf "SVM: training R^2 = %.3f, MSE = %.3f\n", m[1], m[2]
smpl test1 $tmax
m = get_stats(y, svmpred)
printf "SVM: holdout R^2 = %.3f, MSE = %.3f\n", m[1], m[2]
```

The dataset contains the log-values of 51,808 housing units along with 162 covariates, most of which are encodings of qualitative characteristics. 10,000 of the observations are designated for training, leaving 41,808 for testing.

We begin by renaming, for the sake of brevity, the dependent variable LOGVALUE as `y`. The dummy variable `holdout` has value 1 for observations in the testing set, 0 for observations in the training set. In the original data these are interspersed, but for gretl's `svm` we want all the training observations to come first, hence the line

```
dataset sortby holdout
```

Then, since many of the independent variables are encodings (“factors,” in R parlance), we turn them into sets of 0/1 dummies using the `cdummi fy` function: the list `X` then contains 338 series.

In BLOCK 2 of Listing 1 we first set the sample to the training data only and purge the list `X` of perfectly collinear terms, leaving 274 series.² We then estimate a linear model via OLS. After switching to the holdout sample, we use the OLS parameter estimates (`$coeff`) to generate fitted values using the `lincomb` function, and calculate the measures of fit employed by Mullainathan–Spiess (hereinafter M-S).

In BLOCK 3 we come to the SVM part. This is straightforward (though it takes a while to run). We tell `svm` how many observations to use for training, and (optionally) tell it to be quiet. The remainder of this block just uses the original `y` data and the predictions from `svm` (which we name `svmpred`) to calculate the measures of fit in the training and testing sub-samples.

The output obtained from running this script should closely resemble the following (which was obtained in 70 seconds on a quad-core desktop with Intel i7 Haswell processors running Arch Linux):

```
OLS: training R^2 = 0.473, MSE = 0.589
OLS: holdout  R^2 = 0.417, MSE = 0.674
SVM: training R^2 = 0.494, MSE = 0.565
SVM: holdout  R^2 = 0.448, MSE = 0.639
```

The OLS R^2 values agree with those shown in Table 1 of M-S, which inspires confidence that we’re really working with the same data. M-S show results from four machine-learning algorithms: (1) Regression tree tuned by depth, (2) LASSO, (3) Random forest (a linear combination of trees) and (4) Ensemble (a weighted combination of the first three methods). Our “out of the box” `libsvm` results are in the middle in terms of out-of-sample (or “holdout”) fit: better than (1) and (2) but not quite as good as (3) or (4), which have R^2 values of 0.455 and 0.459 respectively as against our 0.448.

In section 4.7 below we follow up on this in light of cross validation.

2.2 Second scenario

Say you have separate data files for training and testing and you don’t wish to combine them. You’d like to do training on the first dataset and testing on the second.

In the first variant of this scenario we assume you’re nonetheless willing to do the two operations in the context of a single `hansl` script. In that case your solution (or a bare-bones version of it) is shown in Listing 2. Note that in this script (and others to follow) we construct the list to pass to `svm` as simply

```
list L = dataset
```

²These terms would have been dropped automatically by the `ols` command, but we also want to remove them from the list passed to `svm`.

This works if (a) your dependent variable occupies position 1 in the dataset and (b) you wish to use all the other series in the dataset as supports. Otherwise you need to compose L yourself.

Listing 2: Scenario 2: using two data files in one script

```
open train.gdt
list L = dataset
bundle b = defbundle("ranges_outfile", "train.ranges")
bundle savemod = defbundle()
svm(L, b, &savemod)

open test.gdt --preserve
list L = dataset
bundle b = defbundle("ranges_infile", "train.ranges", "loadmod", 1)
svm(L, b, &savemod)
```

In Listing 2 we save the ranges (for scaling) to file in the training run, and also use the third argument to `svm` to save the model as a bundle (namely `savemod`). After opening the test data file (with the `--preserve` option to avoid destroying `savemod`) we then load the ranges from file, and also load the saved model bundle via `svm`'s third argument.³

OK, but what if you want two separate scripts, to be run on distinct occasions? Then try Listing 3. In this case we save both the ranges and the model to file in the first script, then reload them in the second.

Listing 3: Scenario 2: using two separate scripts

```
# script 1
open train.gdt
list L = dataset
bundle b = defbundle("ranges_outfile", "my.ranges", "model_outfile", "my.model")
svm(L, b)

# script 2
open test.gdt
list L = dataset
bundle b = defbundle("ranges_infile", "my.ranges", "model_infile", "my.model")
svm(L, b)
```

3 Options

Table 1 shows the options that can be set via the parameter bundle passed to `svm`—other than those pertaining to cross-validation, for which see section 4.

As mentioned above, `gretl` selects a default SVM type based on the character of the dependent variable. However, this can be overridden by including an appropriate integer code under the key `svm_type`—see panel (a) of the table—as in

³See section 3 for explanation of usage of this argument as well as other options illustrated in these examples.

(a) Values for `svm_type` (default automatic):

<i>code</i>	<i>string</i>	<i>SVM</i>	<i>comment</i>
0	C-SVC	C-SVC	multi-class classification
1	nu-SVC	ν -SVC	multi-class classification
2	one-class	one-class SVM	
3	eps-SVR	ϵ -SVR	regression
4	nu-SVR	ν -SVR	regression

(b) Values for `kernel_type` (default 2, RBF):

<i>code</i>	<i>string</i>	<i>kernel</i>	<i>comment</i>
0	linear	linear	$u'v$
1	polynomial	polynomial	$(\gamma u'v + c_0)^d$
2	RBF	Radial Basis Function	$\exp(-\gamma \ u - v\ ^2)$
3	sigmoid	sigmoid	$\tanh(\gamma u'v + c_0)$
4	TBA	TBA	

(c) Additional libsvm parameters:

<i>key</i>	<i>parameter controlled</i>
degree	degree of polynomial (d) in kernel function (default 3)
gamma	γ in kernel function (default $1/\#$ of covariates)
coef0	c_0 in kernel function (default 0)
C	the cost parameter C of C-SVC, ϵ -SVR, and ν -SVR (default 1)
nu	the parameter ν of ν -SVC, one-class SVM, and ν -SVR (default 0.5)
toler	tolerance of termination criterion (default 0.001)
epsilon	the ϵ in loss function of ϵ -SVR (default 0.1)
cache_size	cache memory size in MB (default 1024)
shrinking	use shrinking heuristics, 0 or 1 (default 1)
probability	produce probability estimates, 0 or 1 (default 0)

(d) Additional gretl-specific parameters:

<i>key</i>	<i>comment</i>
n_train	integer: the number of training observations
scaling	0 = none, 1 = $[-1, 1]$, 2 = $[0, 1]$
loadmod	0/1: load model from bundle-pointer
predict	0, 1 or 2: do prediction if relevant (see text)
quiet	0/1: suppress printed output from svm
model_outfile	filename for writing model
model_infile	filename for reading model
ranges_outfile	filename for writing data ranges
ranges_infile	filename for reading data ranges
data_outfile	filename for writing data values
range_format	string: libsvm or gretl

Table 1: Parameters and options that may be set using the second (bundle) argument to the `svm` function

```
bundle parms = defbundle()
parms.svm_type = 4 # select nu-SVR
```

We also mentioned that gretl defaults to the RBF kernel; this can be overridden by use of the `kernel_type` key:

```
parms.kernel_type = 3 # sigmoid, unlikely to be superior!
```

3.1 Libsvm options

Panel (c) of the options Table shows additional options for controlling the behavior of libsvm. We discuss the `epsilon` option in section 4.5 and the `probability` option in section 6. For further details please see the libsvm documentation, [Hsu *et al.* \(2016\)](#).

All but one of the default values shown are as per libsvm itself, the exception being `cachesize`, the size of the memory cache for use in training, in megabytes. Training can take quite a while on a big dataset, and a large cache is likely to speed things up. The libsvm default value is 100 MB, but many computers today can support a gigabyte of cache so we raised this to 1024. If your machine's memory is limited you may need to reduce this, for example

```
parms.cachesize = 200
```

3.2 Other options

That leaves the gretl-specific settings in panel (d) of the options Table, some of which may require more explanation.

`n_train`: has already been discussed: it simply tells gretl how many training observations can be found at the start of the dataset. But in this context “the start” means the first usable observation. If your dataset is complete (no missing observations) then specifying `n_train = 5000` will result in the use of observations 1 to 5000 for training. But if there are, say, 10 missing observations at the beginning of the data, the training range will then be 11 to 5010.

`scaling`: is the data-scaling option: 0 means that no scaling should be done (not advisable unless the input data are already suitably scaled); 1 means to scale to a range from -1 to $+1$ (the default); and 2 means to use a range of 0 to 1. At present we do not support user-specified lower and upper limits, and neither do we support scaling of the dependent variable (the above applying only to the independent variables).

`loadmod`: this binary variable governs the interpretation of the optional third (pointer-to-bundle) argument to `svm`. By default it is assumed that this should be used to save an SVM model in the form of a bundle (an auxiliary “return” value, if you will). Note that this overwrites any prior content the bundle may have had. By giving `loadmod` a non-zero value you are telling gretl to expect a model bundle on input instead (and not to overwrite it).

`predict`: this governs how much prediction `svm` does. The default is 2, meaning that if a model is built then prediction should be done on both the training data used to build the model (to get a measure of fit) and also on any testing data present. By setting `predict = 1` you are requesting that prediction be done for (at most) only the training. By setting it to 0 you are telling gretl not to make any calls to the libsvm prediction function (or none beyond those required by cross validation—see section 4).

The several `outfile` and `infile` keys can be used to supply filenames for writing or reading, respectively, certain sorts of data. Only if such names are given does `svm` write or read any files; by default everything goes on in memory.

`model_infile` can be used to get gretl to read a previously trained libsvm model file (which implies that `svm` will skip the training step), and `model_outfile` can be used to produce a file readable by the `svm-predict` executable. The `ranges` keys work in a similar manner, for

files containing libsvm data-range information. However, gretl's own range files contain more information than standard libsvm ones and cannot be read by the `svm-` executables, so if you wish to output a ranges file readable by them you must in addition specify

```
parms.range_format = "libsvm"
```

(substituting the name of your parameter bundle). There is at present no facility for the converse—that is, getting gretl's `svm()` to read a ranges file written by `svm-scale`. Attempting to do so will generate an error, regardless of the setting of `range_format`.

Finally, `data_outfile` can be used to write data in the format required by `svm-train` and `svm-predict`. Note that there is no `data_infile` key: `svm` only accepts data passed from a gretl dataset via its first (list) argument.

4 Cross validation

Libsvm provides a cross-validation function that works as follows. Given a number, $v \geq 2$, of “folds,” the data are divided randomly into v equally sized subsets,⁴ then for each subset i a sub-model is trained on the complementary subset of the data (the other $v - 1$ folds) and predictions are generated for subset i . Once this process is complete we have “pseudo out of sample” predictions for all the data supplied (which should be either the full training dataset or perhaps a subset thereof), and suitable figures of merit can be calculated for these predictions.

Here's the context in which this can be particularly useful: when libsvm trains a model it automatically adjusts a (possibly large) number of coefficients, but it takes certain kernel parameters as given. For the recommended RBF kernel this includes C and γ ; for the SVM type ϵ -SVR it also includes ϵ ; and for ν -SVC, ν -SVR and one-class classification it includes ν . It may be that prediction can be improved by tuning these parameters. However, simply tuning on the full training dataset is likely to result in over-fitting, to the detriment of prediction on the testing data. So the recommended procedure is:

1. Run a line or grid search over ranges of values for one or more of the above-mentioned parameters, on each iteration calculating a suitable figure of merit via cross validation.
2. Thereby identify the parameter value(s) that produce the best cross-validation results (the maximum percentage of correct predictions for classification and, by default, minimum MSE for regression—but see section 4.2 below).
3. Use the optimized parameters to train a model on the full training dataset, and then use this model to generate predictions for the testing set.

It is still possible that this will result in over-fitting to the training data, but that risk should at least be lessened.⁵

The cross-validation options supported by gretl's `svm` are shown in Table 2.

We have more to say about specification of the folds in section 4.3 below.

Note that only one of `search` and `grid` should be given. And if either of these is given, the `folds` option need not be supplied: in its absence the number of folds defaults to 5.

By default grid ranges are specified in terms of the base-2 logs of the parameters.⁶ The `svm` default grid is based on that used by the `grid.py` tool supplied with the libsvm distribution:

⁴That is by default, but see also section 4.3 below.

⁵For further discussion see section 3 of Hsu *et al.* (2016).

⁶But see section 4.1 below.

	<i>key</i>	<i>type and effect</i>
	<code>fold</code> s	integer ≥ 2 : number of folds (default 5)
<i>or</i>	<code>foldvar</code>	0/1: flags inclusion of a folds variable (see text)
	<code>search</code>	0/1: parameter search using default grid
<i>or</i>	<code>grid</code>	matrix: user-specified grid
	<code>search_only</code>	0/1: see text
	<code>contiguous</code>	0/1: see section 4.3
	<code>seed</code>	integer: see section 4.4
	<code>refold</code>	0/1: see section 4.4
	<code>regcrit</code>	integer: see section 4.2

Table 2: Cross-validation options in `svm` parameter bundle

	<i>start</i>	<i>stop</i>	<i>step</i>
C	-5	9	2
γ	3	-15	-2

This means that C will range from 2^{-5} to 2^9 , quadrupling at each step, while γ will range from 2^3 down to 2^{-15} , shrinking by a factor of 4 at each step. Note that implicitly ϵ or ν (where applicable) are clamped at their initial values (whether their defaults or user-specified).

A user-defined matrix under the `grid` key should conform to this general pattern, but with some flexibility: it should have from 1 to 3 rows and either 3 or 4 columns.⁷ The first row is assumed to pertain to C ; the second row, if present, to γ ; and the third, if present, to ϵ or ν . If you just want a line-search for C a one-row matrix will suffice. If you want to search for C and ϵ , with γ clamped, you need a three-row matrix: a row of zeros (in this example, the second or γ row) indicates a clamped value.

The `search_only` option has the effect of stopping `svm` from doing any training or prediction following parameter tuning.

To be clear, the number of observations used in tuning—call this T_0 —is either the size of the current sample range, as set by the `smp1` command prior to calling `svm`, or `n_train` if that is specified in the parameter bundle. Once tuning is completed, by default `svm` trains a model on all T_0 observations using the optimized parameters, then performs prediction for any additional data (i.e., from $T_0 + 1$ to the end of the sample range).

You should define `search_only` if you just want to find and save the “best” parameter values. As an aid, setting this flag has an additional effect: if a pointer-to-bundle is supplied as the third argument to `svm`, a matrix named `xvalid_results` is written into it. (Any other content of the bundle is not touched.) This matrix has either three or four columns and as many rows as there are parameter combinations. Columns 1 and 2 hold, respectively, the values of C and γ . If the model uses the parameter ϵ or ν this occupies the third column; the last column holds the associated value of the search criterion. If `search_only` is the only cross-validation option given, this implies use of the default grid and 5 folds.

Note that grid search with cross validation can be very expensive on a big dataset. The default grid has 8×10 pairs of (C, γ) values; with 5 folds this means 400 training runs. Whatever timing you find for a single training run, you can expect a default grid search to take longer by two orders of magnitude. Moreover, training involves more computation the greater the value of the cost factor, C , so if your search space includes large values of C the time taken by cross validation may be three orders of magnitude more than just training on a set of parameters in

⁷The use of a 4th column is explained in section 4.1.

the neighborhood of the default values.

Listing 4 illustrates usage of cross validation in `svm`.

Listing 4: Grid search with cross validation

```
# script 1: just get optimized parameters using 1000
# training observations with custom grid
open data.gdt
list L = dataset
matrix gmat = {-3,7,1; 1,-13,-2}
smp1 1 1000
bundle b = defbundle("grid", gmat, "search_only", 1)
bundle bestparms = defbundle()
svm(L, b, &bestparms)
print bestparms

# script 2: integrated search, training, prediction, with
# default grid and 5 folds
open data.gdt
list L = dataset
bundle b = defbundle("n_train", 2000, "search", 1)
series yhat = svm(L, b)
```

4.1 Local search, linear search

After doing a coarse-grained parameter search you may wish to do an additional search in the neighborhood of some values that look promising. The \log_2 apparatus may not seem very convenient for that purpose, but a little thought reveals how it may be used. Listing 5 presents a function that takes as input a specific parameter value, `p0`, a specification of the lower limit of the search range expressed as a fraction (`frac`) of `p0`, and a number of steps. The return value is a suitable row for the grid matrix.

The sample usage shown in Listing 5 produces

```
r = {-0.32193, 0.32193, 0.16096}
```

which translates into the following values for the parameter itself: (0.8, 0.8944, 1, 1.1180, 1.25). This vector includes the `p0` value of 1.0, around which it is “exponentially symmetrical.”

Alternatively, you may specify that one or more rows of the grid matrix are to be taken as linear rather than \log_2 -based. In that case you must append a fourth column to the grid, with 1 on linear rows and 0 on \log_2 -based rows. Here’s an example:

```
params.svm_type = 4 # nu-SVR
params.grid = {0, 3, 1, 0; 0, 0, 0, 0; 0.5, 0.7, 0.05, 1}
```

This specifies an exponential search for C (from 1 to 8, doubling at each step), a clamped value for γ and a linear search for ν (0.5, 0.55, ..., 0.7).

4.2 Cross validation criteria

As mentioned above, cross validation results in the selection of a set of parameter values that are deemed “best” on some criterion. In the case of classification the optimality criterion is

Listing 5: Determining a local search specification

```
# helper function
function matrix make_grid_row (scalar p0, scalar frac, int nsteps)
  if p0 <= 0 || frac <= 0 || frac >= 1
    funcerr "Invalid input"
  endif
  if nsteps % 2 == 0
    nsteps++
  endif
  k = nsteps - 1
  start = log2(frac * p0)
  step = (log2(p0) - start)/(k/2)
  stop = start + k * step
  return {start, stop, step}
end function

# sample usage of the above
matrix r = make_grid_row(1, 0.8, 5)
```

always the proportion of correctly classified cases but in SVM regression a choice is offered, via the optional `regcrit` parameter. This must be an integer from 1 to 4, with 1 being the default, as follows:

- 1 Minimum MSE (Mean Squared Error)
- 2 Minimum MAD (Mean Absolute Deviation)
- 3 Minimum rounded MAD
- 4 Minimum rounded misses

By “rounded MAD” we mean the MAD calculated using the absolute difference between the actual value of the dependent variable and the model’s prediction rounded to the nearest integer. And by “rounded misses” we mean cases where the rounded prediction does not equal the actual dependent variable. Options 3 and 4 are applicable only when the dependent variable is integer-valued; an error is flagged if this condition is not met.

4.3 User-specified folds

In some contexts there may be reason to divide the data into subsets for cross validation on some chosen criterion rather than via random selection. The `svm` function supports two (mutually incompatible) options to this effect.

- If the parameter bundle contains the key `foldvar` with a non-zero value, `svm` will take the *last* series in the incoming list as a “folding index” rather than a regressor. Such a series must contain only integer values from 1 to the desired number of folds, which form a consecutive sequence when sorted. Each observation is then assigned to fold *i* if its value for the folding index is *i*.
- If the parameter bundle contains the key `consecutive` with a non-zero value this indicates that the folds should simply be blocks of consecutive observations, of equal size except that any remainder is assigned to the last fold. The number of folds defaults to 5 but can be adjusted via the `folds` key. This option may be useful if the order of observations in the dataset is already randomized.

4.4 More on random cross-validation folds

Unmodified `libsvm` calls the C-library function `rand` to produce its permutations—and it doesn't set a seed, so `rand` uses its default seed of 1 on each initialization. This means that each time a given `libsvm` program is executed (on a given platform) it will generate exactly the same set of “random” folds. However, it also means that if a given program calls the `libsvm` cross-validation function several times, as happens in the course of parameter search, it will get a different set of folds each time since automatic initialization of `rand` occurs only once per program invocation.

This behavior seems debatable on both counts. First, if the folds are advertised as random, one might expect to get non-identical results when running a given cross-validation program multiple times (unless one takes steps to ensure they are the same). Second, in the context of parameter search one might wish to suppress variation in outcome stemming from data-subsetting in order to focus on variation stemming from the differing parameter vectors. Moreover, the `rand` function is not guaranteed to produce the same sequence of values for a given seed on different operating systems.

Gretl's `svm` function addresses these points as follows. First, we use the SFMT implementation of the Mersenne Twister from the `gretl` library in place of `rand`; this ensures consistency across platforms. By default we obtain a seed based on the time at which `svm` execution starts, and we (re-)initialize SFMT with this seed immediately prior to each call to the `libsvm` cross-validation function. This means that each invocation of a `gretl svm` cross-validation program is likely to produce slightly different results, while in the context of a given search each parameter combination will employ the same set of folds. This default behavior can be modified in two ways.

- To obtain strictly reproducible results you can override the execution-time based SFMT seed with an integer `seed` value specified in the parameter bundle passed to `svm`.
- To emulate `libsvm`'s behavior, whereby each call to randomized cross validation generates a new set of folds, specify a non-zero value under the key `refold` in the parameter bundle.

When random folding with the default time-based seed is employed, it may be useful to know what seed was actually used: you can retrieve this value from the parameter bundle under the key `autoseed`. (This item is added to the bundle only if random folding is performed and no seed value is specified on input.)

One further note: when pseudo-random numbers are required in the context of `svm` they are drawn from an independent instance of SFMT, distinct from the one governed by the `set seed` command and employed by functions such as `normal()` and `randgen()`.

4.5 More on SV regression

We noted above that `libsvm` supports two types of SV regression: ϵ -SVR and ν -SVR. These are alternative parameterizations of the regression problem. To understand this point one must know what ϵ and ν represent.

- $\epsilon \geq 0$ sets a margin within which prediction errors are ignored (costed at zero). To enforce a tight fit—at the risk of over-fitting—therefore, one would specify a small value of ϵ .
- $0 < \nu \leq 1$ controls the number of support vectors used by the SVM, expressed as a fraction of the maximum (which equals the number of observations in the training data). Use of more support vectors will generally give a better fit—again at the risk of over-fitting.

These values cannot be set independently of each other. In ϵ -SVR ϵ is parametric and ν is adjusted endogenously; the converse holds for ν -SVR.

The literature on this topic suggests that if you want a quick fit at moderate computational cost you might choose ν -SVR with a relatively low value of ν . On the other hand, if you want to obtain the best possible prediction and are not too concerned about computational cost it would be common to use ϵ -SVR.

One point to note here is that ϵ is not scale-free: the implied “tightness” of setting, say, $\epsilon = 0.05$ will depend on the scale of the regressand. Experimentation may be useful.

4.6 Use of MPI in cross validation

As mentioned above, cross validation can be very expensive computationally if you have a big training dataset. Fortunately, it’s also “embarrassingly parallel” (as the computer scientists say); that is, trivial to parallelize. The (potentially quite large) set of parameter combinations can be parceled out to separate processes, since each set of training runs (across the folds, for a given combination of parameters) is independent of the others. This is an obvious use-case for MPI.

MPI is “Message Passing Interface,” a standard for running several distinct processes that execute the same program using different data. Support for MPI in gretl is documented in [Cottrell and Lucchetti \(2017\)](#). Usually we leave it up to users of gretl to exploit MPI via scripting (which demands a certain proficiency in coding), but in the case of cross validation under `svm` we have implemented MPI support internally. All the user has to do is execute a call to `svm` that calls for cross validation, either via the command-line program `gretlmpi` or within an `mpi` block in a script that’s run in the gretl GUI. For details of these options, please see [Cottrell and Lucchetti \(2017\)](#).

To make a serious dent in the time taken by a big cross validation problem using MPI one would want to exploit a high-performance cluster if possible. However, to illustrate the potential of MPI, even just on a single machine, we constructed a task that is big enough to produce meaningful differences in execution times but not so big as to waste a lot of time. We used one of the libsvm sample regression datasets,⁸ `cadata`, a native gretl version of which is available as

<http://gretl.sourceforge.net/svm/cadata.gdtb>

Like the Mullainathan-Spiess dataset this concerns housing values, but it is smaller both in number of observations (20640) and number of covariates (8). Listing 6 shows a script which performs cross-validation using 3000 observations, with a grid for C and γ which gives 32 parameter combinations. The times taken to produce the results matrix under different configurations on a Dell desktop with 4 Intel Haswell cores and up to 8 hyperthreads are shown in Table 3. In this example it turns out that multi-threading via OpenMP is not greatly advantageous, but MPI works quite nicely—for up to six MPI processes.

For reference, the command line for the fastest run represented in Table 3 (MPI 6, OMP 1) was

```
OMP_NUM_THREADS=1 mpirun -N 6 gretlmpi ca_xvalid.inp
```

The optimized result (the same on all runs) was a cross-validation MSE of 0.0581 at $C = 2$ and $\gamma = 2$.

4.7 Mullainathan and Spiess revisited

Having discussed cross validation we return to the replication of [Mullainathan and Spiess \(2017a\)](#) discussed in section 2.1. How much can we improve on the results of ϵ -SVR “out of the box” by means of parameter search?

⁸See <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>.

Listing 6: Parameter search, script `ca_xvalid.inp`

http://gretl.sourceforge.net/svm/ca_xvalid.inp

```
set verbose off
open cadata.gdtb -q
list X = dataset - median_value
series lny = log(median_value)
list L = lny X
bundle b = defbundle("n_train", 3000, "shrinking", 0, "search_only", 1)
b.grid = {-1, 5, 2; 3, -12, -2}
b.seed = 1611771
bundle res = defbundle()
svm(L, b, &res)
if $mpirank < 1 # just print once
  matrix R = res.xvalid_results
  print R
endif
```

<i>configuration</i>	<i>wall time</i>	<i>%CPU</i>
single-threaded	1:17	99
OMP 4	1:00	274
OMP 8	1:04	515
MPI 4, OMP 2	0:27	535
MPI 4, OMP 1	0:27	357
MPI 6, OMP 1	0:23	478

Table 3: Execution times (m:s) for `ca_xvalid.inp` (OMP = number of OpenMP threads, MPI = number of MPI processes)

We began with some “sighting shots”—small, coarse-grained cross-validation runs, performed on a standard desktop machine and using the `libsvm` default of 5 random folds—which gave us the idea that C values between 2.5 and 3.0 could be helpful (as against the default of 1.0). It was also apparent that moderate variations in γ and ϵ were worth exploring.

On that basis we set up a finer (linear) grid with 80 parameter combinations and used `gretlmpi` on an HPC node with 32 Xeon cores, running 16 MPI processes with two threads each. In addition, rather than random folds we used the `folds8` variable in the M-S dataset, which specifies 8 random subsets of the training sample based on clusters in the data.⁹ Cross-validation took almost an hour and yielded minimum MSE at 0.606208 for $C = 2.8$, $\gamma = 0.00365$, $\epsilon = 0.09$. The selected C was in the range we expected and ϵ was somewhat smaller than its default of 0.1; the selected γ was equal to the default (the reciprocal of the number of covariates).

We then trained a model on all 10,000 training observations using the selected parameters and produced predictions for all the data, giving

```
SVM: training R^2 = 0.532, MSE = 0.522
SVM: holdout  R^2 = 0.454, MSE = 0.633
```

as opposed to the results shown earlier

```
SVM: training R^2 = 0.494, MSE = 0.565
SVM: holdout  R^2 = 0.448, MSE = 0.639
```

The new “holdout” performance is practically indistinguishable from that obtained by M-S using the “Random forest” method; it still falls a little short of the “Ensemble” method but we’re talking about the third digit of R^2 . Perhaps we could do a little better with a fuller exploration of the parameter space but these results seem quite satisfactory.

4.8 Parameter tuning plot

It may be helpful to get a visual fix on what’s going on in the tuning of the RBF kernel parameters via cross validation. There’s an experimental (and so far, little tested) option for this: you can include in the parameter bundle under the `plot` key either “display” or an output filename, as in

```
parms.plot = "display"
# or
parms.plot = "tuning.pdf"
```

This has an effect only if a two-dimensional (C, γ) parameter search is performed. If `svm` catches on we have in mind to generalize this and make it more convenient.

5 Classification via SVM

The focus of this document has been on regression, but `gretl`’s `svm` function also supports classification. As mentioned above, the `svm` function automatically switches to classification mode—by default using C -SVC—if the dependent variable is marked as “coded” using the `setinfo` command. However, the `svm_type` switch in the parameter bundle passed to `svm` can be used to force the issue in favor of classification via C -SVC (`svm_type = 0`) or via ν -SVC (`svm_type = 1`).

⁹See [Mullainathan and Spiess \(2017b\)](#) for details.

5.1 Classification example

Listing 7 shows an example of classification based on the script `keane.inp`, which is supplied with the `gretl` distribution. The data file is `keane.gdt` (also supplied), a subset of the data used in Keane and Wolpin (1997). The original script exemplifies multinomial logit regression. The dependent variable, `status`, encodes the jointly exhaustive states “in school” (1), “at home” (2) and “in work” (3); the explanatory variables are years of education (`educ`), years of work experience (`exper`) and its square (`expersq`), and a `black` dummy variable.

Listing 7 plays C-SVC against standard Maximum Likelihood estimation. With only 4 covariates and 3 classes `svm` runs quite quickly so the script is integrated: parameter search using the default grid is followed by training and prediction. The dataset comprises several years of data; we estimate the respective models on the data from 1982 and test using the 1984 data. The results (obtained in about half a minute on the desktop machine described earlier) are:

```
MLE: training percent correct = 61.16 (n=1864)
MLE: testing percent correct  = 69.76 (n=1802)
SVM: training percent correct = 61.70 (n=1864)
SVM: testing percent correct  = 70.20 (n=1802)
```

So we see a slim predictive advantage for C-SVC over multinomial logit estimation. Given the paucity of the explanatory data it is perhaps surprising that the SVM even equals the MLE results.

6 Probability estimation

By default the output of `svm` is just a series of point-predictions of the dependent variable conditional on the covariates or “features”. It is possible, however, to obtain a measure of the uncertainty attaching to these predictions. The character of the probability information delivered by `libsvm` differs between SV regression and SV classification, but the mechanism for obtaining it via `gretl` is basically the same: you set `probability` to 1 in the parameter bundle, and supply a pointer-to-bundle as the fourth argument to the `svm` function, as in

```
...
params.probability = 1
bundle bprob = defbundle()
series yhat = svm(L, params, null, &bprob)
```

(where the `null` argument indicates that we’re skipping the optional third parameter). On successful completion `bprob` will contain probability information.

6.1 Regression error distribution

In the case of SV regression, the extra information takes the form of a scalar value under the key `svr_sigma`: this is an estimate of the scale parameter, σ , for the Laplace density of the prediction errors, $\hat{y}_i - y_i$, based on the training data, namely

$$\frac{1}{2\sigma} \exp\left(-\frac{|\hat{y}_i - y_i|}{\sigma}\right)$$

Using this information one can, for example, calculate confidence intervals for the SVR predictions. The following code snippet shows how one could obtain a 90 percent interval, using the `invcdf` (inverse CDF) function with first argument `L` for the Laplace distribution:

```
sigma = brob.svr_sigma
maxerr90 = invcdf(L, 0, sigma, 0.95)
series lo = yhat - maxerr90
series hi = yhat + maxerr90
print yhat lo hi --byobs
```


Listing 7: Parameter search and estimation: C-SVC vs multinomial logit

<http://gretl.sourceforge.net/svm/multinomial.inp>

```
set verbose off

# helper function
function scalar mnlogit_pc_correct (const series y,
                                   const matrix X,
                                   matrix b)

    matrix yvals = values(y)
    scalar n = rows(X)
    b = mshape(b, cols(X), nelem(yvals) - 1)
    matrix P = ones(n, 1) ~ exp(X * b)
    return (100/n) * sumc({y} . = yvals[imaxr(P)])
end function

# open and organize data
open keane.gdt -q
dataset sortby year
setinfo status --coded
ytrain = 82 # train on data from 1982
ytest = 84  # test using data from 1984
list All = status educ exper expersq black

# (1) MLE: multinomial logit
# restrict to complete observations in training year
smpl year == ytrain && ok(All) --restrict
logit status 0 educ exper expersq black --multinomial --quiet
pcc = 100 * sum(status == $yhat) / $T
printf "MLE: training percent correct = %.2f (n=%d)\n", pcc, $nobs
# restrict to complete observations in testing year
smpl year == ytest && ok(All) --restrict --replace
matrix X = {const, educ, exper, expersq, black}
pcc = mnlogit_pc_correct(status, X, $coeff)
printf "MLE: testing percent correct = %.2f (n=%d)\n", pcc, $nobs

# (2) SVM: C-SVC
# sample: all complete observations in the two selected years
smpl (year==ytrain || year==ytest) && ok(All) --restrict --replace
scalar ntrain = sum(year == ytrain)
# start with cross validation search using default grid
bundle parms = defbundle("n_train", ntrain, "search", 1)
parms.seed = 3211765 # for reproducibility
parms.quiet = 1
yhat_svm = svm(All, parms)
# assess fit in training year
smpl year == ytrain && ok(All) --restrict --replace
pcc = 100 * sum(status == yhat_svm) / $nobs
printf "SVM: training percent correct = %.2f (n=%d)\n", pcc, $nobs
# assess fit in testing year
smpl year == ytest && ok(All) --restrict --replace
pcc = 100 * sum(status == yhat_svm) / $nobs
printf "SVM: testing percent correct = %.2f (n=%d)\n", pcc, $nobs
```

6.2 Discrete outcome probabilities

In the classification case, the information supplied via the bundle passed as the fourth argument to `svm` takes the form of the matrices `Ptrain` (if prediction over the training range is requested) and/or `Ptest` (if prediction over the testing range is called for). Each of these matrices has as many columns as there are outcomes, and as many rows as there are observations in the relevant range. The column headings identify the outcome values.

In relation to the example in Listing 7 above, the following modifications could be made to generate and inspect probability estimates. First, after the line “`parms.quiet = 1`” (9 lines from the foot of the script) and replacing the line “`yhat_svm = svm(All, parms)`”, insert:

```
parms.probability = 1
bundle bprob = defbundle()
yhat_svm = svm(All, parms, null, &bprob)
```

Then, to inspect the probabilities for the testing data, one could append to the script:

```
matrix P = bprob.Ptest
print P
```

We show below the first few lines of output from this variant of the script.

1	2	3
0.17717	0.60506	0.21777
0.063540	0.23416	0.70230
0.098154	0.64375	0.25810
0.094531	0.36033	0.54514

The columns are ordered by ascending value of numerical coding of the dependent variable. So, for example, we see that the model estimates a probability of 0.605 for an outcome of 2 (“at home”) for the first observation in the training set, and a probability of 0.702 for outcome 3 (“in work”) for the second observation.

7 Implementation

SVM support is implemented via a gretl “plugin” module. This module is supplied in our packages for MS Windows and macOS; it should be available in the gretl packages prepared by Linux distributions for gretl 2019a and higher.

For anyone building gretl from the git sources, all the required files are included. (That is, it is not necessary to have `libsvm` installed in its own right.) The main files are these:

```
plugin/svm.c
plugin/libsvm/svmlib.cpp
plugin/libsvm/svmlib.h
```

`svm.c` contains “driver” code and implements the gretl-specific options for our `svm` function. The other two files are derived from the `libsvm` package (version 3.23, dated 2018-07-15).

`svmlib.cpp` is a slightly modified version of the `libsvm` C++ source file `svm.cpp`. It has been edited to support parallelization via `OpenMP` when the symbol `_OPENMP` is defined. The changes are as described in answer to “How can I use `OpenMP` to parallelize `LIBSVM` on a multicore/shared-memory computer?” in the file `FAQ.html` in the `libsvm` distribution. In addition, the pseudo-random numbers used in cross validation are taken from `libgretl`’s `SFMT` (Mersenne Twister) instead of the C library’s `rand`.

`svmlib.h` is just a renamed version of the `libsvm` header file `svm.h`.

References

- Cottrell, A. and R. Lucchetti (2017) 'Gretl + MPI'. URL <http://sourceforge.net/projects/gretl/files/manual/gretl-mpi.pdf>.
- Hsu, C.-W., C.-C. Chang and C.-J. Lin (2016) 'A practical guide to support vector classification'. Department of Computer Science, National Taiwan University. URL <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf>.
- Keane, M. P. and K. I. Wolpin (1997) 'The career decisions of young men', *Journal of Political Economy* 105: 473–522.
- Mullainathan, S. and J. Spiess (2017a) 'Machine learning: An applied econometric approach', *Journal of Economic Perspectives* 31(2): 87–106. URL <https://doi.org/10.1257/jep.31.2.87>.
- _____ (2017b) 'Machine learning: An applied econometric approach online appendix'. URL <https://www.aeaweb.org/articles?id=10.1257/jep.31.2.87>.
- Smola, A. J. and B. Schölkopf (2004) 'A tutorial on support vector regression', *Statistics and Computing* 14: 199–222. URL <https://alex.smola.org/papers/2004/SmoSch04.pdf>.