# SVMsequal Tutorial Manual

Hal Daumé III

March 31, 2004

# Contents

# Chapter 1

# Introduction

SVMsequel[1] is a complete environment for training and use support vector machines. In this tutorial, I will assume you have a general idea of basic machine learning concepts like *training* or *estimation*, *prediction* or *classification*, and *cross-validation*. Some familiarity with kernel methods will be helpful (see `http://www.isi.edu/~hdaume/docs/` for class notes I've used for using SVMs for natural language processing if you need a refresher.

## Training Algorithm

The underlying training algorithm used in SVMsequel is based on the kernel minover algorithm [**?**], which gives comparable performance to the more popular sequential minimal optimization (SMO) algorithms employed by most other SVM systems. However, unlike the SMO algorithm [**?**] the kernel minover algorithm (KM) does require certain extraneous parameters to be set (learning rate and number of iterations); SVMsequel will attempt to set these to a reasonable value, though you can change them yourself.

## Shell Environment

SVMsequel differs from every other SVM package I've ever used in that instead of providing several different executables (eg., one for training, one for prediction, perhaps one for normalization or cross-validation, etc.), it provides a *shell* for executing a set of predefined commands. In that sense, it resembles a system like Matlab more than another SVM learning system. However, due to the fact that these scripts can take parameters, it is possible to implement the functionality of the other learning programs through scripts of a few lines of SVMsequel shell code. Some simple ones are included in the distribution to get you going (these will be described in more detail later).

[1]The etimology of SVMsequel is that it is the successor to the SVMseq software I released about a year ago. The name SVMseq is no longer appropriate because the learning algorithm has changed, but I wanted to maintain some name-brand consistency. Thus, "SVMsequel" was born.

## Kernels Offered

Additionally, SVMsequel provides more kernels than most available software packages. Most typically provide linear, polynomial, RBF and sigmoidal kernels. We provide these, as well as information diffusion kernels, (limited) diffusion kernels on graphs (for discrete valued features), string kernels (based either on dynamic programming solutions or solutions based on suffix trees) and tree kernels. In addition, if none of these is sufficient, you can also provide our algorithm with precomputed kernel matrices and we will be able to solve the SVM problem using that matrix. Furthermore, SVMsequel gives you the ability to *combine* kernels in a weighted fashion, enabling you to use discrete kernels together with real-valued kernels.

## Model Types

SVMsequel can directly learn binary classifier, multiclass classifiers using one-versus-all, one-versus-none or all-versus-all classification. It can also learn probabilistic classifiers, using [**?**]. And all in one program!

## Efficiency

Finally, SVMsequel is fast. It is *much* faster that its cousin SVMseq, and can be faster than, eg., SVM light or LIBSVM (though these are excellent programs). The KM algorithm is an interative algorithm (for most problems, several hundred to a thousand interations suffice) and each iteration is *linear* in the number of examples[2] (disregarding the complexity of computing the kernel). This gives an incredibly efficient algorithm for training these highly non-linear machines.

## Conditions of Use

SVMsequel can be used for any non-commercial purpose[3]. I request two things if you do use it: (1) please let me know what you are using it for (I love to hear that people find my software useful) and (2) please give appropriate credit.

[2]In the original paper [**?**], the algorithm is not designed to be run in linear time, though it is straighforward to modify it slightly so that this is possible, without giving up on optimality.

[3]If you wish to use it for a commercial purpose, please contact me by email at `hdaume@isi.edu`.

# Chapter 2

# Architecture and Data Types

There are four main "data types" in SVMsequel. The most common is a *dataset*, which is a collection of datapoints, perhaps with classes. Datsets are broken down into *clumps*, which are, in general, used to distinguish different kinds of data (eg., distinguishing string data from real-valued data). In addition, there are *models*, which are build from data and finally *variables*, which are string values used for scripting.

## 2.1 Datasets and Clumps

In SVMsequel, a learning problem is defined by a dataset. A dataset is a collection of datapoints, perhaps with associated classes (if you want to do model estimation, you will need the associated classes; if you only with to predict, you will not need them). Since SVMsequel can handle *heterogeneous* data (for instance, data consisting of strings as well as real-valued data), each dataset is split up into a set of *clumps* (perhaps there's better terminology, but this is what we're sticking to). This is perhaps best served with an example.

Suppose we're doing document classification. Using string kernels, we are able to have a very large (or infinite) set of features corresponding to string subsequences. However, we might also want to include more traditional document classification features based on td-idf or some other real-valued score. In this case, we would have two clumps. One would be the clump for which the kernel value is computed using string kernels; the other would be the clump for which the kernel value is computed using linear or RBF kernels.

There are three steps in creating a dataset and preparing it for estimation:

1. Create the dataset and give it a name.
2. Give the dataset as many clumps as required and supply them with their necessary paramters.
3. Load data from (a) file(s) into the clumps (and possibly give classifications to the datapoints).

We will discuss these in more depth in the next chapter.

## 2.2 Models

Once you have build a dataset (complete with clumps), you can estimate a model for it. This can either be done in one step if you set paramters yourself, or it can be done using cross-validation if you don't know the optimal paramters. Once you have a model, you can use it to predict classes on (unseen) data. You can also train probabilistic parameters for it. SVMsequel will enable you to do any or all of these things.

## 2.3 Variables

For scripting purposes, it is sometimes useful to have variables. There are two basic types of variables. You can have named variables, with string names, or argument variables, which are given numbers based on their ordering (this will perhaps become more clear when we describe scripting). Variables are always reffered to by a dollar character, like in most shells. For instance, `$kernel` is a named variable and `$2` is an argument variable corresponding to the second (unnamed) argument to a function.

# Chapter 3

# Building Datasets

As previously described, there are three steps in building a dataset:

1. Create the dataset and give it a name.
2. Give the dataset as many clumps as required and supply them with their necessary paramters.
3. Load data from (a) file(s) into the clumps (and possibly give classifications to the datapoints).

We will describe each of these in turn.

However, in addition to just creating a dataset, there are also many things you can do with it, including changing parameters, saving it to disk (and loading it back), and normalizing it (either in its own right or with respect to some other dataset or model). We will also discuss these. Finally, we will conclude with some examples

## 3.1 Creating a Dataset

Creating a dataset it easy. You choose a name for it (for example, `train` and then execute:

```
> new data train
Data train added
```

We can verify that this completed successfully by running the `who` command, which will tell us what active values exist:

```
> who
Data sets:
        train   (0 clumps, 0 points)
```

```
Models:

Variables:

```

As we can see, there is one data set with zero clumps and zero data points.

We can get more information about this data (in particular, what the clumps are) by executing `data info` on it. In this case, this provides no more information, though, since there are no clumps:

```
> data info train
train has 0 clumps and 0 points:
```

## 3.2 Adding Clumps

Now that we've created a dataset (called `train`), we need to add clumps to it. Let's suppose our data is as below, in SVMseq format:

```
+1 string:1:svmsequel#is#a#great#program 2:0.5 3:1 8:10
+1 string:1:so#is#svmlight 3:5.3 6:5
-1 string:1:svmsequel#is#a#terrible#program 3:2.0 4:-3.3 8:0.3
-1 string:1:neither#is#svmlight 8:0.2 9:5.3
```

This data is included in the file `simple.svm` in the `examples` directory of the SVMsequel distribution.

In this data, we wish to create two clumps. The first clump will correspond to the string feature (feature 1) and the second clump will correspond to the rest of the features (in this case, 2 through 9). We will use the dynamic programming string kernel for the strings and an RBF kernel over sparse vectors for the real-valued data. We add these clumps by saying:

```
> add dpstring clump to train
Clump #0 added to train
Parameters of clump #0 of train set
> add sparse clump to train kernel rbf
Clump #1 added to train
Parameters of clump #1 of train set
```

The format of the `add` command is:

```
add <clump-type> clump to <dataset> <parameters>
```

The valid options for `<clump-type>` and the associated paramters are listed below:

| `<clump-type>` | **Description** |
|---|---|
| dense | This is for dense vectors of real-valued data. In this data type, all of the values in the vector are stored (including zeros). For dense vectors, this is more efficient (in terms of computing dot products) than sparse vectors, but can take more memory.<br>The relevant `<parameters>` for the dense clumps are the standard kernels and their arguments. You specify the kernel type, as above, by saying `kernel <kernel-type>`. The supported kernel types are `linear`, `poly`, `rbf`, `sig` and `id`.<br>• The linear kernel takes no parameters.<br>• The `poly`nomial kernel takes parameters `constant <a>`, `coefficient <b>` and `degree <d>`. If unspecified, these default to 0, 1 and 2, respectively. The kernel function is then $K(u, v) = (a + bu \cdot v)^d$.<br>• The `rbf` (radial basis function) kernel takes one parameter, `gamma`, which defaults to 1 if unspecified. The kernel has value $K(u, v) = exp(-\gamma\|u - v\|^2)$.<br>• The `sig`moidal kernel takes two parameters, `constant <a>` and `coefficient <b>` and then has value $K(u, v) = tanh(a + bu \cdot v)$.<br>• The `id` (information diffusion) kernel takes one parameter, `beta <b>`. |
| sparse | Sparse vectors are also for real-valued data, but are less memory intensive when many of the components are zero. They take the same kernel parameters as dense vectors. |
| dpstring | This is the dynamic-programming based string kernel. It is for string data (string data is specified with spaces replaced by hash/pound signs). It takes two paramters, `lambda <l>` and `maxn <n>`. Given these paramters, it only computes subsequences up to length `n` and down-weights spread-out subsequences by `l` raised to the length. `l` should be between 0 and 1. |

| `<clump-type>` | **Description** |
|---|---|
| ststring | Same as dpstring, but uses suffix-trees to compute the string kernel. This can be much faster for long strings. It only takes the lambda parameter above. |
| discrete | This is for discrete valued data and can only hold one data point per clump. The kernel is based on the diffusion kernel on graphs and takes two parameters, `cardinality <A>` and `lambda <l>`. The cardinality parameter specifies how many possible values this feature can take and the lambda value is the weight value used for the kernel. |
| matrix | This is used if you want to compute kernel values yourself. This takes no parameters. You will need to load the actual matrix later (see later section). |

In addition to the parameters specific to the clump type, each clump can take a weight value (say `weight <w>`), which specifies how important it is, relative to other clumps. In general, you won't need to specify this; the default of 1 will work fine.

Now that we've specified our clumps, we can run `data info` and make sure everything is okay:

```
> data info train
train has 2 clumps and 0 points:
  0: dpstring, weight=1, lambda=0.75, maxn=4
  1: sparse  , weight=1, kernel=rbf gamma=1., highest=-1
```

Everything listed here looks fine. The `highest=-1` note on clump #1 says that the highest feature number used in this clump is $-1$, indicating that no data has yet been loaded.

## 3.3 Loading Data into Clumps

Now that we've configured that data set, we need to load data into it. We do this with the `load` command. In this case, we say:

```
> load svm file simple.svm to train clump 0 from 1
      clump 1 from 2-9
Loading data from simple.svm into 'train'
Data loaded from simple.svm into 'train'
> data info train
train has 2 clumps and 4 points:
  0: dpstring, weight=1, lambda=0.75, maxn=4
  1: sparse  , weight=1, kernel=rbf gamma=1., highest=7
```

The general format of the `load` command is

```
load <ftype> file <path> to <dataset> <columninfo>
```

Where `<ftype>` is either "svm," for SVM-style files, "csv" for comma-separated files or "ssv" for space-separated files. The value `<path>` is the path of the file to load, `<dataset>` is the destination dataset and `<columninfo>` specifies how features in the file correspond to clumps in the dataset.

In particular, `<columninfo>` is a list of arguments of the form:

```
clump <n> from <f>
```

where `<n>` is a valid clump number in `<dataset>`, and `<f>` has the form of `f` or `f-g` where `f` and `g` are feature numbers (or column numbers in csv or ssv files), or `x,y` where `x` and `y` are respectively column numbers or ranges. For instance, all of the following are valid: `3` and `3,8` and `3,8-12` and `1-2,4-9,10,15,20-30`. Additionally, if you simply want to load all columns into a single clump, you can say `clump <n> from all`.

You can optionally say `noclasses` if you wish to ignore any class information in the file, or `rank` if this is ranked data (*not yet implemented*). Furthermore, if loading from csv or ssv data, and you wish to load the class information from a particular column, you must say `classes <n>` where `<n>` is the column that contains the class information.

## 3.4 Normalizing

It is often useful to normalize data to fall in the range $[-1,1]$. We can do this using the `normalize` command. Note that when normalizing, you should normalize the training data by itself, and then normalize any testing data using the same normalization routines as used for normalizing the training data. In order to facilitate this, normalization information is stored in datasets and models. To normalize your training set you say:

```
> normalize data train
Data 'train' normalized
```

Later, to normalize testing data, you would say either

```
normalize data test by data train
```

or

```
normalize data test by model m
```

where `m` is a model built from normalized data.

## 3.5 Saving and Loading Data

Once you have adequetly prepared your data, you can save it to a file by:

```
> save data train to train.dat
Data set train saved to simple.dat
```

We can now quit, rerun, and load the data back:

```
> quit

% ./svmsequel
...

> load data train from train.dat
Data set train loaded from to simple.dat
> data info train
train has 2 clumps and 4 points:
  0: dpstring, weight=1, lambda=0.75, maxn=4
  1: sparse  , weight=1, kernel=rbf gamma=1., highest=7
```

# Chapter 4

# Training Models

You have two choices for estimating the paramters of a model. You can either use cross-validation to estimate parameters (such as C, gamma values, etc.), or you can just use the ones you've set. The latter is easier, so we'll do it first.

## 4.1 Training with Known Parameters

This is very simple. Assuming the dataset you wish to train with is in the variable `train`, all we need to say is:

```
> estimate model m from train
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
Model m estimated from train
```

In general, when estimation is taking place, the characters 'a-z' and the 'A-Z' will be output, letting you know how far along we are. In addition, you can set any of the values `c`, `numi` (number of iterations), `epsilon` (stop when margin is at most `epsilon` from 1) and `eta`, the learning rate. These default to 1, 1000, 0.0001 and 0.1, respectively if not specified. For instance, run:

```
> estimate model m from train numi 100000
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
Model m estimated from train
```

And (assuming you're computer is not too much faster than mine), you will see that the letters are printed out a bit more slowly.

We can get the model info by saying:

```
> model info m
m has 2 clumps:
  0: dpstring, weight=1, lambda=0.75, maxn=4
  1: sparse  , weight=1, kernel=rbf gamma=1., highest=7
4 support vectors, binary classifier
```

## 4.2 Cross-Validating for Parameters

Suppose we don't know, or don't want to guess at the parameters. Then we can use cross-validation to estimate them from the data. First, we need to prepare the data for cross-validation by saying:

```
> cross-validate train 10 folds
Cross validation for 10 folds initialized for 'train'
```

You can, of course, use any number of folds you wish (ten is fairly common). Since we only have four data points, it doesn't make much sense to do more than four, but we'll do ten just to illustrate.

Now, we can actually do the cross-validation. Suppose we wish to estimate $C$ and $\gamma$ from clump 1. We believe reasonable values of $C$ are $2^{-3}, 2^{-2}, \dots, 0, \dots, 2^2, 2^3$, and similarly for $\gamma$ (for normalized data, these are reasonable assumptions). Given this, we say:

```
> optimize model m from train c:log2:-3:1:3
           1:gamma:log2:-3:1:3
Optimizing:
  'c'
  'gamma' from clump 1

Currently optimizing c
Fold 0: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP...
...
        c = 1 => error rate of 2
...
        c = 0.5 => error rate of 1
...
        c = 2 => error rate of 2
...
        c = 0.25 => error rate of 4
Optimal value at 0.5 (err=1)


Currently optimizing gamma from clump 1
...
        gamma = 1 => error rate of 3
...
        gamma = 0.5 => error rate of 3
```

```
...
        gamma = 2 => error rate of 4
Optimal value at 1 (err=3)

All parameters optimized (accuracy = 3/4 = 75%)
  'c'                    = 0.5
  'gamma' from clump 1  = 1

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
Model m estimated from train
```

What is happening here is that first it optimizes $C$, beginning with the middle three values of the range you've specified. Based on this, it chooses a direction to go (either up or down) and then keeps going that direction until there's no improvement. In this case, it finds a value of $C = 0.5$. Then, it moves on to optimize $\gamma$ in the same way. In the end, it reports the optimal parameters, and estimates the model (the last line of "abc...XYZ") based on these parameters. Here, the accuracy of $75\%$ is bad, but only because this data set is way too small to be doing this much cross-validation on.

The general format for the `optimize` command is:

`optimize model <m> from <dataset> <args> <params>`

Here, `<args>` is the same as in `estimate` (namely, `c`, `eta`, `epsilon` and `numi`). The `<params>` specify what parameters to optimize. In general, each of these takes the form:

`<clump>:<param-name>(:log2)?:<min>:<step>:<max>`

where `:log2` is optional. There is one special case: when you are optimizing $C$, obviously this doesn't belong to a clump so you just say:

`c(:log2)?:<min>:<step>:<max>`

Not surprisingly, this will search for the given parameters in the given clump between `min` and `max` with steps of size `step`. In the case that `log2` is specified, it will search between $2^{<min>}$ and $2^{<max>}$ with steps of size $2^{<step>}$, which we have used above, since both $C$ and $\gamma$ are strictly positive values.

## 4.3 Probabilistic Classification

For binary classifiers (and soon in a future version for multi-class OVA classifiers), you can train probabilistic models:

```
> estimate sigmoid for m from train
Training sigmoid: ab, done!
> model info m
m has 2 clumps:
  0: dpstring, weight=1, lambda=0.75, maxn=4
  1: sparse  , weight=1, kernel=rbf gamma=1., highest=7
4 support vectors, binary classifier , probabilistic
```

Here, you specify both the model and the training data to estimate the parameters from. The “ab” could go up to “XYZ,” but the estimation took less time than required.

## 4.4 Saving and Loading Models

We can save and load the model by:

```
> model info m
m has 2 clumps:
  0: dpstring, weight=1, lambda=0.75, maxn=4
  1: sparse  , weight=1, kernel=rbf gamma=1., highest=7
4 support vectors, binary classifier
> save model m to simple.model
Model m saved to simple.model
> quit

% ./svmsequel
...

> load model m from simple.model
Model m loaded from to simple.model
> model info m
m has 2 clumps:
  0: dpstring, weight=1, lambda=0.75, maxn=4
  1: sparse  , weight=1, kernel=rbf gamma=1., highest=7
4 support vectors, binary classifier
```

# Chapter 5

# Prediction

Once you have a model and dataset (preferably one you didn't train on), you can predict output, very easily by:

```
> predict train with m
Accuracy = 4/4 = 100%
```

You can also get the outputs for each point either to stdout or to a file by saying:

```
predict <dataset> with <model> output <filename>
```

where the outputs will be written to `<filename>`, or to stdout if `<filename>` is −:

```
> predict train with m output -
1 1.12808 p=0.769962
1 0.925791 p=0.727604
-1 -0.900615 p=0.258394
-1 -0.969052 p=0.24404
Accuracy = 4/4 = 100%
```

The first column is the predicted class and the second is the actual value of the decision function. Since we have a probabilistic classifier, it also reports probabilities for each point belongs to class 1, which coincide with the function outputs.

# Chapter 6

# Multi-class Classification

All that has preceeded applies to multi-class classification, too. The only difference is that instead of having $\{-1, +1\}$ for classes, you must have some subset of $\{0, 1, 2, \ldots\}$ for classes. Additionally, you must specify when estimating whether you want to use one-versus-all (OVA) or all-versus-all (AVA) classification.

Consider the following data:

```
1 1:1    2:1
1 1:0.9  2:1.1
2 1:-2.1 2:2
2 1:-2   2:1.9
3 1:3.1  2:-3.2
3 1:3.1  2:-3
4 1:-3.9 2:-4.1
4 1:-4.1 2:-4
```

This data has four classes (note that we didn't start at zero – it doesn't matter) and is stored in `mc.svm`. We will describe both OVA and AVA models of this data below.

## 6.1 One-Versus-All

In OVA classification on $n$ classes, $n$ classifiers are trained, each classifier $f_i$ separates class $i$ from all classes $j \neq i$. We can load this data and train an OVA model as follow (we elect to not do cross-validation; if we used `optimize` instead of `estimate`, there would be no difference):

```
> new data train
Data train added
> add dense clump to train
Clump #0 added to train
Parameters of clump #0 of train set
> load svm file mc.svm to train clump 0 from all
```

```
Loading data from mc.svm into 'train'
Data loaded from mc.svm into 'train'
> normalize data train
Data 'train' normalized
> estimate model m from train ova
  class 1: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 2: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 3: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 4: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
Model m estimated from train
> model info m
m has 1 clumps:
  0: dense   , weight=1, kernel=linear
8 support vectors, one-versus-all (4) classifier
```

We can see that a separate optimization was done for each class and that in the end, there are 8 support vectors (all of them).

We can use this model to predict, just as before:

```
> predict train with m output -
1 1:0.187218 2:0.0414253 3:0.0260298 4:-0.529744
1 1:0.186938 2:0.063326 3:-0.0136598 4:-0.528681
2 1:-0.0150976 2:0.446893 3:-0.814939 4:0.0482647
2 1:-0.0148177 2:0.424993 3:-0.775249 4:0.0472024
3 1:-0.000545996 2:-0.691944 3:1.24892 4:-0.00697583
3 1:0.0178959 2:-0.6659 3:1.21183 4:-0.0588879
4 1:-0.748591 2:-0.187602 3:-0.0644298 4:2.11791
4 1:-0.758371 2:-0.156822 3:-0.125266 4:2.14599
Accuracy = 8/8 = 100%
```

In the prediction, the chosen class is listed first, followed by the function outputs for each classifier. The class is chosen as the classifier with maximal value (unless probabilistically trained, which is not yet implemented).

## 6.2 All-Versus-All

In AVA classification with $n$ classes, $\binom{n}{2}$ classifiers are trained, $f_{i,j}$ for separating class $i$ from class $j$. The prediction is chosen by a voting scheme. This is idential to before:

```
> new data train
Data train added
> add dense clump to train
Clump #0 added to train
Parameters of clump #0 of train set
```

```
> load svm file mc.svm to train clump 0 from all
Loading data from mc.svm into 'train'
Data loaded from mc.svm into 'train'
> normalize data train
Data 'train' normalized
> estimate model m from train ava
  class 1/2 (0): abcdefghijklmnopqrstuvwxyzABCDEFGHI...
  class 1/3 (1): abcdefghijklmnopqrstuvwxyzABCDEFGHI...
  class 2/3 (2): abcdefghijklmnopqrstuvwxyzABCDEFGHI...
  class 1/4 (3): abcdefghijklmnopqrstuvwxyzABCDEFGHI...
  class 2/4 (4): abcdefghijklmnopqrstuvwxyzABCDEFGHI...
  class 3/4 (5): abcdefghijklmnopqrstuvwxyzABCDEFGHI...
Model m estimated from train
> model info m
m has 1 clumps:
  0: dense   , weight=1, kernel=linear
8 support vectors, all-versus-all (4) classifier
```

Here, we can see for each pair of classes a different classifier being trained. This may seem like a lot more classifiers than in OVA, but each is being trained on less data, so it's not necessarily slower. Again, the model chooses 8 support vectors.

Prediction is similar (long lines are broken and indented):

```
> predict train with m output -
1 2:1:-0.128337 3:1:-0.125255 3:2:0.0114156
      4:1:-0.532561 4:2:-0.475452 4:3:-0.468716
1 2:1:-0.106242 3:1:-0.159651 3:2:-0.045132
      4:1:-0.533595 4:2:-0.499952 4:3:-0.432425
2 2:1:0.459376 3:1:-0.698836 3:2:-1.1598
      4:1:0.00543186 4:2:-0.478466 4:3:0.767402
2 2:1:0.43728 3:1:-0.664441 3:2:-1.10325
      4:1:0.00646582 4:2:-0.453967 4:3:0.731111
3 2:1:-0.689594 3:1:1.08971 3:2:1.78068
      4:1:0.0591978 4:2:0.795506 4:3:-1.11973
3 2:1:-0.680332 3:1:1.04279 3:2:1.72527
      4:1:0.00490774 4:2:0.723461 4:3:-1.13031
4 2:1:0.491253 3:1:0.535417 3:2:0.0108853
      4:1:2.13128 4:2:1.92631 4:3:1.83857
4 2:1:0.530813 3:1:0.490087 3:2:-0.0745069
      4:1:2.15635 4:2:1.91333 4:3:1.91645
```

Again, the first column is the chosen class (by number of "votes") and then each column is the output of a classifier. For instance, the first column, `2:1:-0.128337` says that when comparing class 2 to class 1, the function output was $-0.128337$. This means (since it is negative) that class 1 got a vote. Had it been positive, class 2 would have gotten that vote.

# Chapter 7

# Scripting

You might be put off by using the shell, but the true power of the shell is in flexible scripting. This is perhaps best served by example. Consider the following script, called `train.sss`:

```
#!/usr/local/bin/svmsequel
require 2 usage: train.sss <in.svm> <out.model> <args>

new data train
add sparse clump to train kernel rbf
load svm file $1 to train clump 0 from all
data info train
normalize data train
estimate model m from train $3+
model info m
predict train with m
save model m to $2
```

The first line of this script points to the svmsequel executable, so it can be run from the command line. The next command, `require`, takes one parameter, which is a number, saying that at least that many arguments must exist. If they do not, the script fails with the rest of the line as the error message. We can see this by running it without any arguments:

```
% ./train.sss
usage: train.sss <in.svm> <out.model> <args>
Fatal error: exception Script.Not_enough_arguments
```

The remainder of the commands we've see before. The only difference is the `$1`, `$2` and `$3+`. For standard variables (command line arguments), they can be referred to by number, starting with 1 (0 is the exectuable being run). So the first is used as the data file, the second is used as the output for the model. In general, `$n+` means to use arguments $n$ and above, so long as they exist.

We can run this on our multi-class data, but will need to specify "ova" in the additional arguments, but:

```
% ./train.sss mc.svm mc.model ova

Data train added
Clump #0 added to train
Parameters of clump #0 of train set
Loading data from mc.svm into 'train'
Data loaded from mc.svm into 'train'
train has 1 clumps and 8 points:
  0: sparse  , weight=1, kernel=rbf gamma=1., highest=1
Data 'train' normalized
  class 1: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 2: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 3: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 4: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
Model m estimated from train
m has 1 clumps:
  0: sparse  , weight=1, kernel=rbf gamma=1., highest=1
8 support vectors, one-versus-all (4) classifier
Accuracy = 8/8 = 100%
Model m saved to mc.model
```

Thus, we have more or less replicated (in a short script), the standard functionality of svm training programs.

Occasionally we might want more control over arguments than is allowed by the numbering, so we also allow named arguments. For instance, the above script does not allow you to specify kernel type or C value, which are things we would commonly wish to do. We do this using named arguments. We modify `train.sss` to:

```
#!/usr/local/bin/svmsequel
require 2 usage: train.sss <in.svm> <out.model> <args>
require kernel usage: need to specify kernel type
require c usage: need to specify c

new data train
add sparse clump to train kernel $kernel
load svm file $1 to train clump 0 from all
data info train
normalize data train
estimate model m from train c $c $3+
model info m
```

```
predict train with m
save model m to $2
```

Now, in addition to the two standard arguments, we also require a `kernel` and `c` named argument. We then reference those in the `add` and `estimate` lines. You specify these to the script by prefacing their name with an at sign, as in:

```
% ./train.sss mc.svm mc.model ova
usage: need to specify kernel type
Fatal error: exception Script.Not_enough_arguments

% ./train.sss mc.svm mc.model ova @kernel=rbf
usage: need to specify c
Fatal error: exception Script.Not_enough_arguments

% ./train.sss mc.svm mc.model ova @kernel=rbf @c=1
Data train added
Clump #0 added to train
Parameters of clump #0 of train set
Loading data from mc.svm into 'train'
Data loaded from mc.svm into 'train'
train has 1 clumps and 8 points:
  0: sparse  , weight=1, kernel=rbf gamma=1., highest=1
Data 'train' normalized
  class 1: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 2: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 3: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
  class 4: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM...
Model m estimated from train
m has 1 clumps:
  0: sparse  , weight=1, kernel=rbf gamma=1., highest=1
8 support vectors, one-versus-all (4) classifier
Accuracy = 8/8 = 100%
Model m saved to mc.model
```

The problem with this script is that it would be nice if we could default to some kernel if `kernel` is not specified, and similarly for `c`. Unfortunately, if we just remove the `require` lines, then once it gets to the `add` line, it will fail if `kernel` isn't specified. However, we can make the arguments *optional* by adding a `?` to their name in their usage. We can exemplify this with the `echo` command in the `echotest.sss` script (this also shows how to define our own local variables):

```
#!/usr/local/bin/svmsequel

@value = foo
echo Kernel is "$?kernel", value is $value
```

And run it:

```
% ./echotest.sss
Kernel is "", value is foo

% ./echotest.sss @kernel=rbf
Kernel is "rbf", value is foo

% ./echotest.sss @kernel='rbf gamma=2'
Kernel is "rbf gamma=2", value is foo
```

The obvious problem is that this won't correctly handle the case when the kernel is not specified, unless we require that the user say `@kernel='kernel rbf'`, which we don't wish to do. Thus, we introduce conditional variable definitions. In general, we want to say that if `kernel` is undefined, then it has no value; otherwise it has value `kernel $kernel`, where `$kernel` was it's old value. This is done with a simple modification to local variable assignment:

```
#!/usr/local/bin/svmsequel

@value = foo
@kernel =? kernel $?kernel | nothing
echo Kernel is "$kernel", value is $value
```

This line says that if kernel is defined, then to use the value before the vertical var; otherwise, use the value afterwards. We must use the "?" in the first part, so that we don't fail. Then, we know that the kernel variable will be defined in the echo statement:

```
% ./echotest.sss
Kernel is "nothing", value is foo

% ./echotest.sss @kernel='rbf gamma=2'
Kernel is "kernel rbf gamma=2", value is foo
```

Putting this all together, we get the following training script:

```
#!/usr/local/bin/svmsequel
require 2 usage: train.sss <in.svm> <out.model> <args>

@kernel =? kernel $?kernel |
@c      =? c $?c |

new data train
add sparse clump to train $kernel
load svm file $1 to train clump 0 from all
```

```
data info train
normalize data train
estimate model m from train $c $3+
model info m
predict train with m
save model m to $2
```

We can see the usage by:

```
% ./train.sss mc.svm mc.model ova
Data train added
Clump #0 added to train
Parameters of clump #0 of train set
Loading data from mc.svm into ’train’
Data loaded from mc.svm into ’train’
train has 1 clumps and 8 points:
  0: sparse  , weight=1, kernel=linear, highest=1
...

% ./train.sss mc.svm mc.model ova @kernel=rbf
Data train added
Clump #0 added to train
Parameters of clump #0 of train set
Loading data from mc.svm into ’train’
Data loaded from mc.svm into ’train’
train has 1 clumps and 8 points:
  0: sparse  , weight=1, kernel=rbf gamma=1., highest=1
```

In the first, it defaults to a linear kernel; in the second, the kernel is well-specified. A prediction script is even easier to implement:

```
!/usr/local/bin/svmsequel
require 2 usage: predict.sss <in.svm> <in.model>
                      (output=file)

@output =? output $?output |

load model m from $2

new data test
add sparse clump to test
load svm file $1 to test clump 0 from all
normalize data test by model m

predict test with m $output
```

We can run this by:

```
% ./predict.sss mc.svm mc.model
Model m loaded from to mc.model
Data test added
Clump #0 added to test
Parameters of clump #0 of test set
Loading data from mc.svm into 'test'
Data loaded from mc.svm into 'test'
Data 'test' normalized by model 'm'
Accuracy = 8/8 = 100%

% ./predict.sss mc.svm mc.model @output=-
Model m loaded from to mc.model
Data test added
Clump #0 added to test
Parameters of clump #0 of test set
Loading data from mc.svm into 'test'
Data loaded from mc.svm into 'test'
Data 'test' normalized by model 'm'
1 1:0.443482 2:-0.492971 3:-0.745604 4:-0.743417
1 1:0.432369 2:-0.463073 3:-0.772513 4:-0.74618
2 1:-0.316357 2:0.406548 3:-0.746161 4:-0.729692
2 1:-0.292845 2:0.379392 3:-0.766047 4:-0.734562
3 1:-0.278431 2:-0.481259 3:0.716481 4:-0.75955
3 1:-0.255501 2:-0.49948 3:0.694688 4:-0.769405
4 1:-0.253646 2:-0.49132 3:-0.782601 4:0.89087
4 1:-0.254942 2:-0.486033 3:-0.785759 4:0.892467
Accuracy = 8/8 = 100%
```

Also included in the scripts directory is a `cv.sss` script for performing standard cross-validation.

# Chapter 8

# “Undocumented” Features

The following documentation negates the title of this chapter, but nonetheless, there are a few commands we haven’t discussed yet that don’t really fit into other chapters.

## 8.1 Modify Clump

Once you’ve added a clump, you can change any of it’s parameters (though not the clump type) by saying:

```
modify clump <n> of <dataset> to <args>
```

With the obvious filling-in of parameters. The args are any args that can be used during creation (i.e., `add clump`) of a clump.

## 8.2 Load Kernel Matrix

If you want to use your own kernel matrix, you need to load it into a matrix clump by saying:

```
load kernel matrix from <path> to <dataset> clump <n>
```

With the obvious interpretation. The file should be space separated and should contain as many lines as columns. Since the matrix is symmetric, it can also be an upper-triangular matrix. Two equivalent examples are:

```
1  2  3  4
2  5  6  7
3  6  8  9
4  7  9  0
```

and

```
1  2  3  4
   5  6  7
      8  9
         0
```

of course, the data need not actually line up nicely like this – that was just for you to see it better.

## 8.3 Load and Save Environment

Sometimes it’s useful to save every dataset, model and variable, so you can quit and restart exactly in the same place as before. This is easily done:

```
save environment to <path>
load environment from <path>
```

will do this, though the files can get quite big.

## 8.4 Running Scripts

You can run scripts as a sort of “subroutine” if you wish, by saying:

```
run <script> <args>
```

Where `<script>` is the name of the script file to run, and `<args>` are the relevant arguments, as if they had been specified on the command line.

## 8.5 Clearing Data

You can remove data sets and models from memory (if you’re running low) by saying one of:

```
clear data <dataset>
clear model <model>
```

with the obvious interpretation.