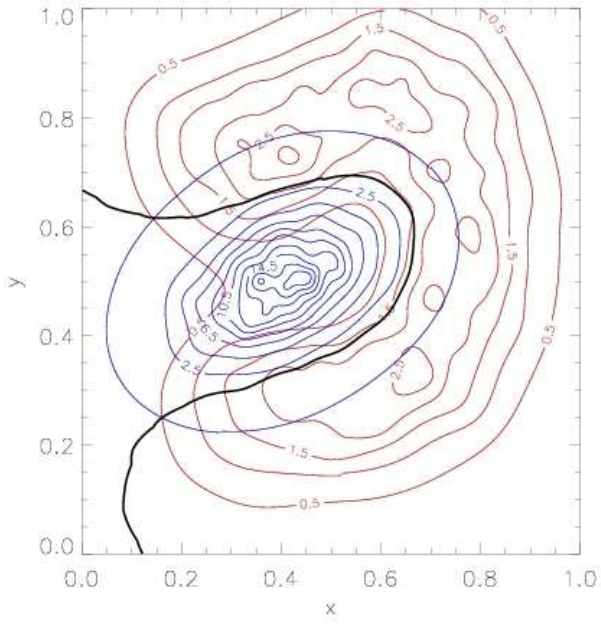
libG User Guide

Peter Mills
peteymills@hotmail.com

August 28, 2019



CONTENTS

1 INSTALLATION

First, make sure you have the required dependencies. You will need the *GNU Scientific Library (GSL)*. If it is not already installed on your system, download and install it. Take note of the location of the compiled libraries and include files. You will also need to install *libpetey*, another, much smaller numerical library–this now comes bundled with libAGF.

Download the source from Github (`https://github/peteysoft/libmsci`) or your favourite mirror. Unzip and unpack the tarball to the desired directory. Modify the makefile with the appropriate directories, compiler names and compiler flags. Type `make` from inside the libagf directory to perform the build and `make install` to copy the executables, libraries and include files to the appropriate directories.

A configure script is not supplied: please refer to the Peteysoft coding standards `coding_standards.txt` in the base of the installation directory (`https://github.com/peteysoft/libmsci/blob/master/doc/coding_standards.txt`).

Make sure the following macros have been set properly in the make file:

`LIB_PATH` =	location of your compiled libraries
`BIN_PATH` =	location of binary executables
`INCLUDE_PATH` =	location of your include files
`GSL_LIB` =	location of the GSL libraries (if different from above)
`GSL_INCLUDE` =	location of the GSL include files (if different from above)
`CC` =	name of your C++ compiler

There are two ways to change the optimisation of the binaries. If you would like multiple versions with different levels of optimisation, be sure to change the OPT_VER macro. This will append the optimisation flag to all the object files, libraries and executables. If you only want one version, then modify the CFLAGS macro in the usual manner.

To test the resulting executables, type, `make test`. To see the resulting comparison summary, it may be necessary to type it twice. Overall accuracies of all the classification algorithms should be around 90 % and the uncertainty coefficients should be over 0.5, while the confidence ratings should roughly match the accuracies. When comparing one algorithm to another, accuracies should be around 99% and uncertainty coefficients around 0.92, while the confidence ratings should be 1. or close to it, regardless of accuracy. Prob-

ability density comparisons should be around $r = \{0.98, 0.99, 0.97, 0.99\}$ for KNN and AGF for the first class and KNN and AGF for the second class, respectively.

2 FILE FORMATS

The programs accept three types of files with very simple, binary formats. Although portability is sometimes an issue, the binary formats are easy to generate and allow for rapid access and compact storage. For vector data, use the extension, `.vec`. Vector files have a four byte header (by default, see next paragraph) that indicates the dimensionality of the data. After that, each vector is stored one after the other as arrays of four byte floats. To write such a file in C++, you can use the following commands:

```
#include <stdio.h>
#include <stdint.h>

int main() {

  FILE *fs;

  float ** data; //array of vectors
  int32_t n; //number of vectors
  int32_t D; //number of dimensions

  n=100; //(e.g.)
  D=2;

  data=new float * [n];
  for (int32_t i=0; i<n; i++) data[i]=new float[D];

  //... commands to fill the matrix ...

  fs=fopen("data.vec", "w");
  fwrite(&D, sizeof(D), 1, fs);
  for (int32_t i=0; i<n; i++) fwrite(data[i], sizeof(float), D, fs);
  fclose(fs);
```

```
  for (int32_t i=0; i<n; i++) delete [] data[i];
  delete [] data;

}
```

In IDL (Interactive Data Language):

```
n=100 ;(e.g.)
d=2 ;(e.g.)
data=fltarr(d, n)
;... commands to fill the matrix ...
openw, 1, "data.vec"
writeu, 1, d, data
close, 1
```

In Fortran:

```
parameter(n=100)
parameter(d=2)
real data(d, n)

...

open(10, file="data.vec", form="binary")

write(10) d
do j=0, n
  write(10) (data(i, j), i=1,d)
enddo

close(10)
```

For scalar data, the programs use a straight binary dump of the data. By default, four byte (32 bit) integers are used for class data and four byte (32 bit) floating point variables are used for continuous data. (To change the types used by the command-line programs, edit the header `agf_defs.h`.) For class data, use the extension `.cls` and for floating point data, use the

extension, `.dat`. Scalar data always corresponds to vector data and is ordered accordingly.

To write such data in C++, you can use the following commands:

```
#include <stdio.h>
#include <stdint.h>

int main() {
  FILE *fs;
  int32_t *data;
  int32_t n;

  n=100; //(e.g)
  data=new long[n];
  //... commands to fill array  ...
  fs=fopen("data.cls", "w");
  fwrite(data, sizeof(int32_t), n, fs);
  fclose(fs);
  delete [] data;
}
```

In IDL:

```
data=lonarr(100)
;...
openw, 1, "data.cls"
writue, 1, data
close, 1
```

In Fortran:

```
parameter(n=100)
integer data(n)

...

open(10, file="data.cls", form="binary")

write(10) (data(i), i=1,n)
```

```
close(10)
```

Note that the class indices must go from 0 to $N_c - 1$, where N_c is the number of classes. The classification routines will output two binary files containing scalar data: one with the extension, `.cls` containing class estimates, and one with the extension `.con` containing "confidence ratings" which are simply conditional probabilities re-scaled in the following manner:

$$c = \frac{N_c P(c|\vec{x}) - 1}{Nc - 1} \tag{1}$$

where $P(c|\vec{x})$ is the conditional probability of the "winning" class. A confidence rating of one indicates theoretically perfect knowledge of the true class, while a value of zero indicates that the result is little better than chance. If conditional or joint probabilities are needed for all the classes, they are written to standard out.

To read the output files, examine the routines `read_clsfile` for reading class data, `read_vecfile` for reading vector data and `read_datfile` for reading scalar floating point data, contained in the source file, `agf_util.cc`, or you can use them directly. Note that vector data is allocated in one contiguous block, meaning that it can also be read and written in one contiguous block. It can also be deleted very simply using two commands:

```
float ** data;
nel_ta n;
dim_ta D;

data=read_vecfile(data, n, D);

delete [] data[0];
delete [] data;
```

In addition, the *sparse* satellite library contains a utility called `sparse_calc` that can operate on both types of binary files containing floating point data (but not class data). To read in and print a file of vector data called, `foo.vec` type the following commands:

```
$>sparse_calc
%s%c>print full(foo.vec)
```

To read in and print a file of scalar data called, `bar.dat`, type the following:

```
%s%c>print vector(bar.dat)
```

Some users may prefer to use ASCII file formats to store their data. If this is the case, there are a number of file conversion utilities included for comparison with other algorithms. See Section 4, COMMAND LINE EXECUTABLES.

Sometimes it's desireable to normalize the data before performing classifications. Normalization in libagf has now been generalized to linear transformation and includes singular value decomposition as well as feature selection. The command, `agf_precondition`, is used to perform linear transformations on test and training (features) data and store the resulting transformation matrix in the same binary format for vector data as described above. The default name is the same as before with the extension `.std` appended to the output base file name. Features data can still be transformed directly within the machine learning routines using the command options `-n`, `-S` and `-a`: see Section 3, COMMAND PARAMETERS. By default, pre-trained model data is stored in the normalized coordinates; to store it in the non-normalized coordinates, use the command switch, `-u`.

3 COMMAND PARAMETERS

Most of the commands have the following syntax:

```
command [options] model [test_data] output
```

where `model` is the base name of the files containing either a set of training data or a pre-trained model, `test_data` is the file containing test data and output is the base name of the files in which to store the final estimates.

To get a summary of the syntax and parameters, simply type the command name with no arguments, e.g.:

```
> classify_b
Syntax:   classify_b [-n] [-u] [-a normfile] border test output

where:
```

```
  border      files containing class borders:
                .brd for vectors
                .bgd for gradients
                .std for variable normalisations (unless -a specified)
  test        file containing vector data to be classified
  output      files containing the results of the classification:
                .cls for classes, .con for confidence ratings

options:
  -n          option to normalise the data
  -u          normalize borders data (stored in un-normalized coords)
  -a normfile file containing normalization data
```

Here is a list of all the options, although not all commands support all options:

Option	Function
–	External command to train a binary classifier.
-+	Extra options to pass to binary training command.
-^	External command to convert data formats for above.
-0	Read from stdin.
-1	Write to stdout.
-a	Name of file containing normalization data.
-A	Use ASCII files instead of binary.
-b	Short output (`cls_comp_stats`)
-B	Sort by class/ordinate.
-c	Algorithm selection: nfold cross-validation: 0=AGF borders (default) classification 1=AGF classification 2=KNN classification 3=AGF interpolation 4=KNN interpolation 5=AGF borders multi-class PDF validation: 6=AGF PDF estimation 7=KNN PDF estimation
-C	No class data.
-d	Number of divisions in cross-validation scheme.
-D	Generate ROC curve by varying the discrimination border.
-e	Return error estimates.
-E	Value for missing data.
-f	For validation schemes: fraction of training data to use for testing.
-F	Select features.
-g	Use lograthmic progression.
-h	Relative difference for calculating numerical derivatives.

-H Omit header data.
-i Class borders: maximum number of iterations in supernewton.
-I Weights calculation: maximum number of iterations in supernewton.
-j Print joint instead of conditional probabilities to stdout.
-k Number of k-nearest-neighbours to use in the estimate
-K Keep temporary files (do not delete on exit).
-l Tolerance of W.
-L Floating point ordinates.
-m Type of metric to use. Only works for knn classification.
-M Get min. and max./LIBSVM format.
-n Takes averages and standard deviations of the training data and normalizes both the training and test data.
-N Maximum number of iterations in supernewton.
-o Name of log file.
-O External command for binary classification prediction.
-p Threshold density in clustering algorithm.
-P Calculate correlation/covariance matrix.
-q Number of trials/divisions.
-Q Algorithm selection:
optimal AGF: how to calculate the filter variances
0=halving max filter variance (default);
1=filter variance min and max;
2=total weight min and max
non-hierarchical multi-class classification:
0=constrained inverse (not always optimal)
1=linear least squares, no constraints or re-normalization
2=voting from pdf, no re-normalization
3=voting from class label, no re-normalization
4=voting from pdf overrides least squares, conditional probabilities are adjusted and re-normalized
5=voting from class overrides least squares, conditional probabilities are adjusted and re-normalized
6=experimental
7=constrained inverse (may be extremely inefficient)

-r Class borders: value of conditional prob. at discrimination border.
-R For random sampling.
-s Number of times to sample the class border.
-S Singular value decomposition: number of singular values to keep.
-t Desired tolerance when searching for class borders.
-T Class threshold for class-borders calculation.
-u Store borders data in non-normalized coordinates.
-U Re-label classes to go from $[0, nc)$.
-v First filter variance bracket.
-V Second filter variance bracket/initial filter variance.
-w Lower bound for W in AGF optimal/constraint weight
-W Parameter Wc–equivalent to k in a k-nearest-neighbours scheme. See paper describing the theory.
-x Run in background.
-X Ratio between sizes of sample classes ($P(2)/P(1)$).
-z Randomize data.
-Z In-house LIBSVM codes.

4 COMMAND LINE EXECUTABLES

This is a list of commands and their function:

4.1 Direct routines

Direct, non-parametric classification, interpolation/regression, pdf estimation using variable-bandwidth kernel estimation:

`agf` Uses adaptive Gaussian filtering
`knn` Uses k-nearest neighbours

All the direct kernel estimation operations have been collected into two executables: one for AGF called, `agf`, and one for k-nearest-neighbours called, `knn`. Both of them take an extra argument which specifies which operation to perform: `classify`, `interpol` or `pdf`.

4.2 Binary classification

Two-class classification by training a model:

`class_borders` Searches for the class borders using AGF.
`classify_b` Performs classifications using a set of border samples.

4.3 Multi-class classification

Multi-class classification using a series of class-borders:

`print_control`	Prints out control files for standard multi-class schemes.
`multi_borders`	Trains a multi-class model based on a control file.
`classify_m`:	Performs classifications using model output from `multi_borders`.
`svm_accelerate`:	Trains a multi-borders model from a native LIBSVM model.
`mbh2mbm`:	Compiles separate binary files into "all-in-one" model.

Multi-borders classification represents a significant step forward with this library. The operation of these programs, called from the command line using `multi_borders` and `classify_m` is quite involved and is described in detail in Section, 6, MULTI-BORDERS CLASSIFICATION.

4.4 Testing/validation

`cls_comp_stats`	Calculates the accuracy of classification results.
`nfold`	Performs n-fold cross-validation for the three different classification algorithms or for the interpolation routines.
`agf_preprocess`:	Splits a dataset for validation schemes.
`pdf_sim`	Generates a synthetic dataset with the same approximate PDF as the training data.
`validate_pdf.sh`	Validates probability density function (PDF) estimates.
`agf_correlate`	Correlates two binary files containing scalar floating point data.
`roc_curve`	Computes the receiver operatoring characteristic (ROC) using three different methods.

4.5 Clustering

`cluster_knn`	Uses k-nearest neighbours and a threshold density to perform a clustering analysis. See description, below.
`browse_cluster_tree`	Create a dendrogram and manually browse through it and assign classes.

The first clustering algorithm works by calculating the density of each training samples. If the density is larger than a certain threshold, a class number is assigned and the program recursively calculates the densities of all the samples in the vicinity, based on its k neighbours, and assigns the same class number to each if they also exceed the threshold. Samples lower than

the threshold are assigned the class label of 0, while clusters are assigned consecutive values starting at 1. This is far simpler than a dendrogram, but somewhat less general although the final result should be similar. But, we've added a dendrogram anyway.

4.6 Experimental

Continuum extension : Programs that extend the classification algorithms to work with continuous data:

`c_borders.sh`	Trains a continuum classification model.
`classify_c`	Returns coninuum extension estimates.

Discrete Bayesian modelling : programs that model the probability distributions through binning.

`sort_discrete_vectors`	Sorts a set of discrete vectors.
`search_discrete_vectors`	Searches within a set of discrete vectors and estimates the probability density.

4.7 Pre-/post-processing

`agf_procondition`	Performs linear transformations on coordinate (feature) data and outputs the transformation matrix. Normalization, singular-value-decomposition (SVD) and feature selection are supported.
`agf_preprocess`	Processes both coordinate data and class data. Supports class selection, re-labelling and partitioning, file splitting for validation and cross-correlation calculation.

4.8 File conversion

`lvq2agf`	Converts the LVQPAK (ASCII) file format to the binary format accepted by libagf. Many users may find these ASCII formats easier to work with.
`svm2agf`	LIBSVM format to libAGF.
`svmout2agf`	Converts (ASCII) output from LIBSVM to binary libAGF format.
`agf2ascii`	Converts binary libAGF format to ASCII (LVQPAK or LIBSVM) format.
`C2R`	Converts a two-class classification to a difference in conditional probabilites. The equation is: $R = (2c - 1)C$ where c is the class and C is the confidence rating.
`mbh2mbm`:	Collates a multi-file multi-borders model into a single ASCII file.
`float_to_class`:	Converts floating point data into class data by binning it into discrete ranges.
`class_to_float`:	Converts class data into floating point.

The LVQPAK file format is probably the easiest to work with. There is one header with the number of dimensions, followed by a listing of the vectors, one vector per line, each column is a dimension except for the last one which is the class. The `sample_class` program prints its results to standard out in an LVQPAK-compatible format–see Section 7, EXAMPLES.

4.9 Commands used by other commands

Note that some commands call other commands, therefore these latter commands must be in your path.

`agf_precondition`	Called by all the machine learning programs if one or more of the `-a`, `-n` or `-S` switches are used
`pdf_agf`, `pdf_knn`	still used by `validate_pdf.sh`

4.10 Deprecated

`classify_a`, `classify_knn`, `int_agf`, `int_knn`, `test_classify`, `test_classify_b`, `test_classify_knn`

To compile and install older routines, type, `make old`.

5 ERROR CODES AND DIAGNOSTICS

All commands will return "0" upon successful completion or one of the following error codes:

Code	Meaning
1	Wrong or unsufficient number of arguments.
101	Unable to open file for reading.
111	File read error.
256	File read warning.
303	Allocation failure.
201	Unable to open file for writing.
211	File write error.
512	File write warning.
401	Dimension mismatch.
411	Sample count mismatch.
501	Numerical error.
511	Syntax error.
503	Maximum number of iterations exceeded.
768	Parameter out of range.
1024	Command option parse error.
21	Fatal command option parse error.
901	Internal error.
911	Other error.
1280	Other warning.

Note that warnings are always divisible by 256 so that when passed back to the command line return a 0 exit status so as not to interupt running scripts. Error codes are also listed in the include file, `error_codes.h` in the libpetey distribution. Note that non-fatal errors reduce to '0' if passed to the command line.

AGF routines also return a set of diagnostics, for example:

```
diagnostic parameter               min        max    average

iterations in supernewton:**         4         52       11.8
tolerance of samples:                  0   3.53e-05   2.59e-06

** total number of iterations:     1178
   number of bisection steps:      276
```

```
   number of convergence failures: 0


diagnostic parameter                min         max     average

iterations in agf_calc_w:             2           6        3.73
value of f:                     2.6e-13       0.284     0.00146
value of W:                       20.00       20.00       20.00

total number of calls: 1430
```

The first parameter indicates the number of iterations required to reach the correct value for the total of the weights, W. For efficiency reasons, this should be as small as possible, however the super-linear convergence of the root-finding algorithm ("supernewton") means that the brackets can be quite far away without much effect on efficiency. To change the values of the filter variance used to bracket the root, use `-v` for the lower bracket and -V for the upper bracket. Normally the defaults should work just fine but to decrease the number of iterations, they should be narrowed. If they fail to bracket the root, the offending bracket is pushed outward. These changes are "sticky", so the brackets can, in fact, be set arbritrarily narrow and even quite far from the root.

To change the tolerance of W, use the `-l` switch. To change the maximum number of iterations, use the `-i` or `-I` switch.

The second parameter is the number of iterations required to converge to the class border, excluding convergence failures. This is only applicable when searching for the class borders. To change the maximum number of iterations (default is 100) in the root-finding algorithm, use the `-i` or `-N` parameter.

The third parameter is the value of the minimum filter weight divided by the maximum and is only applicable when the `-k` option (selecting a number of nearest neighbours) has been set. Ideally, it should be as small as possible, although values as large as 0.2 often produce reasonable results. Be sure to experiment with your own particular problem. To decrease it, increase the number of nearest neighbours used in the calculations which has the undesirable side effect of increasing computation time. In general, k should be a fair bit larger than W.

The tolerance of the samples only applies when searching for class borders. It shows how close the value of $R = P(2|\vec{x}) - P(1|\vec{x})$ (difference in conditional probabilities) is to zero at each border sample. This is set using the -t option and the diagnostic should be close to the set value. It is sometimes larger because the convergence test of the root-finding routine looks at tolerance along the independent variable as well as the dependent variable–whichever is better.

To best understand how to interpret the diagnostics and use these to set the operational parameters, be sure to read the paper entitled, "Adaptive Guassian Filters: a powerful new method for supervised learning", in the `doc/` sub-directory of this installation or Mills (2011) which is a more fleshed-out version of it.

6 MULTI-BORDERS CLASSIFICATION

Multi-borders is an algorithm that generalizes the AGF-borders binary classification algorithm to multiple classes. It is a means of specifying the configuration of the class borders using a recursive control language. The `multi_borders` command is used for training a multi-borders model and has the following syntax:

```
multi_borders [options] control-in train model control-out
```

where:

- `control-in` is the input or training control file;
- `train` is the base-name of the binary files containing the training data
- `model` is the base-name for the files that will contain each of the binary classification models and
- `control-out` is the output or classification control file which is passed to the classification program, `classify_m`.

6.1 Multi-borders classification the easy way

To train a four-class one-versus-one multi-class class borders classification model, use the `print_control` command to generate one of several standard multi-class models and pass it directly to the `multi_borders` command:

```
$>print_control -Q 6 4 | multi_borders foo bar foobar.txt
```

The training data is contained in `foo`; the trained binary classification models are output in a series of files (six in this case) starting with `bar`. The output control file, `foobar.txt`, tells the prediction program, `classify_m` how the binary classifiers are configured:

```
$>classify_m foobar.txt test.vec results
```

Where `test.vec` contains the test points while the results are stored in `result.cls` and `result.con`. The `-Q` options determines the multi-class model used: type `print_control` with no arguments to get a list of supported models.

6.2 Multi-borders classification the hard way

Consider the following control file (`test_multi5.txt`):

```
"-s 100" {
  "."      {0 1}
  ""       0 1 / 2;
  "-s 75"  0 / 1 2;
  {
    "-s 50" {2 3}
    4 5
  }
}
```

When passed to `multi_borders` as follows:

```
$>multi_borders -n -s 125 test_multi5.txt foo bar foobar.txt
```

runs the following set of statements:

```
agf_precondition -a bar.std -n foo.vec bar.139397543805655.vec
cp foo.cls bar.139397543805655.cls
class_borders -s 100 bar.139397543805655 bar.00 0 / 1
class_borders -s 50 bar.139397543805655 bar.01.00 2 / 3
class_borders -s 125  bar.139397543805655 bar.01-00 2 3 4 / 5
class_borders -s 75 bar.139397543805655 bar.01-01 2 3 / 4 5
class_borders -s 100 bar.139397543805655 bar 0 1 / 2 3 4 5
rm bar.139397543805655.vec
rm bar.139397543805655.cls
```

Initial versions printed these commands to standard out with running them the job of the user. Now they are executed directly with the option of running them in the background using the `-x` switch. It is still possible to print out the commands only using the `-O` or `-K` switch.

The contents of `foobar.txt` are as follows:

```
bar {
  bar.00 {
    0
    1
  }
  bar.01-00 0 1 / 2;
  bar.01-01 0 / 1 2;
  {
    bar.01.00 {
      2
      3
    }
    4
    5
  }
}
```

Note that each of the names in this file correspond to pairs of files generated by the commands in the previous script. To classify the data in a file named `test.vec` use the following command:

```
$>classify_m -n foobar.txt test.vec result
```

There are two types of multi-borders classification: *hierarchical* and *non-hierarchical*. In non-hieararchical multi-borders classification, all the classes are partitioned in multiple ways using a binary classifier and the equations relating the conditional probabilities of each class to those of the binary classifiers are solved returning all of the conditional probabilities. In the hierarchical method (also called a *decision tree*), the classes are partitioned using either a binary classifier or a non-hierarchical multi-classifier, then each of those partions are partitioned again and so on until there is only one class left in each of the partions. Hierarchical classification returns only the conditional probability of the winning class. The `classify_m` method

automatical detects whether a control file uses the hierarchical method or only the non-hierarchical method(that is it has only one level) and prints out conditional probabilities as appropriate.

The parameters for training each binary classifier are contained in double quotes in the training control file. To take parameters from the command line, use the null string (`""`) while a period (`"."`) tells the program to use the last set of parameters at the same level or below. A series of statements for training each of the binary classifiers are generated and executed using the "system" command. Once the commands are run, these binary classifiers will be stored in a pair of uniquely named files, the base-names of which replace, in the final control file, the quoted parameter lists from the training control file.

In the control language, going up a level in the hierarchical scheme is denoted by a left curly brace (`{`) while going down is denoted by a right curly brace (`}`). In a non-hierarchical model, we specify the parameters (file name) of each of the binary partitions followed by the partitions themselves: two lists of classes separated by a forward slash (`/`). Class labels for non-hierarchical partitions are relative, that is they go from 0 to the number of classes in the non-hierarchical model less one. Class labels in the top level partitions are absolute, therefore must be unique. These should also go from 0 to one less than the total number of classes in the over-all model, but need not.

In this example, there are six classes. They are first partioned into a group of two and a group of four. The group of four is partitioned into three parts, the first of which is partitioned into two classes and the second and third of which are single classes. The `print_control` command can be used to generate common control files.

A good way to understand the "multi-borders" paradigm is to look at the example cases in the `examples/humidity` data directory. There is also a draft paper contained in the `docs/` sub-directory.

6.3 Multi-borders with external binary classifiers

The multi-borders routines can now interface with external binary classification software, specifically, either LIBSVM, or a pair of programs that have the same calling conventions. For training, use the `--` switch to pass the command name to `multi_borders`. Consider the above control file as an example, but without any of the control switches:

```
"" {
  ""  {0 1}
  ""  0 1 / 2;
  ""  0 / 1 2;
  {
    "" {2 3}
    4 5
  }
}
```

We pass the LIBSVM command, `svm-train`, to `multi_borders`, as follows:

```
$>multi_borders -M -- "svm-train -b 1" -+ "-h 0 -c 25" \
test_multi.txt foo.svm bar foobar.txt
```

which will execute the following statements:

```
agf_preprocess -A -M foo.svm bar.00.2367545368.tmp 0 / 1
svm-train -b 1 -h 0 -c 25  bar.00.2367545368.tmp bar.00
rm -f bar.00.2367545368.tmp
agf_preprocess -A -M foo.svm bar.01.00.2367545368.tmp 2 / 3
svm-train -b 1 -h 0 -c 25  bar.01.00.2367545368.tmp bar.01.00
rm -f bar.01.00.2367545368.tmp
agf_preprocess -A -M foo.svm bar.01-00.2367545368.tmp 2 3 4 / 5
svm-train -b 1 -h 0 -c 25  bar.01-00.2367545368.tmp bar.01-00
rm -f bar.01-00.2367545368.tmp
agf_preprocess -A -M foo.svm bar.01-01.2367545368.tmp 2 3 / 4 5
svm-train -b 1 -h 0 -c 25  bar.01-01.2367545368.tmp bar.01-01
rm -f bar.01-01.2367545368.tmp
agf_preprocess -A -M foo.svm bar.2367545368.tmp 0 1 / 2 3 4 5
svm-train -b 1 -h 0 -c 25  bar.2367545368.tmp bar
rm -f bar.2367545368.tmp
```

Several things should be noted:

- the `-b` switch is required and tells `svm-train` to generate probability estimates
- the `-+` option passes any other switches to use as "defaults"

- when used in this way, training data must be in ASCII format: `-M` switch for the same format as LIBSVM, otherwise it uses the same format as LVQPAC (see File Conversion, above)
- the output control file is the same as before

The training command, passed through `--`, should have the following syntax:

```
train [options] data model/
```

where:

- `train` is the training command
- `options` are a set of options passed through the control file, through the `-+` option, or directly from the end of the command line or through the command name
- `data` is the training data in LVQ or SVM ASCII format
- `model` is the output model reconizable to the prediction command, see below.

Once the training has completed, classifications are performed by passing the prediction command, `svm-predict`, from LIBSVM to `classify_m` using the `-O` option:

```
$>classify_m -M -O "svm-predict -b 1" foobar.txt test.svm output.svmout
```

Once again, when paired with an external command, `classify_m` operates on ASCII files as opposed to the native AGF binary format. Output file format is the same as LIBSVM: a header consisting of the word, "labels", followed by a list of labels, then one class label per line, followed by the conditional probabilities in the same order as the class labels in the header. Since "hierarchical" classification generates only one probability per estimate, in this case only one is written per line. While this is not LIBSVM conformant, the file conversion utility, svmout2agf, nonetheless recognizes it.

The prediction command, passed by `-O`, should have the following syntax:

```
predict test model output
```

where:

- `predict` is the command name–if there are options they must be passed directly as part of this name;
- `test` is the test data in LVQ or SVM ASCII format;
- `model` is the binary classification model;
- `output` are the predicted classes plus both conditional probabilities in the format described above.

Note that the order of the first two arguments are reversed as compared to the libAGF convention in `classify_b` and `classify_m`.

LIBSVM tends to be slow, especially if you have a lot of training data. There are currently three ways of converting LIBSVM models into borders models: the first two use the `multi_borders` command and work on LIBSVM/multi-borders hybrid models. First, you can used the `-O` switch to pass the prediction command (`svm-predict -b 1` for LIBSVM models) to `multi_borders` so that it can train a faster multi-borders model. This solution has the advantage that it can work with any external binary classifier, not just LIBSVM. It has the disadvantage, however, that gradient vectors are calculated numerically, so is not very accurate.

A better solution is the `-Z` switch which tells `multi_borders` to use the "in-house" SVM codes. To accelerate the previous model, pass the output classification control file from the previous pass:

```
$>multi_borders -Z foobar.txt foo bar2 foobar2.txt
```

which executes the following statements:

```
class_borders -Z bar.00 foo bar2.00
class_borders -Z bar.01.00 foo bar2.01.00
class_borders -Z bar.01-00 foo bar2.01-00
class_borders -Z bar.01-01 foo bar2.01-01
class_borders -Z bar foo bar2
```

Note how we're using `class_borders` to perform the training: it uses SVM as a source of conditional probabilities and finds a series of roots (zeroes) to sample the border between the two classes, just as it would with AGF estimates. To facilitate this, an extra parameter is included in the

`class_borders` command: in addition to the output file name in the last parameter, the first parameter is the name of the model used to predict the probabilities, while the second parameter is a file containing training data which is used to sample the space while searching for roots.

Output is in the normal, AGF binary format and `classify_m` can be used as normal for classification:

```
$>classify_m foobar2.txt test.vec output
```

Another way of accelerating LIBSVM models, which only works with "native" LIBSVM models, is with the `svm_accelerate` command. Suppose we've trained a LIBSVM model, `model.svmmod`, as follows:

```
$>svm-train foo.svm model.svmmod
```

We can convert it to a multi-borders model as follows:

```
$>svm2agf foo.svm foo
$>svm_accelerate model.svmmod foo model.mod
```

where `foo.svm` contains the training data in LIBSVM format and `foo` is the training data in libAGF format. Note that the multi-borders model in this case is stored in ASCII format all in one file but is still accepted by `classify_m`:

```
$>classify_m model.mod test.vec output
```

If a multi-file, multi-borders model is in one of the following configurations: one-vs-one, one-vs-the-rest, or partitioning of adjacent classes, then you can convert it to a single ASCII file using the `mbh2mbm` command. The file format fairly simple. There is a four line header with the following information: type of configuration ("1v1", "1vR", "ADJ"), number of classes, list of class labels, and "polarities" of each of the binary classifiers. Next, all the binary classifiers are listed as pairs of matrices: first the border vectors, then the gradient vectors. Each matrix has a one line header with the number of vectors plus the size of each vector.

Running the commands, `print_control`, `multi_borders`, `classify_m`, `class_borders`, `svm_accelerate`, or `mbh2mbm` without arguments prints a generous help screen including a description of most of the features explained in this section. There is also an example contained in the second half of the makefile under the `examples/humidity_data` directory (see, Section 7, EXAMPLES, above).

7 EXAMPLES

The `examples` sub-directory collects together a number of test suites for comparison, validation and application of AGF. The makefiles in these test cases can give you a good idea of how to use the various components of the library.

7.1 examples/class_borders

The directory `examples/class_borders` includes a number of routines for testing the algorithms and comparing them with other, popular classification algorithms. The validation exercise is performed on a pair of two-dimensional, synthetic test classes and is described in Mills (2011). To test the classification algorithms, type, `make test`. To test the pdf estimation routines, type, `make test_pdf`. To test both from the bottom level directory, type `make test`. All tests are done on a pair of synthetic test classes, described in the paper. To generate samples and analytical estimates for these classes, use the following commands

`sample_class`	Generates random samples of the synthetic test classes.
`classify_sc`	Returns class estimates using analytic/semi-analytic estimates of the class pdfs; classifications should therefore be close to the best possible for any supervised algorithm.
`pdf_sc1`	Generates analytic pdf estimates for the first class.
`pdf_sc2`	Generates semi-anaylitic (using quadrature) pdf estimates for the second class.
`sc_borders`	Find the border between the sample classes using the same algorithm as that employed for the `class_borders` command.

To compare the algorithms with LIBSVM (Chang and Lin, 2011) and Kohonen's LVQ algorithm (Kohonen, 2000), type, `make compare`. You must have both modules installed and the executables in your path.

7.2 examples/humidity_data

The test suite in `examples/humidity_data` tests the multi-borders paradigm on a discretized sub-set of the satellite humidity data described in Mills (2009). Use this directory for examples on how to use the multi-borders multi-class classification method.

7.3 examples/Landsat

In the Landsat directory there are a number of scripts for performing surface classifications using Landsat data. In particular, there is a simple app that allows the user to classify pixels by hand. It opens a window with a Landsat scene in it; clicking one of the three mouse buttons allows you to classify pixels in the scene. There are three files that already contain hand-classified forest clearcut data.

BIBLIOGRAPHY

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27.

Kohonen, T. (2000). *Self-Organizing Maps*. Springer-Verlag, 3rd edition.

Michie, D., Spiegelhalter, D. J., and Tayler, C. C., editors (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood Series in Artificial Intelligence. Prentice Hall, Upper Saddle River, NJ. Available online at: `http://www.amsta.leeds.ac.uk/~charles/statlog/`.

Mills, P. (2009). Isoline retrieval: An optimal method for validation of advected contours. *Computers & Geosciences*, 35(11):2020–2031.

Mills, P. (2011). Efficient statistical classification of satellite measurements. *International Journal of Remote Sensing*, 32(21):6109–6132.

Mills, P. (2018a). Accelerating kernel classifiers through borders mapping. *Real-Time Image Processing*. doi:10.1007/s11554-018-0769-9.

Mills, P. (2018b). Solving for multi-class: a survey and synthesis. Technical Report arxiv:1809.05929.

Müller, K.-R., Mika, S., Rätsch, G., Tsuda, K., and Schölkopf, B. (2001). An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12(2):181–201.

Terrell, D. G. and Scott, D. W. (1992). Variable kernel density estimation. *Annals of Statistics*, 20:1236–1265.