A no-nonsense beginner's tool for GMLVQ

Version 3.0

March 13, 2019

Disclaimer

1 Getting started

To get started using this library in Matlab, you'll need to first set up the library correctly. This can be done by extracting the archive into some directory, and adding that directory to your Matlab path. Alternatively, start Matlab in that directory directly. You can test whether the library loaded correctly by trying to type the first letters of 'GMLVQ' in the console and trying the autocomplete by hitting `[Tab]`. It should automatically complete the name.

Secondly, you'll need a properly formatted and labelled data set. Specifically, you'll need two variables `fvec` and `lbl`. `fvec` should be an array of size `nFv x nDim`, containing the set of `nFv` feature vectors in `nDim` dimensions. `lbl` should be a corresponding vector of `nFv` class labels. These labels must cover the complete set 1-`nFv`. As an example, a number of sample data sets are provided in the archive, those will also be used in the demonstrations later on.

All functionality of the library is put inside of the `GMLVQ`-namespace. There are a number of classes, each class with a number of functions, to assist you in using the library intuitively. This means that each classname should be prepended with `GMLVQ.` in the code.

2 Examples

Most functionality of the library will be covered in the following examples, they should give you an idea of how the library should be used and what results you can obtain with it. For more detailed information on the API, later on there will be a complete reference.

2.1 Initialization

To be able to run the algorithm, you will first need to obtain an instance of the `GMLVQ.GMLVQ`-class. This class stores your data set, as well as the parameters you have chosen for the algorithm. Using this class, it is possible to run various tests on the data set and obtain results. This class can be instantiated using the following constructor:

```
GMLVQ.GMLVQ(fvec, lbl, parameters, totalsteps, prototypeLabels)
```

Here, `fvec` and `lbl` are the variables discussed above. `parameters` must be an instance of `GMLVQ.Parameters`, where you can set various parameters for the algorithm. `totalsteps` is the number of time steps to take while running the algorithm (by default: 10), and lastly `prototypeLabels` is a vector specifying how many prototypes, and for which classes to use. By default this is simply one prototype per class, but by setting this variable it is possible to use multiple prototypes per class.

As mentioned above, the `GMLVQ.Parameters`-class contains all parameters to use in the algorithm. This class can be instantiated using a name-value style call of the constructor, for example as follows:

```
GMLVQ.Parameters('paramName', value, 'paramName2', value2, ...)
```

The following parameters can be set in this class, including their type and default (recommended) values. More context on some of these options will be given in later examples.

- `mode = GMLVQ.Mode.GMLVQNS` (`GMLVQ.Mode` enum): The mode to use for the algorithm. Put in one of the elements from the enum `GMLVQ.Mode`:
  - `GMLVQ.Mode.GMLVQ`: Matrix without null-space correction
  - `GMLVQ.Mode.GMLVQNS`: Matrix with null-space correction

  - `GMLVQ.Mode.GRLVQ`: Diagonal matrix (discouraged)
  - `GMLVQ.Mode.GLVQ`: GLVQ with Euclidian distance
- `doztr = true` (boolean): Whether or not to do a z-score transformation on the training data.
- `rndinit = false` (boolean): When `true`, will randomize the initialization of the relevance matrix, instead of using the identity matrix. `false` is recommended for first experiments.
- `mu = 0` (real): A control parameter of the penalty term when lambda is singular. A value of 0 means default GMLVQ behaviour and is recommended for first experiments. Values larger than 0 prevent a singular lambda matrix, while values much larger than 0 let lambda approach identity, i.e. Euclidian behaviour.
- `decfac = 1.5` (real): Step size factor (decrease) for Papari steps
- `incfac = 1.1` (real): Step size factor (increase) for all steps
- `ncop = 5` (integer): Number of waypoints stored and averaged during the algorithm calculation.
- `etam` (real): Stepsize adaptation parameter. Default value 2 for GMLVQ or GMLVQNS modes, 0.2 for GRLVQ and 0 for GLVQ.
- `etap` (real): Stepsize adaptation parameter. Default value 1 for GMLVQ or GMLVQNS modes, 0.1 for GRLVQ and 1 for GLVQ.
- `rngseed = 291024` (integer): The seed to use for the random number generator to allow for reproducible results.
- `rocClass = 1` (integer): The label of the class to compute the ROC against.
- `showlegend = true` (boolean): Whether or not to show legends in the graphs produced upon plotting.
- `randomization = 0.02` (real): A measure for how 'spread out' the initial prototype locations are for execution. Will not affect results when `useKMeans` is `true`.
- `useKMeans = true` (boolean): Whether or not to use k-means to initialize the prototypes.

So combining this, if I want to run the algorithm without the k-means initialization, running for 50 steps, using two prototypes for the second class, I could call the following in the console:

```
gmlvq = GMLVQ.GMLVQ(fvec, lbl, GMLVQ.Parameters('useKMeans', false), 50, [1 2 2
↪   3 4]);
```

2.2 Single runs of GMLVQ

In this section we will only be using the `runSingle()` function of the `GMLVQ`-class. This function performs a single training process for the entire data set. The result is an instance of the `Result`-class, which includes a bunch of performance statistics, such as the cost function, error rates and AUROC as functions of time(steps) and the final LVQ system.

Using the `plot(...)` function you can plot the training curves, the final LVQ system is visualized in terms of prototype vectors and the relevance matrix, and the ROC of the final system is displayed. The data set is also visualized in terms of the leading two eigenvectors of the relevance matrix.

2.2.1 Demo I: A seven-class data set, single run with one prototype per class

This is the UCI segmentation data set, where 1 trivial dimension was removed and 800 samples were randomly selected. From the results, it is clear that classes 2 and 7 are easily distinguishable, while classes 3 through 6 lie very close together. This can also be seen in the class-wise training errors. In the resulting figure, the ROC (w.r.t. class 1) is not shown.

Console session

```
>> load uci-segmentation-sampled.mat
>> gmlvq = GMLVQ.GMLVQ(fvec, lbl, GMLVQ.Parameters(), 50);
Matrix relevances with null-space correction
Defaulted to one prototype per class
Prototype configuration:
     1      2      3      4      5      6      7
Warning: Multi-class problem. ROC analysis is for class 1 (neg.) vs. all others (pos.)
Minimum standard deviation of features: 0.025258
>> result = gmlvq.runSingle();
>> result.plot();
```

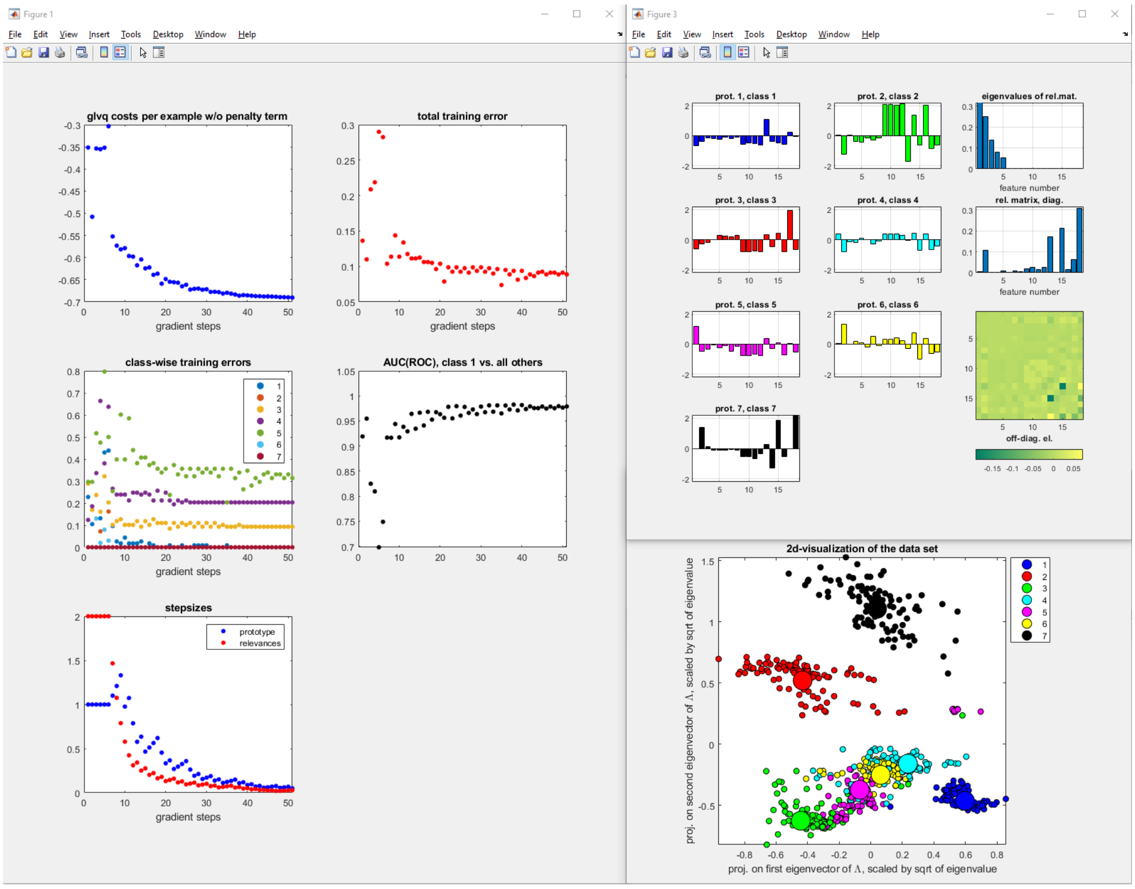


Figure 1
File Edit View Insert Tools Desktop Window Help
glvq costs per example w/o penalty term
gradient steps
total training error
gradient steps
class-wise training errors
1
2
3
4
5
6
7
gradient steps
AUC(ROC), class 1 vs. all others
stepsizes
prototype
relevances
gradient steps
Figure 3
File Edit View Insert Tools Desktop Window Help
prot. 1, class 1
prot. 2, class 2
eigenvalues of rel.mat.
feature number
prot. 3, class 3
prot. 4, class 4
rel. matrix, diag.
feature number
prot. 5, class 5
prot. 6, class 6
off-diag. el.
prot. 7, class 7
2d-visualization of the data set
proj. on second eigenvector of Λ, scaled by sqrt of eigenvalue
proj. on first eigenvector of Λ, scaled by sqrt of eigenvalue
1
2
3
4
5
6
7

2.2.2 Demo II: A simple two-class problem, single run with one prototype per class, ROC against class 1

This data set has 186 features for 110 samples, but yields a very clear separation between both its classes.

Console session

```
>> load twoclass-simple.mat
>> gmlvq = GMLVQ.GMLVQ(fvec, lbl, GMLVQ.Parameters('rocClass', 1), 30);
Matrix relevances with null-space correction
Defaulted to one prototype per class
Prototype configuration:
     1     2
Minimum standard deviation of features: 0.0090926
>> result = gmlvq.runSingle();
>> plot(result); // Both calling styles work
```

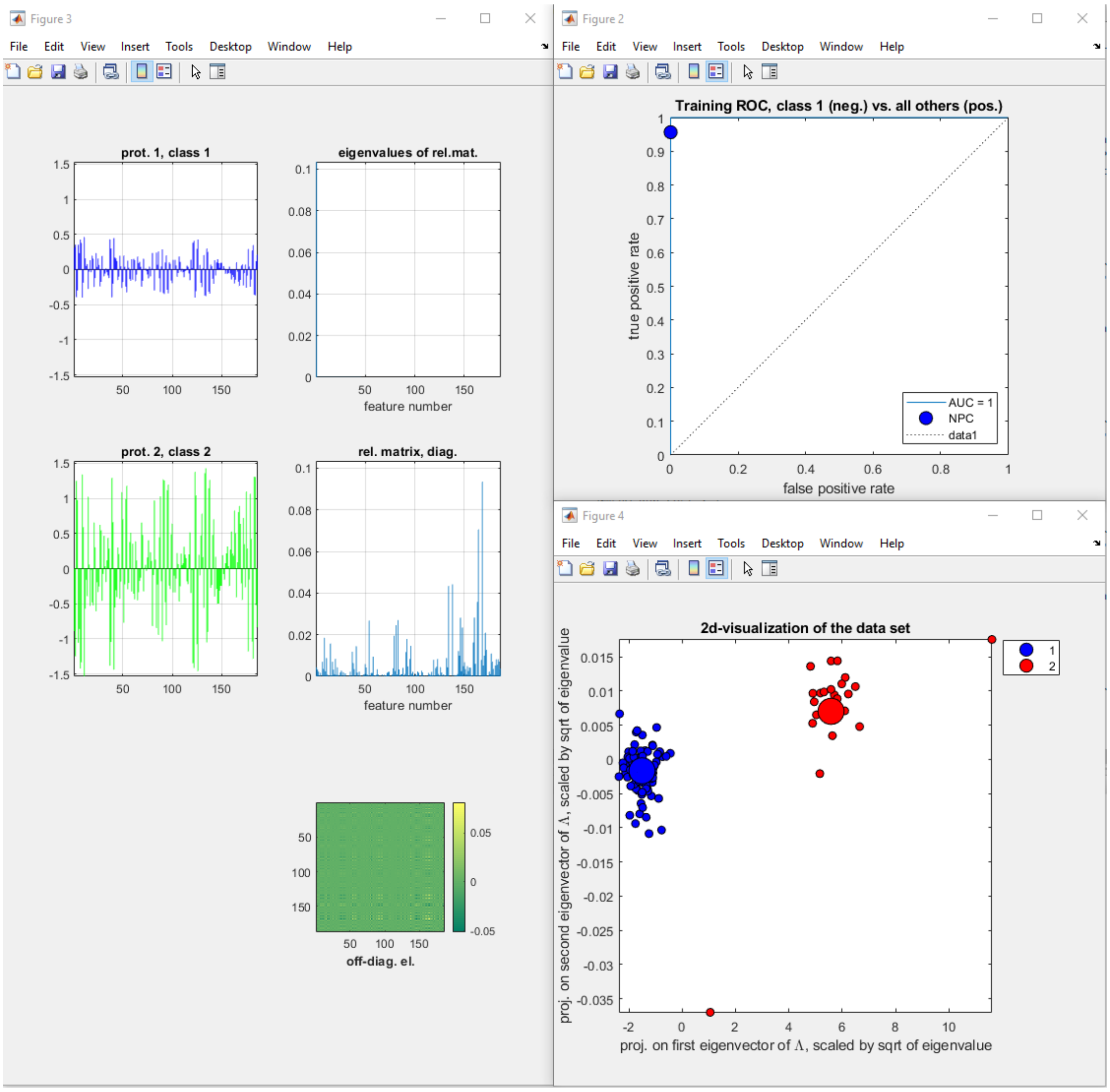


Figure 3
File Edit View Insert Tools Desktop Window Help
prot. 1, class 1
eigenvalues of rel.mat.
feature number
prot. 2, class 2
rel. matrix, diag.
feature number
off-diag. el.
Figure 2
File Edit View Insert Tools Desktop Window Help
Training ROC, class 1 (neg.) vs. all others (pos.)
true positive rate
false positive rate
AUC = 1
NPC
data1
Figure 4
File Edit View Insert Tools Desktop Window Help
2d-visualization of the data set
proj. on second eigenvector of Λ, scaled by sqrt of eigenvalue
proj. on first eigenvector of Λ, scaled by sqrt of eigenvalue
1
2

2.2.3 Demo III: A difficult two-class problem, single run with custom prototypes

This data set has 32 features and 98 samples, but does not have a very clear separation.

Console session

```
>> load twoclass-difficult.mat
>> gmlvq = GMLVQ.GMLVQ(fvec, lbl, GMLVQ.Parameters(), 50, [1 1 2]);
Matrix relevances with null-space correction
Prototype configuration:
     1     1     2
Minimum standard deviation of features: 0.56657
>> result = gmlvq.runSingle();
>> plot(result);
```

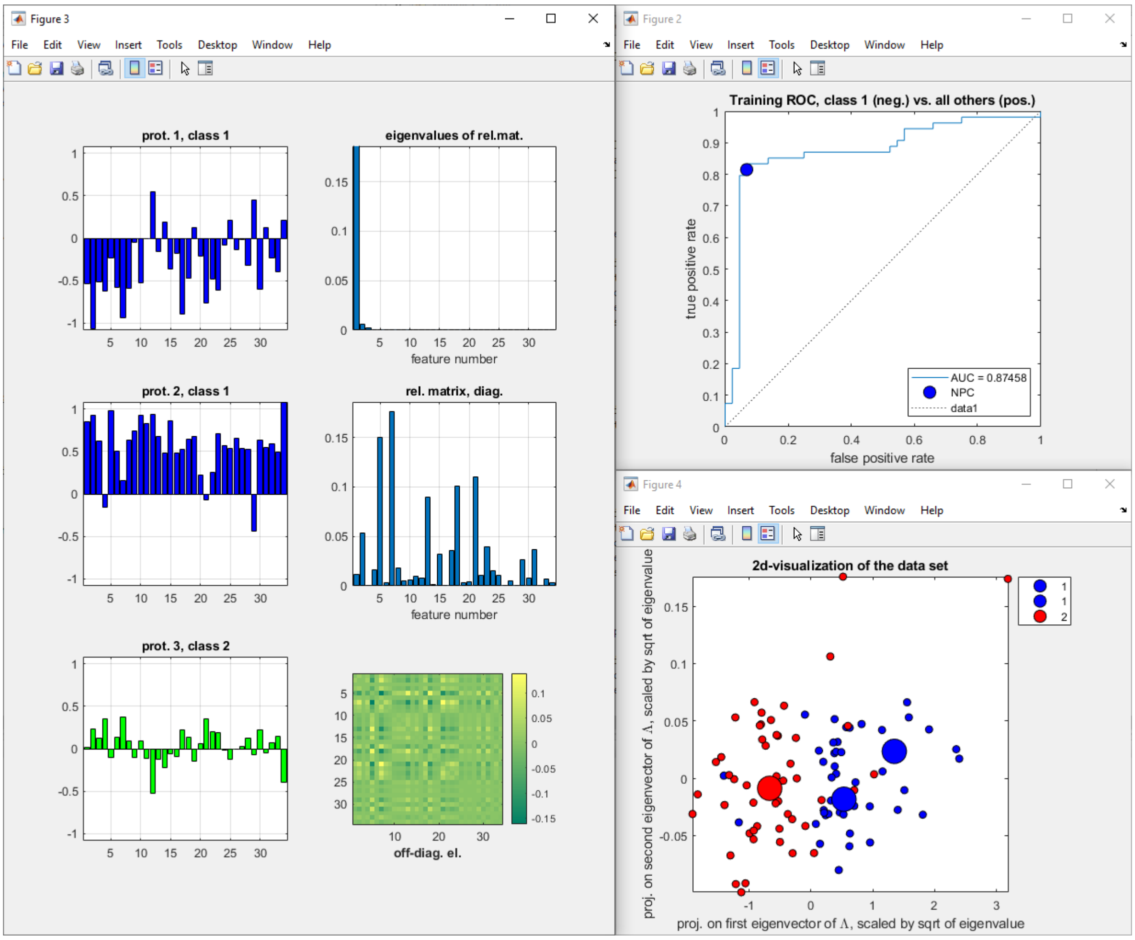


Figure 3
File Edit View Insert Tools Desktop Window Help
prot. 1, class 1
eigenvalues of rel.mat.
feature number
prot. 2, class 1
rel. matrix, diag.
feature number
prot. 3, class 2
off-diag. el.
Figure 2
Training ROC, class 1 (neg.) vs. all others (pos.)
true positive rate
false positive rate
AUC = 0.87458
NPC
data1
Figure 4
2d-visualization of the data set
proj. on second eigenvector of Λ, scaled by sqrt of eigenvalue
proj. on first eigenvector of Λ, scaled by sqrt of eigenvalue
1
1
2

We can see that the eigenvalues are very close to singularity. To prevent this, we can tune the `mu` parameter.

Continued console session

```
>> gmlvq.params.mu = 0.2;
>> result = gmlvq.runSingle();
>> plot(result);
```

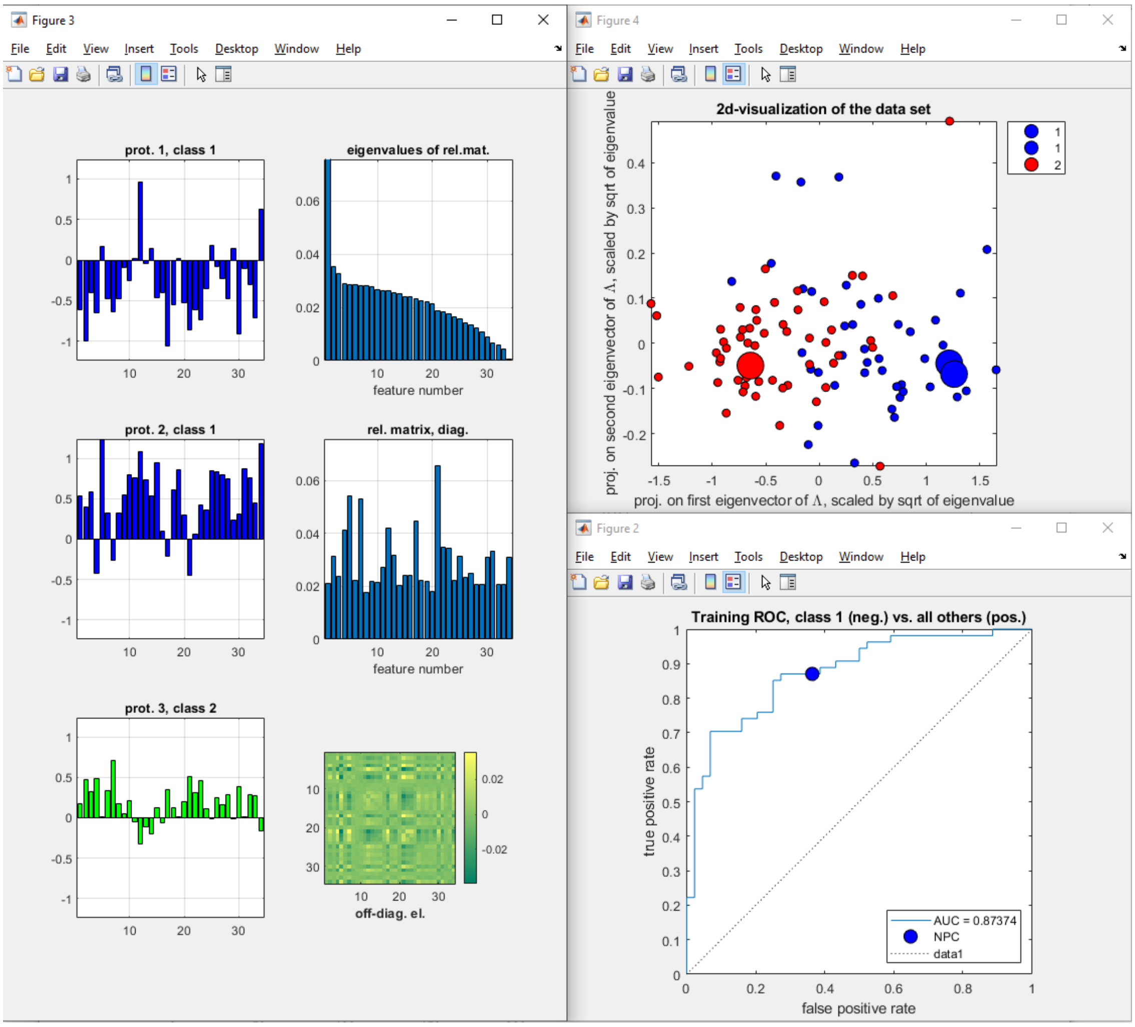


Figure 3
File Edit View Insert Tools Desktop Window Help
prot. 1, class 1
eigenvalues of rel.mat.
feature number
prot. 2, class 1
rel. matrix, diag.
feature number
prot. 3, class 2
off-diag. el.
Figure 4
2d-visualization of the data set
1
1
2
proj. on first eigenvector of Λ, scaled by sqrt of eigenvalue
proj. on second eigenvector of Λ, scaled by sqrt of eigenvalue
Figure 2
Training ROC, class 1 (neg.) vs. all others (pos.)
AUC = 0.87374
NPC
data1
false positive rate
true positive rate

2.3 Multiple runs of GMLVQ

In this section we will be discussion the `runValidation(...)` and `runL1O()` function of the `GMLVQ`-class. These functions perform multiple training processes on a part of the data, while doing validation classification on the remaining data. They return an (specialized) instance of the `ResultSet`-class, which includes the `Result` instances for each run, as well as the average result where all properties are averaged. Again, using the `plot(...)` function you can plot the entire resultset, or one of the subresults.

The `runValidation(nRuns, percentage)` function runs `n` times with `percentage` of the data left out for validation. For each run, it makes a stratified partition of the data, i.e. the relative frequencies of the various classes will be kept constant and it will not happen that one class is completely omitted in one of the runs. In other words: `percentage` of the samples *per class* is left out.

The `runL1O()` function runs `nSamples` times, each time leaving out one sample for validation. L1O can seriously mis-estimate performance, and should really only be used for very small data sets.

Performance

Performing multiple runs (especially L1O) can be really slow. Both the `runValidation(...)` and `runL1O()` functions parallelize the execution by performing each separate run in parallel. This is only possible if you have the Parallel Toolbox installed and activated, otherwise it will default to serial behaviour. Note that this also causes the output of the function to be out-of-order.

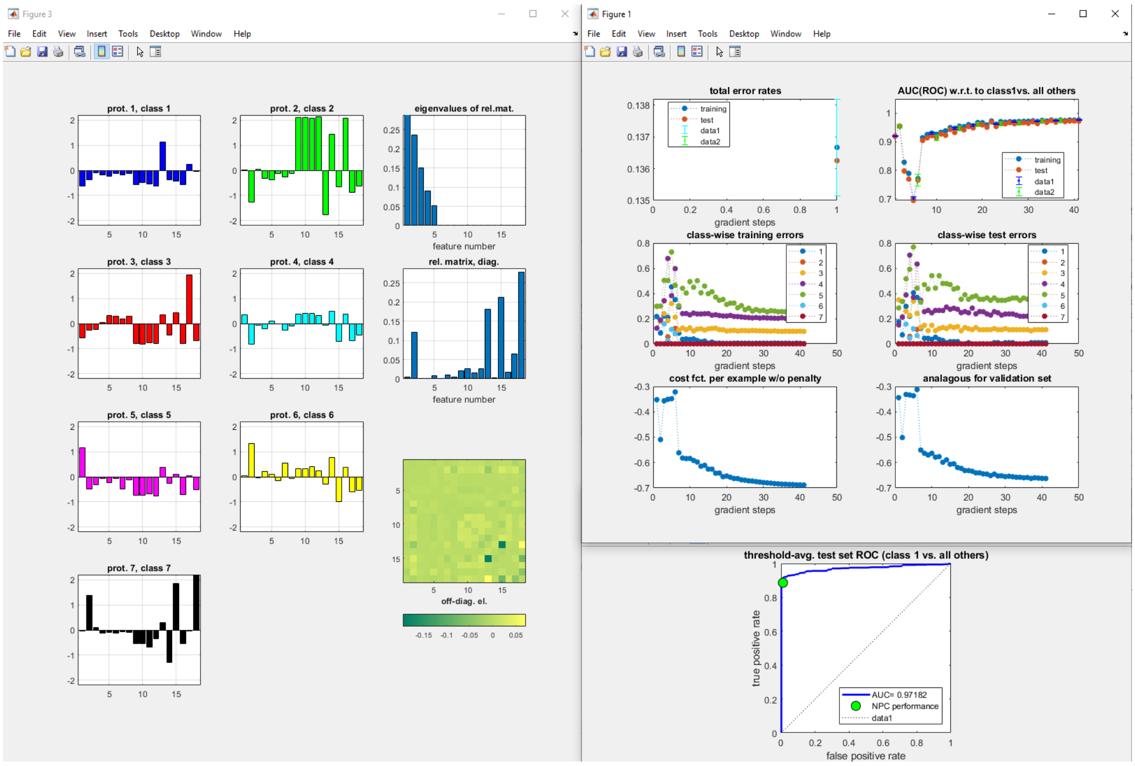
2.3.1 Demo IV: A seven-class data set, multiple runs with 10% of data left out

This is the same data as in Demo I, so we have 800 samples. We see that the output of `runValidation(...)` does not show ordered properly, this is because the code is run in parallel. Even when run in serial, it might show up out-of-order due to underlying code differences.

Console session

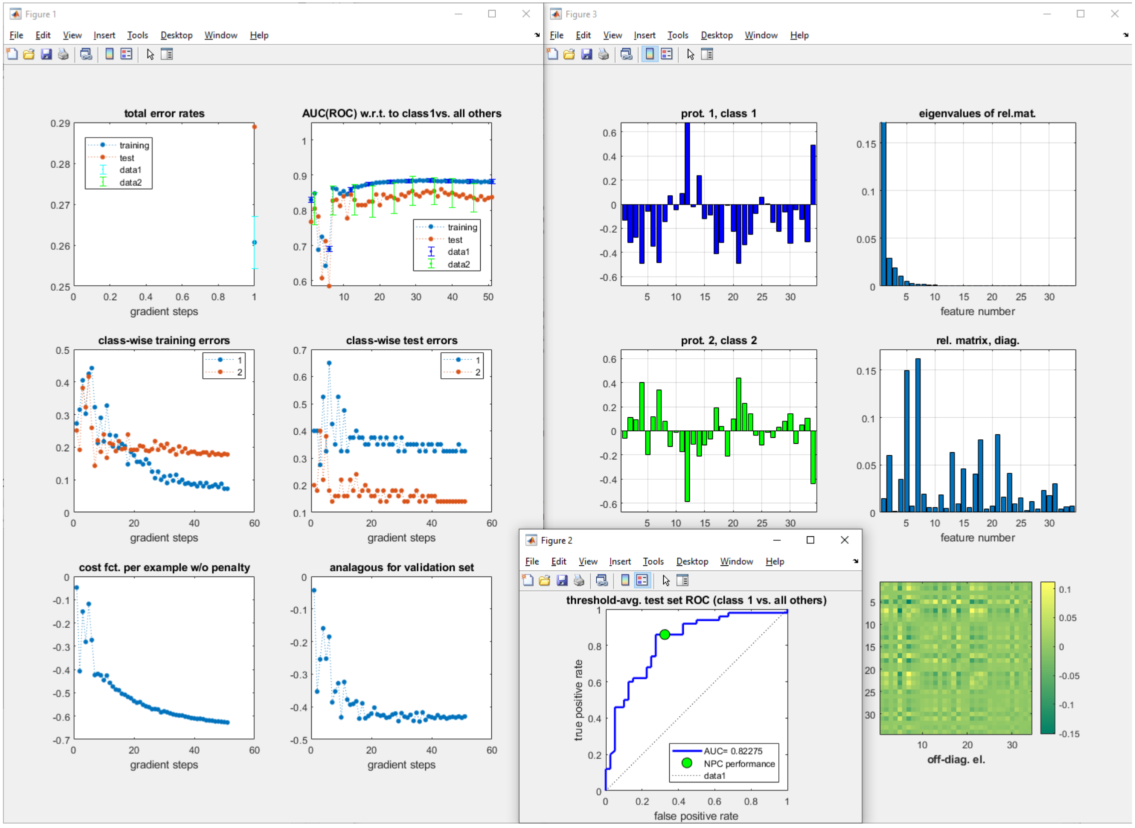
```
>> load uci-segmentation-sampled.mat
>> gmlvq = GMLVQ.GMLVQ(fvec, lbl, GMLVQ.Parameters(), 40);
Matrix relevances with null-space correction
Defaulted to one prototype per class
Prototype configuration:
     1     2     3     4     5     6     7
Warning: Multi-class problem. ROC analysis is for class 1 (neg.) vs. all others (pos.)
Minimum standard deviation of features: 0.025258

>> result = gmlvq.runValidation(10, 10);
Learning curves, averages over 10 validation runs with 10% of examples per class left out for
↪ testing
Validation run 2 of 10
Validation run 4 of 10
Validation run 3 of 10
Validation run 6 of 10
Validation run 5 of 10
Validation run 1 of 10
Validation run 8 of 10
Validation run 9 of 10
Validation run 7 of 10
Validation run 10 of 10
>> plot(result);
```



2.3.2 Demo V: A difficult two-class problem, validation with 10% left out

This is the same data as in Demo III, where the classification was not excellent. We will now run this 10 times with each time 10% of the data left out. The code is similar to that of Demo IV.



We can see that the validation performance is bad compared to the training performance (which we saw in Demo III), which hints at overfitting the data.

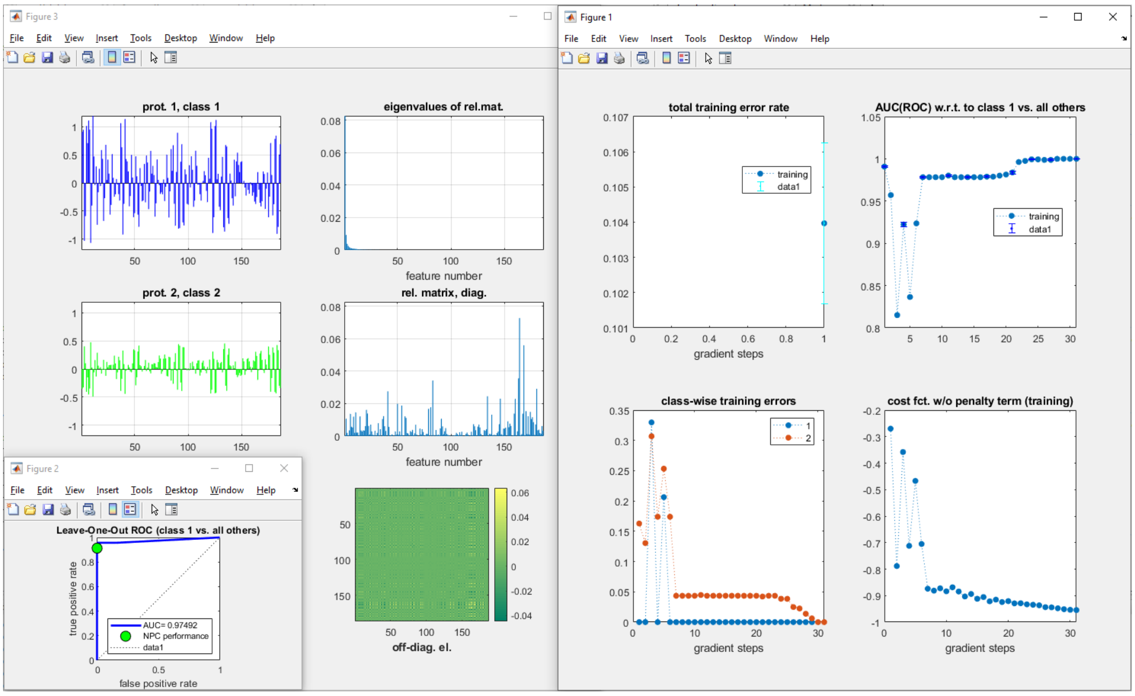
2.3.3 Demo VI: A small subset of a simple two-class problem, Leave-1-Out run

In this case we only have a small subset of the data from Demo II, only 36 samples for 186 features. The best we can do here is to run L1O as last resort.

Console session

```
>> load twoclass-simple-small.mat;
>> gmlvq = GMLVQ.GMLVQ(fvec, lbl, GMLVQ.Parameters(), 30);
Matrix relevances with null-space correction
Defaulted to one prototype per class
Prototype configuration:
     1     2
Minimum standard deviation of features: 0.0089232

>> result = gmlvq.runL1O();
Learning curves, averages over 36 L1O runs
Leave one out: 1 of 36
Leave one out: 8 of 36
....
Leave one out: 28 of 36
Leave one out: 35 of 36
>> plot(result);
```



3 Changelog

Main changes of version 3.0 (March 2019)

- Major API overhaul into an object-oriented design, allowing for re-use of parameters/run setups easily.
- Possibility to store results to (re-)plot them later.
- Parallelized the validation- and L1O-runs if the Parallel Toolbox is present.
- Added possibility to calculate ROC vs. any class instead of just the class with label '1'.
- Using `cvpartition` in Matlab to obtain stratified training data for validation runs.
- Added k-means prototype initialization instead of the previously used average of all points in the class.
- Added a configurable option for the 'randomness' of the initial prototypes (when not using k-means).
- General performance improvements and code quality improvements

Main changes of version 2.3 (January 2017)

- Bug fixed in the averaging of prototypes over validation runs. Previous version yielded averaged prototypes that were stretched by a factor of 2, approximately. The error did not affect the training or validation itself, only the averaged prototypes in the final output.

Main changes of version 2.2 (April 2016)

- ROC calculation has been modified, it is now based on differences $d(x, w_1) - d(x, w_2)$ without normalization by $d(x, w_1) + d(x, w - 2)$. See `compute_roc.m` for comments and details.

Main changes of version 2.1 (August 2015)

- Corrected legends in plots showing step size vs learning time
- Corrected calculation of mean confusion matrix in `run_validation(...)`. Now the confusion matrix is determined in each validation run separately (in terms of percentages) and then averaged over validation runs in the end. Resulting matrix is now in line with the averaged class-wise errors.

Main changes of version 2 (April 2015)

- Less explicit input parameters to functions (moved to function `set_parameters(...)`)
- Improved step size control, really independent for matrix and prototype updates
- Singularity control via penalty term included, parameter `mu` introduces
- `single_run(...)` also displays temporal evolution of step sizes
- Null-space correction controlled in `set_parameters(...)` (no dimension-dependent default anymore)
- Initial step sizes independent of dimension and/or number of examples