A Nelder-Mead Tuner for Svm

prepared by:	Kester Smith
approved by:	
reference:	GAIA-C8-TN-MPIA-KS-016
issue:	1
revision:	0
date:	2009-03-13
status:	Draft

Abstract

Read at your own risk, as this is a working document and has not yet been checked!

Document History

Issue	Revision	Date	Author	Comment
D	0	2009-02-18	KS	Document created

Contents

1 Introduction

Many of the classifiers and parameterizers in use by the MPIA group are based on Support Vector Machines (SVM)[1]. This algorithm is used in three different versions: for multiclass classification, parameterization and in one-class mode to detect outliers. Each version requires a number of parameters to be set. The classification algorithm requires a 'penalty term', C, which governs the amount of regularization of the model, the one-class version requires a similar parameter, nu, which basically governs the fraction of a training set allowed to be classed as outlying. The parameterization version takes a value, epsilon, which specifies the width of the regression 'hypertube' within which points are considered well fitted and therefore ignored (the regression takes as support vectors only points which lie beyond a certain tolerance threshold from the regression line, just as the classifier considers only points on the maximum margin of the hyperplane). In addition, all versions used by MPIA currently take the Radial Basis Function Kernel (RBF), which requires the setting of a kernel parameter, gamma, which sets the scale of the kernel function ($\exp -\gamma |||x_i - x_j|||^2$).

The performance of the Svm is at times critically dependent on the choice of these parameters, and finding the optimum combination of values can be time consuming. The standard way of doing this is to scan the parameter plane, or, for the parameterizer, cube, in steps of factors of two, and then to home-in on the optimum value with a finer search. This method can be tiresome because it typically requires a level of human supervision, making it unsuitable for repeated automated tuning of many models. Simple automation algorithms could be implemented, and some are available in the R package used by the group for testing. For the Java implementation, the tuning algorithm had to be written from scratch. We therefore decided to implement a Nelder Mead (or Amoeba-simplex) type algorithm, instead of the straightforward grid search.

This document acts as a combined software design document, user manual, and testing description for the Nelder Mead tuner. It concludes with a critique of the currently implemented version and some suggestions for future improvement.

2 Outline of the algorithm

The Nelder Mead algorithm is one of the most well known and widely used algorithms for optimization. A good description is available in the Numerical Recipes (Press et al.) and one can also search online in Wikipedia or simply with a Google search to obtain a description, if the following overview is not sufficient.

The algorithm at first initializes a simplex in the parameter search space. For N parameters, this simplex has $N + 1$ elements. An initial set of parameters is chosen, and this choice is used

[1]We use the libSvm implementation with a Java wrapper, See Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm

as one of the elements. The other N elements are created by offsetting each of the parameters by a certain amount (typically 50% of their value, but the choice is entirely at the discretion of the user). This is easy to draw in 2d, and in fact the classification Svm has two parameters, so for this case the illustration is not even a simplification. Having set up the simplex, we evaluate

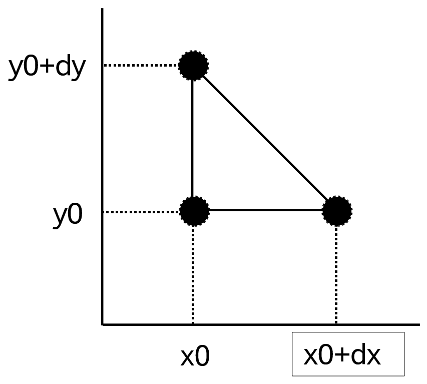

FIGURE 1: The initialisation of the simplex for two parameters, x and y. The simplex has vertices (elements) at (x, y) and at points displaced from these along lines parallel to the axes, $(x + \mathrm{d}x, y)$ and $(x, y + \mathrm{d}y)$.

the model for each element and find the element with the worst performance. This element is eliminated and replaced with its reflection in the centroid of the other elements. (Figure 2). This is the basic move of the algorithm. In this way, the simplex will eventually 'walk' towards a

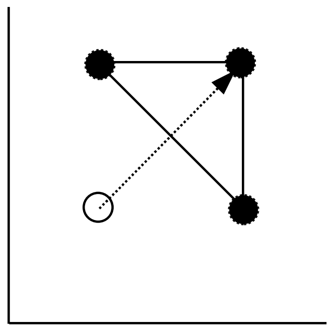

FIGURE 2: The worst point of the original simplex, here shown as an open circle, is reflected in the centroid of the other points. $(x_{worst}, y_{worst}) \rightarrow (x_{best} + x_{mid} - x_{worst}, y_{best} + y_{mid} - y_{worst})$ The new point is shown as a filled symbol. Note that the axes of this plot represent the values of the parameters, and not the quality of the models.

minimum or maximum (hence the name 'amoeba', sometimes applied it). A few modifications to this basic idea are however either necessary or desirable. Firstly, it can be easily seen that there is a problem if the new point created during the step is still the worst point. In this case, it is simply reflected back again and the algorithm enters and infinite loop. To prevent this, a second move is applied, that we refer to here as 'contraction'. It works as follows; the newly reflected point is evaluated and, if it is determined that it is still the worst point, a new point is created by shortening the reflection vector (Figure 3). We now have three candidate points, the

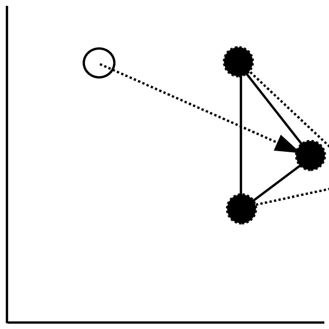
FIGURE 3: If the new point is still the worst, a 'contracted' reflection is attempted.

original worst point, the reflected point, and the new point that is the result of the 'contracted' reflection. We choose the best of these three as the new element. If this 'best-of-the-worst' element is either the original point or the reflected point, we perform another move, which we call here 'shrinking' (Figure 4). This shrinking helps prevent the infinite loop occurring. It

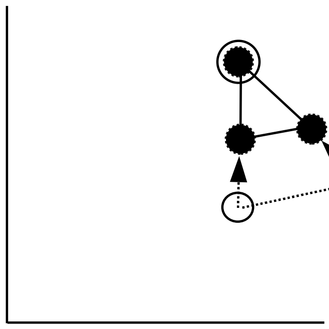
FIGURE 4: If the orignal point, or the reflected point, but NOT the 'contracted' point is retained, the simplex is shrunk towards the best point, as illustrated.

also allows the algorithm to reduce the scale of the search. Some sort of shrinking is anyway necessary if the algorithm is to eventually home in on the solution, and in the situation occuring here, where the old point, its reflection and the contracted reflection are all worse than the two other points, it seems that two other points probably lie on a ridge along which the solution might well also lie, perhaps approximately between them. Shrinking at this stage is therefore a reasonable move.

Finally, if the new point generated by the reflection is in fact the new best point, better than the other original points, we perform a move called an 'expansion'. This is designed to accelerate the downward (or upward) movement of the simplex in the event that we have found a direction of steep descent (ascent). (see Figure 5)

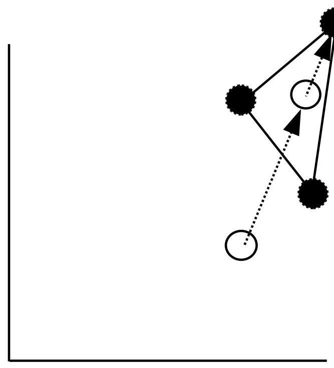

FIGURE 5: If the newly reflected point is the best, we expand the reflection in a search for an even better point.

3 Application to the problem of Svm tuning for Gaia algorithms

For the classification, there are two parameters to tune, the cost parameter C and the Kernel parameter gamma. For the parameterization, we have to add the epsilon parameter. The one class Svm also has two parameters, nu (which is related to the multiclass C) and the Kernel parameter gamma. The one class Svm is not currently tuned, however, and it is not clear how this might be approached (the problem is defining a definitive set of outliers for a test). The approaches for classification and parameterization are essentially the same apart from the extra parameter in the case of parameterization.

3.1 Minimum bounds

One particular problem with the Svm tuning is that the parameters all have lower bounds. For the Kernel parameter gamma, and epsilon in the case of parameterization, the values cannot go to zero or below. The case of C is similar, except that the libSvm implementetion we use takes C as an integer, so we need to prevent the value of C falling below 1 (the tuner deals with C as a double for computational reasons). We deal with this bound by introducing a floor below the parameters. This floor is set to be half of the lowest value for that parameter yet tested in the simplex. If the value of a parameter falls below the lower bound, it is reset to the value of the floor. This allows the algorithm to explore ever lower values without breaking the bound. There is some difficulty when dealing with C, due to the double-to-integer conversion we could end up dealing with values of C all effectively the same due to the integer rounding. In fact though this would lead to identical models (at least in the C parameter) and so we would then expect convergence.

4 Convergence

We consider the algorithm to have converged when a certain minimum spread between the best and worst models is acheived.

5 Code description

The code consists of two Java classes, Tuner and TuningSimplex. The Tuner sets up the training set and the initial parameters and controls the moves made according to the algorithm. The TuningSimplex class stores the values of the parameters and trains and evaluates the models.

6 Controlling the algorithm

For Dsc, the algorithm can be run as part of the training mode. It is controlled by setting properties keys in the dsc_training.properties file. These are as follows;

```
gaia.cu8.dsc.doTuning=false
```

This turns the tuner on and off.

```
gaia.cu8.dsc.tuner.trainFrac=0.5
```

The tuner splits the training data into two parts and uses one for training models and the other for testing. This property selects how much of the training data is to be used as the rtraining set for the tuner. The actual set is selected randomly.

```
gaia.cu8.dsc.tuner.tunerRandomSeed=false
```

The random number generator used to select training sources can be set to be 100 (tunerRandomSeed=true) for reproducability.

```
gaia.cu8.dsc.tuner.startVarC=1.5
```

This selects the initial variation in the C parameter.

```
gaia.cu8.dsc.tuner.startVarGamma=1.5
```

The initial offset to gamma

```
gaia.cu8.dsc.tuner.expandFact=1.8
```

This property controls the additional expansion made with the expand move, described previously. If the normal reflection in the centroid is thought of as a vector in parameter space, the expandFact is the factor by which this is multiplied to make the expanded reflection. The factor should obviously be greater than 1. and is recommended to be kept less than 2.

```
gaia.cu8.dsc.tuner.contractFact=0.8
```

The factor controlling the scale of the contraction move. If the reflection in the centroid is contracted, this is the factor by which the reflection vector is reduced. It should obviously be less than 1. and is recommended to be kept greater than 0.5.

```
gaia.cu8.dsc.tuner.shrinkFact=0.3
```

Thye scale of the shrink move. The two (or three) less good vertices of the simplex are shifted towards the best point. This factor is the fraction of the connecting vectors along which the points move. Must be less than 1. and greater tha nzro, is recommended to be kept less than 0.5.

```
gaia.cu8.dsc.tuner.convergeSpread=0.02
```

This is the spread between the best and worst performing models (in%) accepted for convergence.

```
gaia.cu8.dsc.tuner.maxSteps=50
```

This is the maximum number of iterations attempted before the optimization is abandoned.

The control of the tuner for regression is similar, with the addition of keys to control the epsilon parameter.

7 Some outstanding problems

7.1 Poor starting values

If started with very poor values for the parameters, the algorithm sometimes converges immediately (in the case of classification), since the results for each model tend to $1/N$, where N is the number of classes into which the objects are divided. This problem is easily avoided with some common sense in the choice of initial values. If the search does converge with success rate $1/N$ (for N-class classification) then it is clear that this is a poor solution and the process should be rerun.

7.2 Local maximum

There is always the danger that the algorithm will converge on a local maximum (miniumum) and miss a much better solution elsewhere in the parameter space. Safeguards against this include choosing large enough values of the parameters startVarC and startVarGamma (and startVarEpsilon if needed) that the initial simplex covers a wide area of parameter space, and choosing sufficiently large values of expandFact that the algorithm has a good chance to break out of a local maximum, and sufficiently small values of shrinkFact and contractFact that the algorithm does not too rapidly shrink the simplex towards what may be a local solution. Other than this, the main safeguard is to rerun the algorithm after convergence with a different set of starting conditions and check that it converges to a similar solution. This procedure could be repeated two or three times.

7.3 Overfitting

The algorithm at present splits the supplied data once into a training set and a testing set, and then repeatedly trains models with different choices of parameter. With this scheme, there is a danger that the algorithm will overfit the parameters to the particular choice of training and test objects, and that the solution will therefore no longer be optimal when the full training set is used to train a model for application to a new input data set. Ideally, the model evaluation would include a cross validation scheme to minimise the risk of this, but at present this is not implemented. The user can guard against this eventuality by making multiple runs of the algorithm with different random splits in the input data (tunerRandomSeed=false).

8 A basic test

For this test, the Dsc V5.0 was used, together with the dataset c3_m200_fortest2_4class_75pc_trn. The parameters for the tuner were as follows;

```
gaia.cu8.dsc.doTuning=true
gaia.cu8.dsc.tuner.trainFrac=0.2
gaia.cu8.dsc.tuner.tunerRandomSeed=false
gaia.cu8.dsc.tuner.startVarC=1.5
gaia.cu8.dsc.tuner.startVarGamma=1.5
gaia.cu8.dsc.tuner.expandFact=1.8
gaia.cu8.dsc.tuner.contractFact=0.8
gaia.cu8.dsc.tuner.shrinkFact=0.3
gaia.cu8.dsc.tuner.convergeSpread=0.02
gaia.cu8.dsc.tuner.maxSteps=50
```

The tuner was set up with four different initial values of C and gamma (Table 1). The results in terms of the final solution and the number of models computed are shown. This is illustrated graphically in Figure 6 which shows the progress of the tuner in each case from the starting position to the solution.

Initial parameters for tuner test

Initial C	Initial gamma	Number of models	Final C	Final gamma	Overall completeness
10	10^{-5}	51	1169	7.765×10^{-4}	69.33%
10	0.4	52	29	5.415×10^{-3}	70.75%
100	0.2	66	373	9.675×10^{-4}	69.56%
2759	0.0486	72	457	8.397×10^{-4}	70.53%

TABLE 1: Tests of the tuner for the DSC for four different sets of starting values. The total number of models computed on the way to the solution is indicated (that is, all the models ever included in the simplex, plus any trial contractions/expansions that were rejected). The composition of the training sample for each test case is randomly selected from the input objects and differs between the tests.

The solutions for the cases $(C, gamma)_{init} = (2759, 0.0486)$ and $(C, gamma)_{init} = (100, 0.2)$ are similar. The solution for the case $(C, gamma)_{init} = (10, 10^{-5})$ has a larger value of C. The solution for $(C, gamma)_{init} = (10, 0.4)$ finds a solution at lower C. (C=29). All the solutions however converge around a completeness of $\sim$ 70%. Two explanations are possible, either the model performance is very insensitive to the precise choice of parameters, or the tuner is overfitting based on the choice of the training objects. In fact, the completeness of the models around the region of all the solutions is similar, around 70% or a little less, and so the former explanation is perhaps more likely than the latter. Nevertheless, this test should be conducted again with the same training set in each case, to exclude the possibility that the differences are due to overfitting for particular choices of training objects. .

For comparison, we calculated a grid of models with regularly varying values of C and gamma across the plane searched by Nelder Mead, and we reproduce the results in Figure 7. These models were all calculated with the same split between training and testing objects (20% of hte

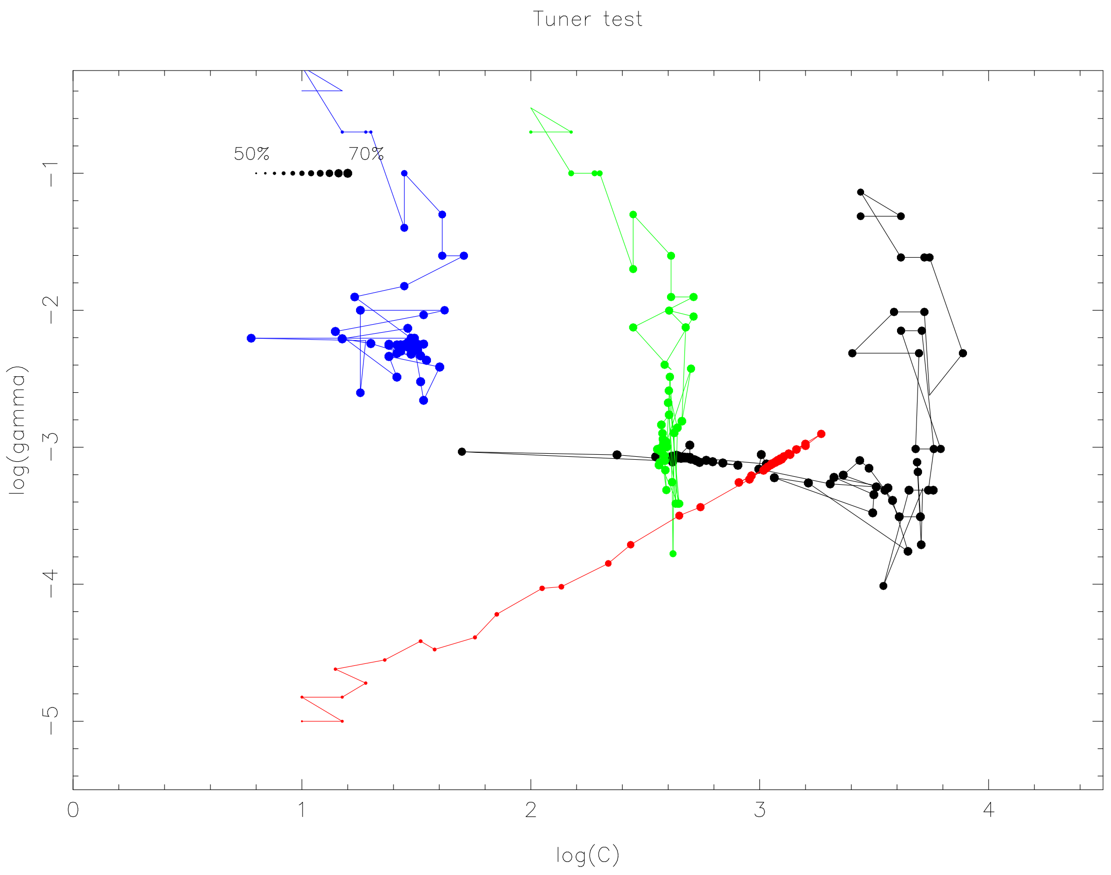

FIGURE 6: The progress of the four test cases from Table 1 towards a solution. The lines link the parameter choices in the order they were computed. The points themselves are scaled in size according to the completeness achieved by the model, with approximately 45% being the worst performance and approximately 70% being the best (see scale). It can be seen that the model starting at (2759,0.048) consistently produces results around the 70% level, whereas the other two starting points are initially very much less successful. All the points in the broad region around the three solutions have completeness of 65% or better, suggesting that the performance is a relatively insensitive function of the parameters in this region.

input objects were used for training). The same objects were used for every model. It can be seen that the optimum models form a ridge along the middle of the plot over a broad range of values of C. In Figure 8 we overplot both the Nelder Mead search and the plane grid search for comparison. It can be seen that the Nelder Mead search finds the optimum ridge quite successfully in all the test cases, but does not really search along the ridge to optimize along it.

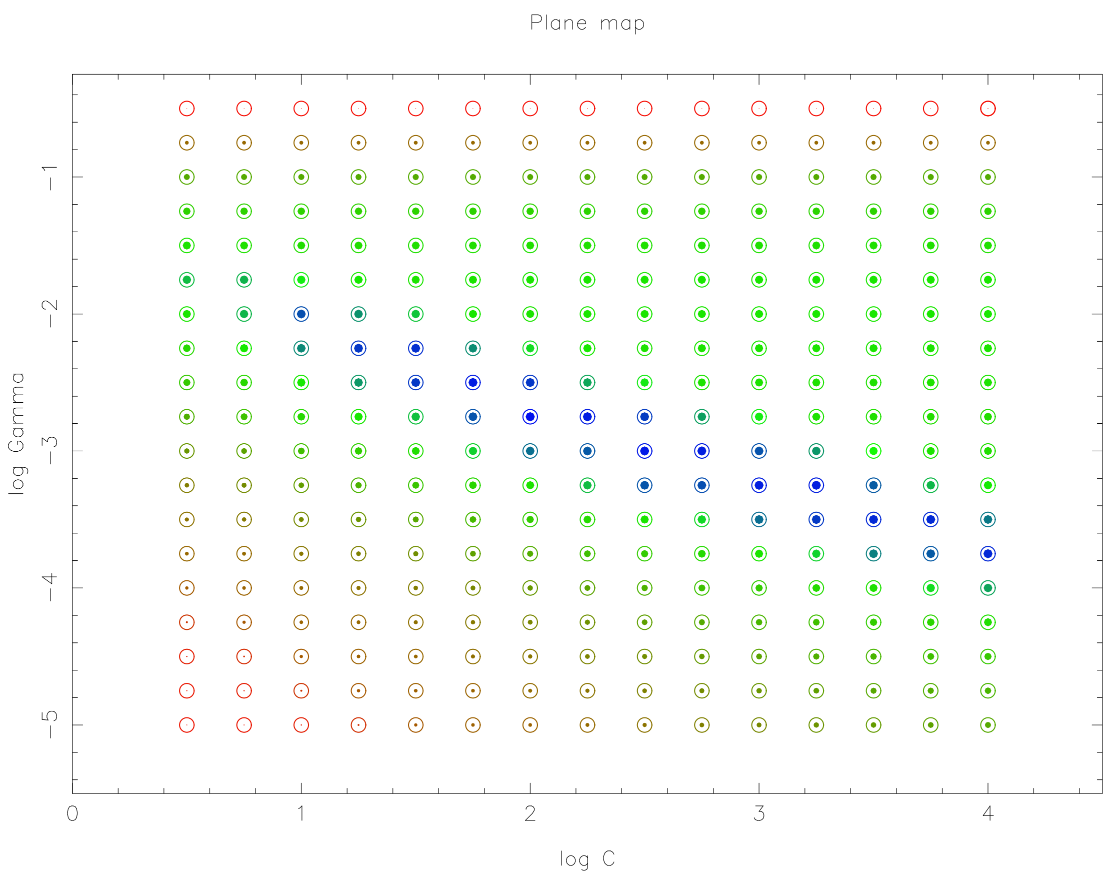

FIGURE 7: This plot shows the correct classification rate for models trained with different values of C and gamma across the same plane searched by the Nelder Mead tests shown in Figure 6. Each point is plotted twice, once with an open symbol of fixed size, and once with a filled symbol with a size proportional to the completeness acheived, with the same scale as in Figure 6. The points are also colour coded according to the success of each model, with the amount of green in the RGB colour proportional to the success rate between 46% and 68% and the amount of red decreasing in proportion (so red points are the worst and green the best). A second scale kicks in at 68%, where we introduce blue in proportion to the success rate between 68% and 70% and reduce green proportionally . The very best points (68% to 70%) therefore appear in blue. A ridge of good solutions extends across the middle of the plane.

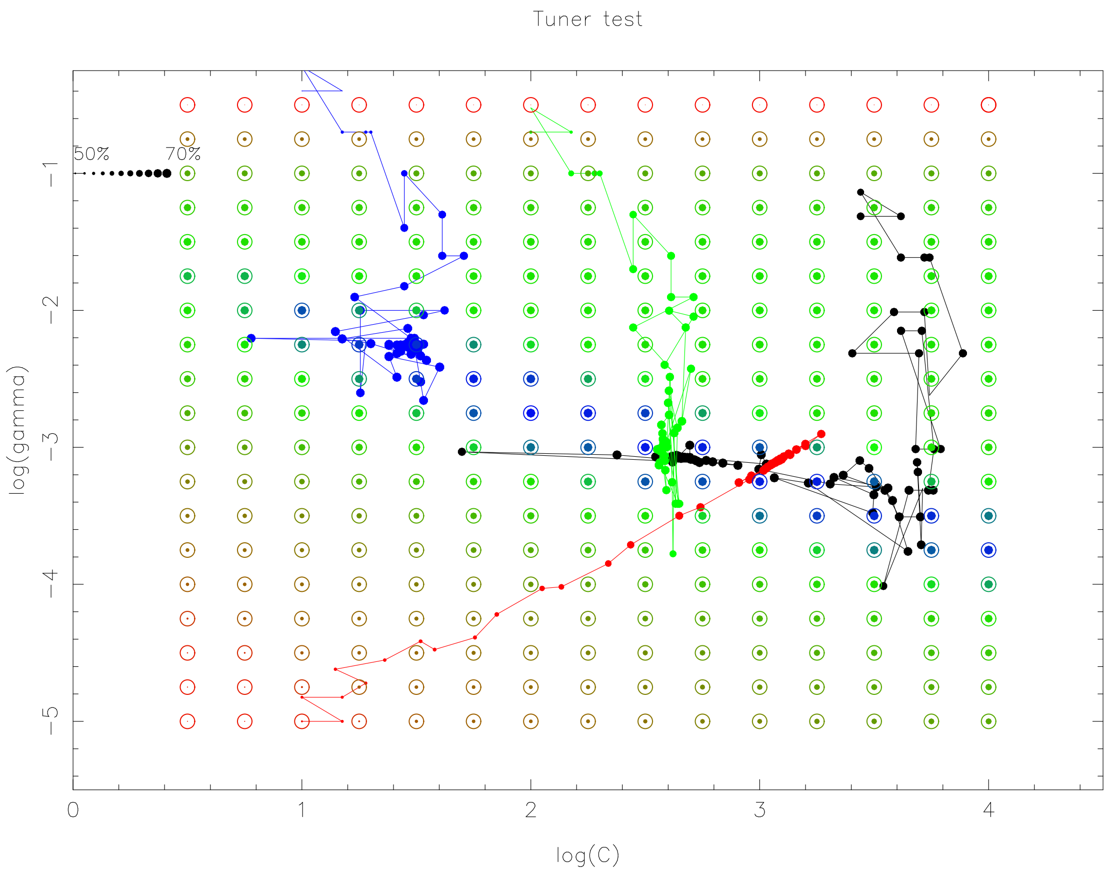

FIGURE 8: This plot essentially shows a superposition of the data in Figures 6 and 7. It can be seen that the Nelder Mead tuner in each case finds the optimum ridge of blue points extending across the middle of the plane. With the tested parameters, it does not progress further to optimize within the blue ridge. Restarting the tuner from each finishing point did not substantially change the result in any case.

9 Future development

I intend to carry out some tests with a search in log values, to see if the algorithm in fact converges faster. It may also be desirable to experiment with shrinking/contracting/expanding rates that vary depending on the parameter concerned. This is motivated by the appearance of the optimum ridge in Figure 7, which extends over a wide range in C but is always more tightly constrained in the gamma direction.

10 Summary

I have presented a short overview of a Nelder Mead implementation to find the optimum parameters for SVM in classification. A similar implementation can also be used to tune the Svm for parameterization, and is used in GSP-Phot. I gave a brief overview of how the code can be used. I presented a test case showing that the algorithm finds reasonable (although not necessarily globally optimum) solutions in several test cases. I alos discuss possible strategies for overcoming the algorithm's shortcomings, and possible future developments of the code.