



2BIO704 - Report for the Programming Coursework (MSc Biometrics, Jean-Michel Besnard)

1 Content, Usage and Definitions

1.1 Content

The root directory contains:

- source code (.java files)
- compiled classes (.class)

There are also three sub-directories offering the following content:

- **GNUPlot**: data and scripts to plot them in GNUPlot
- **Documentation**: the documentation related to this coursework
- **csv**: some csv data files to be used for testing

1.2 Usage

There are two java classes that may be executed (the application and generator of data):

1.2.1 The application

```
java Bio704 <nb threads> <search type> <data file> <weight change offset>
```

- *<nb threads>*: the number of threads to start. It should be as many as the number of CPU in the machine it runs on.
- *<search type>*: the type of search. Either 1, 2 or 3 respectively for stochastic hillclimbing with possible repetition, without repetition and stochastic steepest ascent hillclimbing
- *<data file>*: the path to the csv data file
- *<weight change offset>*: by how much we want to change the weight (0.1 is a reasonable value)

1.2.2 The data generator

`java DataGenerator <width> <length>`

- `<width>`: the width of the data set.
- `<length>`: the length (or number of lines)

1.3 Definitions

- **Error rate**: the sum of the absolute difference between the predicted value and the prediction
- **Better set of weights, error**: we define as “better” what brings lower error rates.

2 Programming and design issues

2.1 Three different algorithms

Three different algorithm (or *search type*, as referred in the usage section) have been implemented.

2.1.1 Stochastic Hillclimbing, with possible repetitions

In the first one we try to create a new set of weights by making a change in a randomly chosen weight and see if the error rate gets better. If it does, then we replace the current set by this new one. Otherwise, we try another one until we consequently haven't been able to make any improvement for a given amount of time, such as the number of possible combinations of changes.

The problem with this algorithm is to determine when we get stuck and should try all over with a totally new set of weights. If the amount of time we need to loop is too low, we may well miss some good weights and determine that we are stuck too early. But if the number is too high, then the algorithm becomes far less efficient and still does not guarantee to not miss a better weight.

2.1.2 Stochastic Hillclimbing, without repetitions

In the second one, we avoid this by creating a “bank” of (randomly organised) possible candidates to succeed to the the current set of weights. Those candidates are all the possible combinations of changes that can be made from the current set of weights. When we fetch one of these possible weights in the “Bank”, this particular set is then removed so that we can not choose it again (no repetition). By doing so, if we keep having no improvement until the “Bank” is empty, then, for sure, we are stuck.

A critic to this is that we go ahead whenever we find a better weight. We may well miss one weight belonging to the same combination that would bring even better results. This is what the third algorithm tries to cope.

2.1.3 Stochastic steepest ascent algorithm

In this algorithm, we still have a “bank”. This one does not need to be random because we are going to go through all the set to find which one is the best among all the combinations. Once we have found the best one, it becomes the current set of weights. If none is improving the current set, then we are stuck.

However this algorithm tends to improve the previous one and may therefore seem more efficient it is actually not the case. The algorithm is much more consuming and does not even bring better results.

3 Comparison of vectors and testing

3.1 Comparison of vectors

To compare the different set of weights, we calculate the sum of the absolute value of the predicted value minus the prediction value for each “line of data”. The lower the error is, the better vector is. Using the absolute value aims at avoiding positive and negative values to compensate themselves within the sum.

In the application, there are two ways to compare two sets of weights. The first one consists in creating two sets of predictions (*PredictionsSet*), both with different set of weights (*WeightsSet*) and “ask” one of them to compare to the other other one by using the method *isBetterThan(PredictionsSet)*.

The second way is to “ask” a *PredictionsSet* if another given set of weights (*WeightsSet*) would be “better”, by calling the method *betterWithThatWeightSet(WeightsSet)*. Doing one or another makes no big difference. In the first case, the set of training data which might be quite big is shared between all the instances of *PredictionsSet* so it makes no difference.

While using a recursive method to find the best set of weight, it has appeared that the recursion went far too deep. When recuding of the offset of the change made to weights, the recursion gets even deeper.

3.2 Facts

Reducing this offset makes the computation far more intense but also allow to get far better results. Using the csv files given for the coursework, when we use an offse of 0.1, we can obtain an error around 300 after few seconds. While using 0.001, it may take up to one minute before getting stuck but the first result will likely be close to 120. So it is wort using very smaller offsets because however they are much longer for processing, they bring very good results immediatly.

Another factor having a major role is the amount of data. The more data do not only make the computation harder but also makes it much harder to find a set of weight that is close to the optimal one. Increasing the number of data tends to increase the difficulty exponentially.

All the data files given for the coursework, when loading them into the application and looking for the best set of weight with an offset, give error rates around 300 after few seconds while all data of the same size generated with the *DataGenerator* class gave rates such as 50 after the same amount of time.

The variation of the weights during the searches is very low when the offset (for changing the weight) is set to 0.1 but they vary far more when the offset gets smaller (like 0.001).

3.3 Testing/experiment

The aim of the experiment is to see how the 3 different algorithms perform with the different offsets.

Using the *genTestingData.sh* script, we generate 5 data files having each 100 lines of 5 data. We want to try the 3 algorithms with 3 different offsets: 0.1, 0.01 and 0.001. To have more accurate results, each try is run 5 times for each of the 5 data files. So, to sum up we have:

$5 \text{ data files} * 3 \text{ offsets} * 3 \text{ algorithms} * 5 \text{ tries} = 225 \text{ searches}$

The loop to run all these tries and generate the output is implemented in the *testing.sh* script. The output data is processed by another script that will output data to plot.

Furthermore, when using heuristic search, the point is to get close vectors in a very short time. So, we will run this script twice: one where each run is limited to 5 seconds and one limited to 30 seconds.

The tests are performed on a Bi-CPU Pentium III 650MHz with 512MB of RAM, running GNU/Linux, so we runs the searches in 2 threads, to make a more efficient use of the 2 CPUs.

The whole experiment can be resumed by the following commands:

```
genTestingData.sh
testing.sh 5 >5_seconds_attempts_2threads.dat
testing.sh 30 >30_seconds_attempts_2threads.dat
genStats.pl < 5_seconds_attempts_2threads >fiveSec.dat
genStats.pl < 5_seconds_attempts_2threads >thirtySec.dat
```

We then have two files that we can plot using GNUPlot: *fiveSec.dat* and *thirtySec.dat*. The *plot5.gp* and *plot30.gp* scripts can be loaded in GNUPlot to obtain the following two graphs:

The ordinate represents the sum of errors over all the tries for each algorithm. For the abscissa, using the values of the offsets would have distorted the graph too much

so we have to consider that 1, 2 and 3 respectively correspond to the 0.1, 0.01 and 0.001 offsets.

Regarding the graphs representing the '5 seconds' tries, the data report that by 7 times (out of 225), the search did not have enough time to find at least one result. This always happened with steepest ascent algorithm (algo 3) when the offset was set to 0.001. The steepest ascent algorithm is the slowest one of all and gets even slower with smaller offsets.

The first graph shows how bad the first algorithm performs in comparison to the other two. For this trial, the amount of consequent non-improvement to determine that we were stuck in that algorithm was set to the amount of possible combination. Increasing this number would offer better results but since the time for processing them is increased, it is not guaranteed that they would have been done within the given time.

It is interesting to see that the algorithm 2 and 3 are almost identical. But, if we look more in the details, we can see that the second algorithm is a little bit better.

Furthermore, the third algorithm in the first trial (5 seconds) did sometime not have time to find a single result.

As a conclusion, the Stochastic Hillclimbing without repetition seems to be faster. Attention should be further paid to how the "bank" of candidates is implemented in order to save some computation resource. Since every search is independant, it is interesting to see that we could have them to run on different machine reporting their result to collector, as it was done for this application with the *ResultCollector* class that is used to handle results performed by threads.

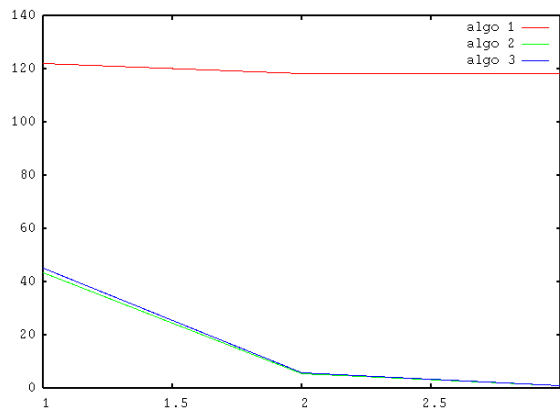


Figure 1: Searches limited to 5 seconds to perform results

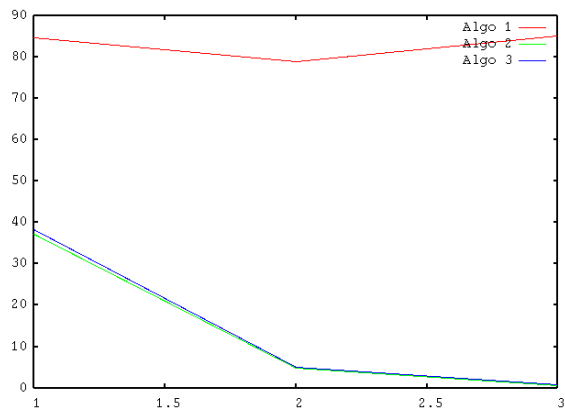


Figure 2: Searches limited to 30 seconds to perform results