# GA-FLM User's Manual

Kevin Duh
duh@ee.washington.edu

Dept of Electical Engineering
University of Washington
Seattle WA, 98195-2500

December 5, 2004

## Introduction

GA-FLM is a genetic algorithms program for automatically learning factored language model structures. It is used as an extension to the factored language model programs in the SRI Language Modeling toolkit.

The program takes as input some training/development text files and some parameter files that specify the type of genetic algorithms and factored language model desired by the user. It then uses standard genetic algorithms search to build a population of factored language models and optimizes for their development set perplexity.

The manual is divided as follows:



## 1 GA-FLM Overview

The program is divided into roughly two components:

The first component is the standard genetic algorithms component, which evolves a population of genes (bit/integer strings) by applying selection, crossover, mutation, and fitness evaluation. The files that do this are:

- `ga-flm.cc`: main program that runs the overal genetic algorithm.
- `ga-operator.cc`: implements various selection/crossover/mutation.

The second component converts a given gene into a factored language model. Specifically, it converts a bit/integer string into a factor-file. This is done by first growing the backoff graph structure using the graph production rules specified by the gene, then printing this graph into a factor-file format. The files responsible for this process are:

- `Gene.cc`, which is a class representing a given bit/string sequence
- `BackoffGraph.cc`, which grows a backoff graph and prints the factor file.
- `Node.cc`, which represents a node in the backoff graph and is contained in the BackoffGraph object.

The general flow of the program is illustrated in Figure 1. First, we initialize a population of genes. Then, for each gene, we grow its backoff graph and print out its factor-file. From this factor-file we train and test the language model. The perplexity is used to determine the fitness value of the gene. After fitness evaluation, selection/crossover/mutation occurs and the next population of genes is ready to be evaluated. The program runs until convergence or maximum generation specified by the user. Finally, “elitist” selection is implemented, so the best gene of each generation is gauranteed a spot in the next generation.

## 2 General Requirements and Installation

Make sure you have the SRI Language Modeling Toolkit Version 1.4.1 or above. The two executables that are needed in that package are ”fngram-count” and ”fngram”, which builds and tests factored language models, respectively. The toolkit can be downloaded at: http://www.speech.sri.com/projects/srilm

To install, simply type ”make” in the root directory. This will create an executable ”ga-flm”.

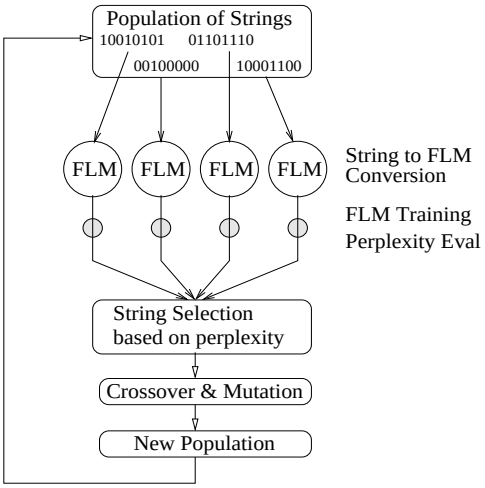


Figure 1: Overall program flow of GA search for FLM structure.

# 3 Running the program

There are two executables that can be used when running a genetic algorithm search. In general, the Perl wrapper version is more convenient because it allows for repeated genetic algorithm runs and prints out summary files after each run.

## 3.1 Running the C++ executable directly: ga-flm

```
ga-flm [-g gaparam] [-f flmparam] [-p pmakeparam] [-s seed]
```

- `[-g gaparam]` specifies GA parameters, such as population, max generation, crossover/selection type, mutation/crossover rate, etc. It also specifies the path where all factor-file data are stored. The default file is GA-PARAMS.
- `[-f flmparam]` tells GA-FLM about factored language model specification, e.g. what factors are available, where is the train or dev set, etc. The default is file FLM-PARAMS.
- `[-p pmakeparam]` include specialized options for PMAKE, a program that helps distribute parallel jobs to a compute cluster. This is specific to the PMAKE program. If PMAKE is not availabe, then set the USE_PMAKE option to “no”. Alternatively, implement another parallel execution function in the ga-flm code. The default file is PMAKE-PARAMS.

- `[-s seed]` is a file specifying a number of seed genes. Usually genetic algorithms begin with a randomly initialized population. Seed genes are user-specified genes that directly inserted in the initial population. In this file, each line represents a gene (see example/SEED_example). There can be any number of seed genes; the SEED file can also be empty (in which case, all genes will be randomly generated). The default file is SEED. (Be careful that the length of the gene in SEED is the same as the total length of the gene expected by the program. If there are $M$ factors available total, then there are $M$ initial factor bits, $B = \sum_{i=2}^{M}(i)$ backoff bits. The final length is thus $M + B + S$, where $S$ is whatever length specified for smoothing options.

- Almost all of the information about the genetic algorithm and factored language model are contained in the parameter files. In general, the user should find it sufficient in his/her use of GA-FLM to just modify parameter files.

### 3.2 Running the Perl wrapper: repeat-gaflm.pl

```
perl repeat-gaflm.pl [num] [-g gaparam] [-f flmparam] [-
p pmakeparam] [-s seed]
```

- `repeat-gaflm.pl` is a Perl wrapper to `ga-flm`. There are two additions that make this Perl wrapper more flexible:

- 1. The user can specify [num], which will call `ga-flm` repeatedly for [num] times. This is useful for when the user wants to run several genetic algorithm experiments.

- 2. The Perl wrapper produces a "timelog" and a "summary" file, which summarizes the perplexity results across the entire genetic algorithm run. This easy-to-read summary file is a convenient way to acquire the best factor-files after a genetic algorithm run.

- The `[g|f|p|s]` parameters are simply passed on to `ga-flm`, and are therefore the same.

- The default for [num], when omitted, is 1.

## 4 Example runs of GA-FLM

The following two examples illustrates the use of GA-FLM to find factored language model structures. The data used is the CoNLL 2002 shared task corpus. It is included in example/data/

### 4.1 Serial execution example

Try the following command:

```
perl repeat-gaflm.pl 2
```

This will run GA-FLM twice. No `[g|f|p|s]` options are given, so the defaults (e.g. GA-PARAMS, FLM-PARAMS, PMAKE-PARAMS, SEED) are read. Note that PMAKE-PARAMS specifies USE_PMAKE to "no", which in the current implementation implies serial execution for evaluating language model fitness. This may be a bit slow, but we have set the population and generation in GA-PARAMS to be small, so the GA-FLM should not take too long.

From GA-PARAMS, we know that the results are stored in example/serial. Go to that directory. You'll find a bunch of files:

- `.flm` files and `.EvalLog` files are the factor-file and perplexity file of a specific gene, respectively. The `.flm` files are generated from transforming the gene sequence to a factored language model. The `.EvalLog` are obtained after training the factored language model on the train-set and testing on the dev-set.
- The `logfile` files contain information about population fitness across generations.
- The `summary` file contains all the factor-files of the genes over the entire genetic algorithm run, together with their corresponding perplexities. The end of the file contains a list of genes in descending order of perplexity.
- The `.complexity` files represents the number of parameters used by the factor file and is used by the Bayesian Info Criteria (BIC) fitness function. If the InversePPL fitness function is specified instead, these files are not needed.

### 4.2 Parallel Pmake execution example

Genetic algorithms are very suited to parallel processing. The building and testing of each factored language model can be done independently at parallel machines, and no communication between machines are needed. Try the following command if you have the PMAKE parallel jobs distribution program available:

```
perl repeat-gaflm.pl -g example/GA-PARAMS-parallel -p
example/PMAKE-PARAMS-parallel
```

Note that we have only changed the GA-PARAMS and PMAKE-PARAMS files. Specifically, the PMAKE-PARAMS file now tells GA-FLM to use paral-

lel processing. The GA-PARAMS file specifies the results to be stored in example/parallel.

If you do not have PMAKE, you should implement an interface to whatever parallel processing program exist on your computer system. (Serial processing is inherently slow and not recommended; it is included in this package only for demonstration and debugging purposes). See Section 7 for instructions on how to do this.

# 5 GA-PARAMS File

Here is an example GA-PARAMS file:

```
Population Size = 4
Maximum no of generations = 2
P(Crossover) = 0.9
P(Mutation) = 0.01
Crossover type = 2-point
Selection type = SUS
(tournament_selection) n = 3
Fitness scaling constant = 1000000
Fitness function = BIC
BIC_constant_k = 1.0
PATH_for_GA_FLM_files = example/serial/
```

All the above parameters are quite useful to the general user. They can be divided into three groups.

## 5.1 Genetic algorithm operators

The first 5 parameters are standard parameters to genetic algorithm. Usually, to find good factored language models, one needs to experiment with several different crossover/selection/mutation and population sizes. In more detail:

- Crossover type may be {2-point,1-point,uniform}
- Selection type may be {SUS, roulette, tournament}
- (tournament_selection) n is the number of genes picked in tournament selection

### 5.2 Fitness function

The following 3 parameters define the fitness function:

- Fitness scaling constant is a linear scaling constant that makes sure fitness values are in reasonable ranges. Make this very high if you want a high convergence threshold (genetic algorithm stops quicker), or lower if you want to fine-tune the last generations.
- Fitness function is currently {BIC,inversePPL}, two different methods to evaluate fitness of a language model.
- BIC_k can be any float number. It is used as the Bayesian Info Criteria weight for complexity penalty. It is ignored when inversePPL is selected as fitness function.

The inversePPL fitness function is implemented as:

$$fitness_{inversePPL} = c * (\frac{1}{perplexity}) \quad (1)$$

where $c$ is the fitness scaling constant.
The BIC fitness function is implemented as:

$$fitness_{BIC} = c * (-logprobability + \frac{k}{2} * log(N)) \quad (2)$$

where $c$ is the fitness scaling constant, $k$ is the weighting for logprobability vs. complexity penality, and $N$ is the number of parameters used by the factored language model under evaluation. ($N$ is the total number of n-gram types for the factored language model, which is taken from the `.complexity` file).

You may wish to implement your own fitness function. To do so, hack the following part of the code: `ga-flm.cc::collectPPL()`. Add your own fitness function after the BIC and inversePPL fitness functions. The quantities "logprob", "complexity", and "ppl" (perplexity) are available for fitness computation. Your fitness function may also need some input specified through parameter files. In that case, add some code to the "READ GA-PARAMS FILE" portion of `ga-flm.cc::initialize(void)`. Refer to the "BIC k value" implementation there for an example.

### 5.3 Path for GA-FLM files

- PATH_for_GA_FLM_files is where all the EvalLog, ga.pmake, and factor-files are stored

# 6 FLM-PARAMS File

Here is an example FLM-PARAMS File:

Data Path = example/data/
TrainSet = wsj_20_train.factored.txt
DevSet = wsj_20_dev.factored.txt
fngram-count Path = fngram-count
fngram Path = fngram
fngram-count Options = ””
fngram Options = ””
Factor to Predict = W
Total Available Factors = W,P,C
Context of Factors = 1,2
Smoothing Options Length = 16
Discount Options = kndiscount, cdiscount 1,wbdiscount, ndiscount
Default Discount = wbdiscount
Max cutoff (max gtmin) = 5

The above parameters relate to the factored language model training/testing and can be divided into 3 general sections: parameters regarding data, parameters regarding the SRI Language Modeling program, and parameters regarding the factored language model specification:

## 6.1 Data location

- Data_Path is the directory path for both training and dev sets.
- TrainSet and DevSet are the filenames of the training and dev set. These files are the factored textfiles assumed by the SRILM fngram programs. (e.g. The sentences are composed of factored bundles like W-word:P:pos:C:chunk). See example/data for examples.

## 6.2 SRI Language Modeling program options

- FngramCount_Path and Fngram_Path are the paths of the `fngram-count` and `fngram` files, respectively.
- FngramCount_options and Fngram_options may contain any command-line options for training/testing the language model. These options are passed directly to `fngram-count` and `fngram`. For example,
  FngramCount_options = ”-unk”

If no options are needed, use FngramCount_options = "" (with the quotes)

### 6.3 Factored Language Model Specifications

- Factor_to_Predict is the factor we wish to predict through the language model. Usually, this is W (word), but may be any factor.

- Total_Available_Factors and Context_of_Factors together specify the tags of factors that will be used in the GA optimization. There are two ways to specify these two parameters:
    1. Specify both factors and context desired, e.g.
        Total_Available_Factors = W,S,P
        Context = 1,2
        = You'll get factors W(-1),S(-1),P(-1),W(-2),S(-2),P(-2).
    2. Specify factor-tags directly without specifying context,e.g.
        Total_Available_Factors = W1,S1,P1,W2,S2,P2
        Context = 0
        = Note that in this case, you must set context=0.

  The advantage of method 1 is that there's less typing to do. The advantage of method 2 is that you can specify arbitrary factor combinations. For example, you can specify W1,S1,S2 which is not possible with method 1.

  Be sure that the tags and numbers for context are separated by commas (,) as that is the delimiter that signals the program how to tell the tags apart.

- Smoothing_Options_Length is the number of smoothing option parameters. This defines the length of the smoothing options gene. Since smoothing is defined as tuples of (discount,gtmin), this should be any even number $\{0,2,4,...\}$. If you believe smoothing options are important in the optimization, make this gene long.

- Discount_options can be any of the usual fngram discounting options (e.g. kndiscount,ukndiscount,wbdiscount,cdiscount 1,kndiscount). Like the Total_Available_Factors, be sure to use commas to delimit discounting options. Any number of discounting options may be used.

- Default Discount is the one used for unigram and other cases when there are more backoff edges to smooth than number of smoothing options specified in Smoothing_Options_Length.

- Max_Cutoff indicates the max possible value for gtmin. gtmin is the min cutoff count for a ngram to be included in the language model estimation.

# 7 PMAKE-PARAMS File

Here is an example of the PMAKE-PARAMS file:

Use Pmake? = yes
Num_Pmake_Jobs = 20
Scratch_Space_to_Use = example/parallel/
Export_Attributes = Linux !1000RAM !2000RAM

- Use_Pmake = {yes,no}. Current implementation can evaluate FLMs serially or parallelly using Pmake. Enter yes here if Pmake is available. Otherwise, enter no to use serial or adapt the code to your own parallel processing platform.
- Num_Pmake_Jobs is the max number of pmake jobs (e.g. -J option in pmake)
- Scratch_space is the place for storing big language model files, count files, and train/dev set data. In the Pmake implementation, all these files are stored locally in the scratch space of each computer. They are deleted after each fitness evaluation.
- Export attributes are the Pmake export attributes. (e.g. Linux !1000RAM !2000RAM).

You may wish to implement your own parallel execution code. To do so, hack the following part of the code:
`ga-flm.cc::executeFngramCommands(fngramCommands)`.

The `fngramCommands` is a queue object containing a list of commands for all genes that need to be evaluated in this generation. These commands are independent, and can therefore be executed in parallel without any mutual communication. As long as you ensure that all commands in `fngramCommands` are executed before `ga-flm.cc::executeFngramCommands(fngramCommands)` exits, then GA-FLM will work fine. Refer to the serial execution and the `executeFngramCommandsByPmake(fngramCommands)` for example implementations.

After implementing this code, be sure to modify the `USE_PMAKE` variable and the PMAKE-PARAMS file so you can tell GA-FLM which execution mode you desire.

## 8 Credits and References

GA-FLM is the work of several people. The algorithm for automatically learning factored language model structure was developed with Katrin Kirchhoff. Jeff Bilmes gave invaluable support on the SRILM factored language model programs. Sonia Parandekar wrote almost all of the basic genetic algorithms code.

Published research using GA-FLM may cite the paper [2].

For additional information on the genetic search for factored language model structure, refer to [3], the extended technical report version of [2]. For information on factored language models, refer to [1] and [5]. [6] gives an overview the SRI Language Modeling toolkit. A detailed manual explaining the fngram programs in SRILM may be found in Section 5 of [4].

## References


[1] Jeff A. Bilmes and Katrin Kirchhoff. Factored language models and generalized parallel backoff. In *Proceedings of HLT/NAACL*, pages 4–6, 2003.

[2] Kevin Duh and Katrin Kirchhoff. Automatic learning of language model structure. In *Proc. of the 20th International Conference on Computational Linguistics (COLING-2004)*, Geneva, Switzerland, Aug 2004.

[3] Kevin Duh and Katrin Kirchhoff. Automatic learning of language model structure (extended version of COLING-2002 paper. Technical Report UWEETR-2004-0014, University of Washington, Dept. of Electrical Engineering, April 2004.

[4] K. Kirchhoff et al. Novel speech recognition models for Arabic. Technical report, Johns Hopkins University, 2002.

[5] K. Kirchhoff et al. Novel approaches to Arabic speech recognition: Report from 2002 Johns-Hopkins summer workshop. In *Proceedings of ICASSP*, pages I–344–I–347, 2003.

[6] Andreas Stolcke. SRILM- an extensible language modeling toolkit. In *Proceedings of ICSLP*, pages 901–904, Denver, Colorad, September 2002.