SAUTRELA: A HIGHLY MODULAR OPEN SOURCE SPEECH RECOGNITION FRAMEWORK

Mikel Penagarikano and German Bordel

Department of Electricity and Electronics
University of the Basque Country, 48940 Leioa, Spain
E-mail: mpenagar@we.lc.ehu.es, german@we.lc.ehu.es
Web: gtts.ehu.es, www.sautrela.org

ABSTRACT

This paper describes the Sautrela system (www.sautrela.org), a highly modular and pluggable open source framework for generic purpose signal processing, focused on speech recognition. The aim of Sautrela is to unify in a single framework almost all the tasks related to pattern recognition such as signal processing, model training and decoding. This framework has been developed using the Java™ Technology and thus ensures its portability to a large variety of computer platforms.

1. INTRODUCTION

Since the beginning of the speech recognition research, many different speech recognition systems have been developed [1, 8, 16, 15]. All of them were mainly focused on the research of a couple few stages of the recognition system, and turned out to be quite hardwired implementations since they were optimized for the studied methodology. A good example of this can be the family of Sphinx systems at CMU. The original system [7] was focused on the use of discrete Hidden Markov Models (HMM), whereas later forks focused on semi-continuous HMMs [5] and continuous HMMs [11, 13]; Each new fork didn't include the previous design, but implemented a new approach to acoustic modelling (nowadays, Sphinx-2 and Sphinx-3 are still live projects). Even more, all those systems were designed as decoder machines while other involved matters, such as models training, were achieved by other packages.

On the other hand, Sautrela aims to unify in a single modular framework most important tasks related to pattern recognition including signal processing, model training and decoding. The framework shares its modular essence with the design of a recent fork of the Sphinx system, Sphinx-4 [14]. Although Sphinx-4 started as a port of Sphinx-3 to Java programming language, it evolved into a completely different speech recognition system; flexibility played a mayor role in the design, resulting on a really modular framework for speech decoding research. Sautrela, also developed using the Java™ Technology, goes one step beyond and defines a framework where not only decoding, but training and any signal processing derived task can be achieved. In addition, no assumption is made on the nature of the processed signal, thus being possible to use many different signal source types.

2. SOME REMARKS ON JAVA TECHNOLOGY

Java is a platform-independent object-oriented programming language. Java byte-codes can be executed in almost all operating systems by the corresponding Java Virtual Machine (JVM), ensuring the portability of the framework to a large variety of computer platforms without any need of configuration or recompilation. Java code can also be compiled to native code in order to increase performance, but at the expense of platform independence.

The performance of Java applications is sometimes seen as a negative issue. First implementations of JVM used to run java code more slowly than equivalent programs written in C or C++, but today's just-in-time compilers (JIT) or newer dynamic-recompilation VMs achieve closer performance. It should be also taken into account that Java offers a rich and very efficient data structure framework that can be widely used without any effort. Note that the comparison carried out between the native code Sphinx-3 and the Java based Sphinx-4, showed that the performances of both systems were about the same [14].

The run-time of Java is dynamic in the sense that classes are linked on demand, and therefore, any application written in java is straightforward pluggable. In addition, the object-oriented paradigm offers the possibility to design the framework by only providing the required interfaces, leaving for a later step the implementation of each concrete module.

Java supports both multi-threading and synchronization at the language level. The ability of performing multiple tasks simultaneously enhances the performance and func-

tionality of a software design. As it will later be stated, Sautrela is based on multiple data processing modules running on different threads. This schema not only simplifies the required programming effort, but also takes advantage of Symmetric Multiprocessing machines (SMP) and emerging multi-core processor architectures (common desktop computers already enhances the performance of multi-threaded applications). In addition, the automatic garbage collection allows the programmer not to worry about memory management.

Finally, it should be mentioned that the really important set of enhancements introduced to the last version of Java (1.5 - sept 2004), dramatically contributes to improve the previous formulation of Sautrela, taking full advantage of these benefits.

3. SAUTRELA FRAMEWORK

The Sautrela architecture is inspired on the front-end module defined in the Sphinx-4 speech recognition system, where input signals are processed by a pluggable data-processor pipeline in order to obtain the features to be processed by the decoder module. Those data-processors perform all the signal processing steps previous to the decoding task (such as preemphasis, windowing, Fourier transform, filtering, cosine transform and mean normalization), and the decoder module does the search task based on the processed input signal. Thus the Sphinx-4 system differentiates both modules, the front-end and the decoder.

On the contrary, Sautrela treats the decoder like another pluggable data-processor, unifying the system architecture. Even more, all training procedures such as cluster training for vector quantization and acoustic/lexical/language model training are formalized using the dataprocessor paradigm, and thus integrated in a single and simple framework. Such a framework relaxes almost all constraints that other systems present and offers a high freedom to explore emerging methodologies on pattern recognition without the need to build a new system from scratch. Although most of the currently developed modules are related to speech processing (the team involved in the development is mainly focused on speech technologies and software engineering), it represents a useful tool for many other disciplines. Summarizing, Sautrela defines a flexible signal processing framework usable for almost all the tasks involved in many pattern recognition research areas.

3.1. Engine

The Engine (see Fig.1) is the basis of the Sautrela framework. An Engine is an arbitrary-length connected list of processing modules (DataProcessor) running on multiple threads. Each module performs a data-processing stage,

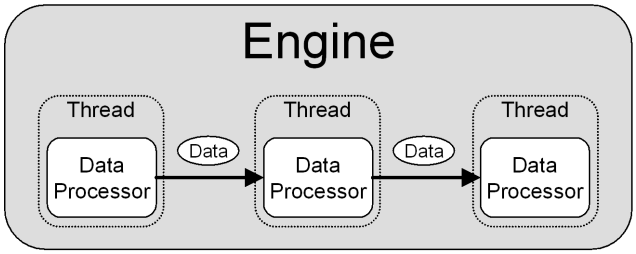

Fig. 1. An Engine is a connected list of DataProcessors running on multiple threads. Each module performs a data-processing stage.

while the Engine represents the whole process. There are two ways of creating a working Engine at runtime: It can be instantiated and then filled with newly instantiated modules, or it can also be entirely instantiated from a XML descriptor that contains the list of involved modules as well as the information to correctly instantiate and configure each of them. In the first case, any class implementing the simple DataProcessor interface can be plugged into, whereas in order to be instantiated from XML, those modules must also conform the Java Beans design pattern.

Encapsulated in a generic Data object, input data signal propagates through the Engine, being processed by each module. Data objects are used not only for signal encapsulation, but many other tasks such as input-output control and marks. For example, data streams can be simply created with encapsulated begin and end marks, and since those streams are composed by Data objects, it is possible to define nested data streams (for example, a speech segmentator could convert an input signal in sentence streams, while a later acoustic segmentator could create inner streams representing word or phone boundaries).

Each module in the Engine can run on a separate thread, and once an Engine is fully defined and started, all the processing modules start running and there is no way to directly stop the engine. Indeed, the first module in the pipeline will decide when there is no more data to process (or could be externally notified to stop) and a terminating signal will be propagated encapsulated in a Data object. The multithreaded nature of the engine allows plugging processing modules with any input-output configurations in the time domain, such as Multiple-Input Multiple-Output. It should be noted that we are talking about the time domain, and thus we mean that a module is not forced to generate a valid output data for each input data, and can also generate output data even in the absence of input.

The engine is responsible of the interconnection of all the modules. This is performed using intermediate buffers. As depicted in Fig.2, the input Data of a module is pulled from the prior buffer, processed and pushed into the later

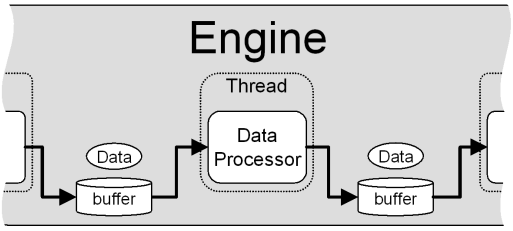

Fig. 2. Input Data of a module is pulled from the prior buffer, processed and pushed into the later one. Buffering policies allow static/dynamic size as well as blocking.

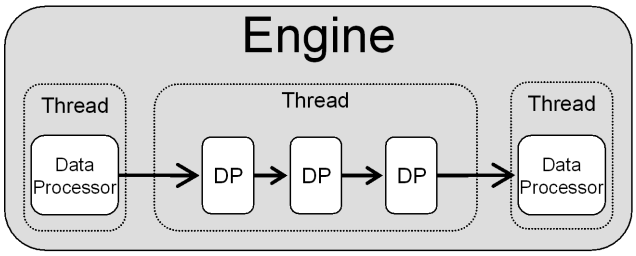

Fig. 3. Sautrela can manage a list of connected modules as a simple DataProcessor, simply plugging it into an engine like any other threaded module.

one. Different buffering policies are available: buffer size can be fixed or ideally unlimited (in the absence of memory or resource constraints) and a blocking mechanism can be applied to the push action (it waits if necessary for space to become available). Note that the pull action is always blocking oriented, since no Data can be retrieved from an empty input buffer. Blocking should not be used when live input signals are used, whereas it is strongly recommended for batch processing (otherwise any module slower than the input signal reading one would result in a bottleneck).

3.2. DataProcessor

A DataProcessor represents an isolated processing module. It is defined as a very simple interface that should be implemented by any class in order to be pluggable into the framework. Such a module must implement a simple input-output method for processing the input Data already returned by the predecessor DataProcessor. In the case of a simple Single-Input-Single-Output module, this method will return the processed Data, while in more complex modules, encapsulated empty signals (EmptyData) are used for input-output control: a Multiple-Input-Single-Output module returns EmptyData on non empty Data inputs to report that no output Data is available yet, and a Single-Input-Multiple-Output module returns valid non empty Data on EmptyData inputs to report that more output Data is still available (output data end is reported returning an EmptyData). EmptyData object is just used for module's input-output management, and is never propagated through the engine. Note that the Engine must deal with the connection of modules, multiple input-output management and Data buffering, while modules must only implement the simple DataProcessor interface.

First and last modules of an Engine are somehow special: The initial module must behave as a No-Input Multiple-Output system, that is, must always generate non empty Data on EmptyData input, whereas the last one should behave as a Multiple-Input No-Output system, since processed Data cannot be managed. Typically, the first module requires Data from a device (file system or audio capture) and the last one dumps Data to another device (file system or audio playback).

A DataProcessor finishes its processing as soon as it processes an incoming encapsulated close signal (CloseData): it returns a CloseData object (or even a Data sequence with a final CloseData), all Data is sent to the next module and the thread of the module terminates. The remaining modules are terminated as CloseData propagates through the Engine.

A multi-threaded engine can take advantage of threading techniques of the latest microprocessors as well as SMP computers, while at the same time it could result on an unnecessary overload if too many simple modules were plugged. In those cases, it is possible to share a single thread among a set of consecutive modules (see Fig.3). Sautrela can manage such a list of connected modules as a simple DataProcessor, and therefore simply plug it into an engine. The ability to join a set of modules on a single thread can be used to optimize overall performance of the system depending on the underlying hardware.

Next, some implemented modules are briefly explained.

3.2.1. Simple signal processing modules

Sautrela provides a set of signal processing modules mainly focused on state of the art speech processing technologies: live audio recording and playback, batch audio reading and writing, silence detection, preemphasis, raised cosine transform windowing, discrete Fourier transform (FFT), Mel frequency filtering, discrete cosine transform (DCT), cepstral mean normalization (CMN), Mel-cepstra frequency coefficient extraction (MFCC), delta and acceleration (delta-delta) coefficients and vector quantization (VQ). All modules can be configured at runtime and there are no static limits like maximum window size, FFT sizes and so on.

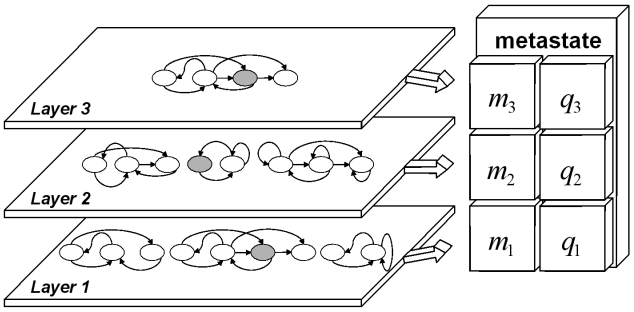

Fig. 4. Different knowledge levels can be integrated into a single layered Markov model. The whole model can be seen as a non-deterministic WFA and its *meta-states* are composed by a model-state pair (m_i, q_i) out of each layer.

3.2.2. An ELBG clustering module

Although a vector quantization module is implemented in the framework, a codebook must be obtained prior to use it. Namely, some clustering algorithm must be applied to the training data in order to get an optimum set of codewords. Sautrela implements a multidimensional clustering module using the Enhanced LBG algorithm [9] that can be connected at the end of a processing pipeline. Once all input Data is buffered clustering algorithm is performed. When convergence is reached, the module saves the codebook and the processing thread is finished.

Note that there is no need to build a pre-processed database in order to set up a training Data set, since the same pre-processing modules used at decoding can also be used at training and thus it is possible to work with original input signals.

3.2.3. A general purpose decoding module

Sautrela defines two major interfaces to unify many different stochastic model techniques related to speech recognition: the deterministic and the non-deterministic Weighted Finite-state Automata (WFA). Regardless of its internal structure, any model that implements one of them can be managed by the system: An integrated module takes advantage of the said interfaces and turns out to be a simple and general decoder. However, a speech recognition system is composed of many different models standing for as many more knowledge sources. Hence, a single integrated stochastic model is needed, and layered Markov models (LMM) [10] have been implemented for that purpose.

A LMM consists of a number of layers, each of them composed by a finite set of deterministic WFAs (see Fig.4). Each set represents a knowledge layer that models its units in terms of lower level layer units (for example, the set of pronunciation models of an speech decoder models the

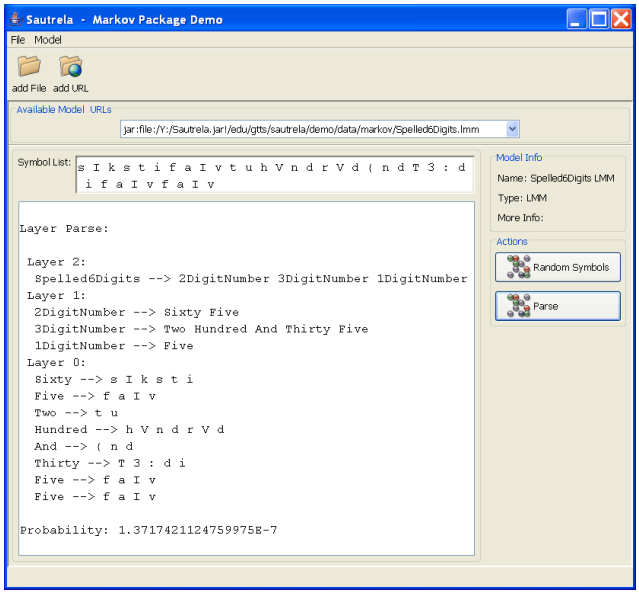

Fig. 5. A 3-layer LMM model for the phonetic transcription of a spelled 6-digit number. The top layer (1 model) represents how a 6-digit number can be segmented in 1, 2 and 3 digit numbers. The middle layer (3 models) describes the 1, 2 and 3 digit numbers in terms of words. And last, the bottom layer (31 models) represents the phonetic transcription of each word. In the example, a random phoneme list is decoded.

words in terms of phonemes, diphonemes or other similar units). In other words, each layer is connected to the underlying one by means of a mapping of its alphabet into the lower's model set. The states of a LMM, called *meta-states*, are characterized by a vector made up of a model-state pair per layer, and the alphabet of the whole model is the bottom layer's alphabet (a similar architecture can be found in [1]).

Regardless of the number of layers involved, a LMM can be seen as a non-deterministic WFA and straightforward integrated in the aforementioned decoder module. The decoder, implements well known search algorithms such as frame-synchronous Viterbi, A*, and a configurable pruning [6]. There are no static limitations on the number of layers, models, states or alphabet size that shape a LMM, and so it turns out to be a highly flexible structure to implement a generic stochastic decoder (see Fig.5).

State of the art speech recognition systems use deterministic WFAs and hidden Markov models (HMM) as major techniques for modelling, and both can be integrated in a LMM (a set of HMM can be broken down in a 2-layer LMM that can be then added to another LMM [10]). Therefore, many hardwired model mixing found at most systems can be formalized as a LMM. For example, the Sphinx-4 system implements a Linguist (the structure used to create a

```
<?xml version=”1.0” encoding=”UTF-8” ?>
<Engine name="KT Language Model Trainer">

  <DataProcessor name=”dp 1”
    code=”edu.gtts.sautrela.util.TextReader” />
    <param name=”inputURL” value=”file://simple.text” />
  </Dataprocessor>

  <DataProcessor name=”dp 2”
    code=”edu.gtts.sautrela.util.TextCleaner” />
    <param name=”regexp” value=”file://rules.regexp” />
  </Dataprocessor>

  <DataProcessor name=”dp 3”
    code=”edu.gtts.sautrela.model.Trainer” />
    <param name=”wfaURL” value=”file://emptymodel.lm” />
    <param name=”criteria” value=”ML” />
  </Dataprocessor>

  <Thread name="A">
    <addDataProcessor name="dp 1" />
    <addDataProcessor name="dp 2" />
  </Thread>

  <Thread name="B">
    <addDataProcessor name="dp 3" />
  </Thread>

  <Link from="A" to="B" bufferSize=”10” blocking=”true”/>
</Engine>
```

Fig. 6. This XML represents and Engine designed to train a KT[3] Language Model. Three main DataProcessors are used: a simple text reader that reads text from a given URL and generates String streams and a second module that applies regular expressions to ensure proper string format, both running on the same thread, and a general trainer that calculates the optimum probabilities for the given automata and optimization criterion.

search graph) made up of three knowledge layers: a bottom acoustic layer with HMMs, an intermediate dictionary and a language layer at the top. Dictionary and language layers are in fact made up of Markov models, although they can be defined in a conceptually different manner, as for the language model that can be given as a graph-driven grammar or a stochastic n-gram model.

3.2.4. A general purpose training module

A somehow analogous idea has been carried out to design a general purpose training module. Using the WFA interface of a model, the training processor can estimate the optimum probabilities (from the interface point of view) for a given observation set and a maximization criterion. The model is then responsible of translating those probabilities to its internal representation. Well known training criteria [12] and corresponding transformations [2, 4] are implemented. Figure 6 shows an instantiable xml description of an engine that contains a trainer processor.

4. GRAPHICAL ENGINE BUILDER

Sautrela incorporates a graphical building-blocks interface able to manage the XML descriptors and the engines themselves. This graphical tool simplifies the assembly and tuning of the engines and offers a friendly user interface to the framework. The builder relies on the Java Beans software component model, which must be implemented by any external pluggable module in order to be accessible from the builder.

5. FUTURE WORK

Some preliminary speech recognition experiments are now being carried out with Sautrela. As a result, some performance figures and structural metrics will be obtained for future reference. It is planned to make this first version available to general public as open source by the fourth quarter of 2005, though a beta version and other documentation related to the project can be found at *www.sautrela.org*.

Future development efforts point to various directions. Some of them are:

Allowing a distributed Engine: in order to increase the performance for intensive experiments, and to admit more flexible architectures, it is planned to weak the DataProcessor links. Given that these links are now very neat, it seems reasonably to think that this enhancement will not present much difficulty.

Some Signal Processing applications like multimedia processing or multi-modal systems need not just a linear structure of DataProcessors but the possibility to insert multiple input and/or multiple output modules giving place to parallel paths for the data. There is no much problem to adapt the actual Sautrela to these needs, slightly affecting to the DataProcessor interface and to the XML descriptor structure and processing.

6. CONCLUSION

Sautrela, a novel framework for signal processing and pattern recognition has been presented. It turns out to be a highly modular, flexible and pluggable system, useful for generic signal processing and pattern recognition, including both model training and decoding. General purpose processing modules as well as training modules have been introduced, and with respect to the decoding, it has been shown that commonly hardwired mixture of models can be reflexively formalized as a layered-knowledge model and integrated in an implemented generic decoder module.

Developed using the Java™ technology, Sautrela is architecture-neutral and, at the same time, can benefit from emerging thread oriented hardware.

7. REFERENCES

[1] J.K. Baker. The dragon system – an overview. *IEEE Trans. Acoust. Speech Signal Processing*, ASSP-23(1):24–29, 1975.

[2] Leonard. E. Baum and George R. Sell. Growth transformations for functions on manifolds. *Pacific Journal of Mathematics*, 27, 1968.

[3] G. Bordel and A. Varona. K-tlss(s) language models for speech recognition. In *Proceedings of the ICASSP*, 1997.

[4] P. S. Gopalakrishnan, D. Kanevsky, A. Nádas, and D. Nahamoo. An inequality for rational funations with applications to some statistical estimation problems. *IEEE Trans. Information Theory*, 37:107–113, 1991.

[5] X. Huang, F. Alleva, H. Hon, M. Hwang, and R. Rosenfeld. The SPHINX-II speech recognition system: an overview. *Computer Speech and Language*, 7(2):137–148, 1993.

[6] F. Jelinek. *Statistical methods for speech recognition*. MIT Press, Cambridge, MA, USA, 1997.

[7] K. Lee, H. Hon, and R Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(1):35 – 45, January 1990. see also IEEE Transactions on Signal Processing.

[8] B.T. Lowerre. *The harpy speech recognition system*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1976.

[9] G. Patanè and M. Russo. The enhanced lbg algorithm. *Neural Networks*, 14(9):1219–1237, November 2001.

[10] M. Penagarikano and G. Bordel. Layered Markov Models: a new architectural approach to automatic speech recognition. In *Proceedings of the MLSP Workshop*, 2004.

[11] P. Placeway, S. Chen, M. Eskenazi, U. Jain, V. Parikh, B. Raj, M. Ravishankar, R. Rosenfeld, K. Seymore, M. Siegler, R. Stern, and E. Thayer. The 1996 Hub-4 Sphinx-3 System. In *Proceedings of the 1997 DARPA Speech Recognition Workshop*, 1997.

[12] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):237–240, 1989.

[13] K. Seymore, S. Chen, S. Doh, M. Eskenazi, E. Gouvea, B. Raj, M. Ravishankar, R. Rosenfeld, M. Siegler, R. Stern, and E. Thayer. The 1997 CMU Sphinx-3 English Broadcast News Transcription System. In *Proceedings of the 1998 DARPA Speech Recognition Workshop*, 1998.

[14] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel. Sphinx-4: A Flexible Open Source Framework for Speech Recognition. Technical Report TR-2004-139, Sun Microsystems, 2004.

[15] P.C. Woodland, J.J. Odell, V. Valtchev, and S.J. Young. Large vocabulary continuous speech recognition using HTK. In *Proceedings of the ICASSP*, Adelaide, April 1994.

[16] S. Young. The htk hidden markov model toolkit: Design and philosophy. Technical Report TR.153, Department of Engineering, Cambridge University, Cambridge, UK, 1993.