THE KALDI SPEECH RECOGNITION TOOLKIT

Daniel Povey[a] Arnab Ghoshal[b] Gilles Boulianne[c]
Lukas Burget Ondrej Glembek[d] Nagendra Goel[e]
Mirko Hannemann[d] Petr Motlicek Yanmin Qian[f]
Petr Schwarz[d] Jan Silovsky[g] Georg Stemmer[h]
Karel Vesely[d]

Idiap-RR-04-2012

JANUARY 2012

[a]Microsoft Research, USA
[b]Saarland University, Germany
[c]Centre de Recherche Informatique de Montr´eal, Canada
[d]Brno University of Technology, Czech Republic
[e]Go-Vivace Inc., USA
[f]Tsinghua University, China
[g]Technical University of Liberec, Czech Republic
[h]SVOX Deutschland GmbH, Germany

The Kaldi Speech Recognition Toolkit

Daniel Povey[1], Arnab Ghoshal[2],
Gilles Boulianne[3], Lukáš Burget[4], Ondřej Glembek[4], Nagendra Goel[5], Mirko Hannemann[4],
Petr Motlíček[6], Yanmin Qian[7], Petr Schwarz[4], Jan Silovský[8], Georg Stemmer[9], Karel Vesely[4]

[1] *Microsoft Research, USA,* dpovey@microsoft.com*;*
[2] *Saarland University, Germany,* aghoshal@lsv.uni-saarland.de*;*
[3] *Centre de Recherche Informatique de Montréal, Canada;* [4] *Brno University of Technology, Czech Republic;*
[5] *Go-Vivace Inc., USA;* [6] *IDIAP Research Institute, Switzerland;* [7] *Tsinghua University, China;*
[8] *Technical University of Liberec, Czech Republic;* [9] *SVOX Deutschland GmbH, Germany*

***Abstract*—We describe the design of Kaldi, a free, open-source toolkit for speech recognition research. Kaldi provides a speech recognition system based on finite-state automata (using the freely available OpenFst), together with detailed documentation and a comprehensive set of scripts for building complete recognition systems. Kaldi is written is C++, and the core library supports modeling of arbitrary phonetic-context sizes, acoustic modeling with subspace Gaussian mixture models (SGMM) as well as standard Gaussian mixture models, together with all commonly used linear and affine transforms. Kaldi is released under the Apache License v2.0, which is highly nonrestrictive, making it suitable for a wide community of users.**

I. Introduction

Kaldi[1] is an open-source toolkit for speech recognition written in C++ and licensed under the Apache License v2.0. It is intended for use by speech recognition researchers. The goal of Kaldi is to have modern and flexible code that is easy to understand, modify and extend. Kaldi is available on SourceForge (see http://kaldi.sf.net/). The tools compile on the commonly used Unix-like systems and on Microsoft Windows.

Researchers on automatic speech recognition (ASR) have several potential choices of open-source toolkits for building a recognition system. Notable among these are: HTK [1], Julius [2] (both written in C), Sphinx-4 [3] (written in Java), and the RWTH ASR toolkit [4] (written in C++). Yet, our specific requirements—a finite-state transducer (FST) based framework, extensive linear algebra support, and a non-restrictive license—led to the development of Kaldi.

The work on Kaldi started during the 2009 Johns Hopkins University summer workshop project titled "Low Development Cost, High Quality Speech Recognition for New Languages and Domains," where we were working on acoustic modeling using subspace Gaussian mixture model (SGMM) [5]. Important features of Kaldi include:

Integration with Finite State Transducers: We compile against the OpenFst toolkit [6] (using it as a library).

Extensive linear algebra support: We include a matrix library that wraps standard BLAS and LAPACK routines.

[1]According to legend, Kaldi was the Ethiopian goatherd who discovered the coffee plant.

Extensible design: As far as possible, we provide our algorithms in the most generic form possible. For instance, our decoders work with an interface that provides a score for a particular frame and FST input symbol. Thus the decoder could work from any suitable source of scores.

Open license: The code is licensed under Apache v2.0, which is one of the least restrictive licenses available.

Complete recipes: Our goal is to make available complete recipes for building speech recognition systems, that work from widely available databases such as those provided by the Linguistic Data Consortium (LDC).

Thorough testing: The goal is for all or nearly all the code to have corresponding test routines.

The paper is organized as follows: we start by describing the structure of the code and design choices (section II); then describe the matrix library of Kaldi (section III) and the FST library (section IV). This is followed by describing the individual components of a speech recognition system that the toolkit supports: feature extraction (section V), acoustic modeling (section VI), phonetic decision trees (section VII), language modeling (section VIII), and decoders (section X). Finally, we provide some benchmarking results in section XI.

II. Overview of the toolkit

We give a schematic overview of the Kaldi toolkit in figure 1. The toolkit depends on two external libraries that are also freely available: one is OpenFst [6] for the finite-state framework, and the other is numerical algebra libraries. We use the standard "Basic Linear Algebra Subroutines" (BLAS)and "Linear Algebra PACKage" (LAPACK)[2] routines for the latter purpose, the details of which are described in section III.

We aim for the toolkit to be as loosely coupled as possible to make it easy to reuse and refactor. This is reflected in the structure of the toolkit, where the library modules can be grouped into two distinct halves, each depending on only one of the external libraries. A single module, the `DecodableInterface` (cf. section X), bridges these two halves. This sort of decoupling is also reflected in the design of individual classes. We have tried to structure the toolkit in

[2]Available from: http://www.netlib.org/blas/ and http://www.netlib.org/lapack/ respectively.

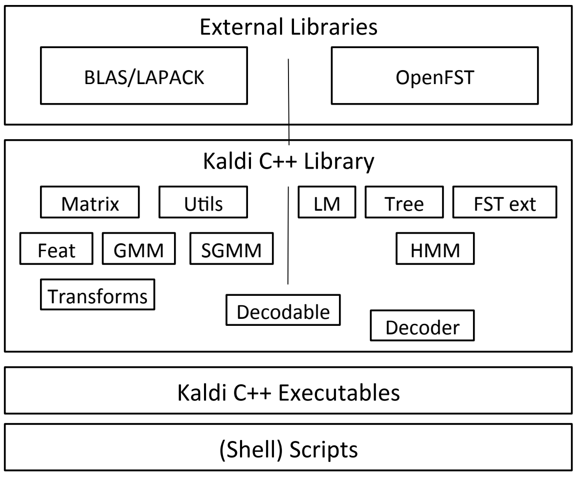

Fig. 1. A simplified view of the different components of Kaldi. The library modules can be grouped into those that depend on linear algebra libraries and those that depend on OpenFst. The *decodable* class bridges these two halves. Modules that are lower down in the schematic depend on one or more modules that are higher up.

such a way that implementing a new feature will generally involve adding new code and command-line tools rather than modifying existing ones. We consider this an advantage for maintainability and extensibility.

Access to the library functionalities is provided through command-line tools written in C++, which are then called from a scripting language for building and running a speech recognizer. While this is similar to the traditional approach followed in several toolkits (e.g. HTK), the Kaldi approach differs fundamentally in how we view the tools. Each tool has very specific functionality: for example, there are separate executables for accumulating statistics, summing accumulators, and updating a GMM-based acoustic model using maximum likelihood estimation. As such the code for the executables tend to be very simple with only a small set of command line arguments. Moreover, all the tools can read from and write to pipes which makes it possible to chain together different tools.

An approach that has recently become popular is to have a scripting language such as Python call the C++ code directly, and to have the outer-level control flow and system design done in this scripting language. This is the approach used in IBM's Attila toolkit [7]. The design of Kaldi does not preclude doing this in future, but for now we have avoided this approach because it requires users to be proficient in two different programming languages.

A final point to mention is that we tend to prefer provably correct algorithms. There has been an effort to avoid recipes and algorithms that could possibly fail, even if they don't fail in the "normal case" (for example, FST weight-pushing, which normally helps but which can fail or make things much worse in certain cases).

III. The Kaldi Matrix Library

The Kaldi matrix library provides C++ classes for vector and different types of matrices (description follows), as well as methods for linear algebra routines, particularly inner and outer products, matrix-vector and matrix-matrix products, matrix inversions and various matrix factorizations like Cholesky and singular value decomposition (SVD) that are required by various parts of the toolkit. The library avoids operators in favor of function calls, and requires the matrices and vectors to have correct sizes instead of automatically resizing the outputs.

The matrix library does not call the Fortran interface of BLAS and LAPACK directly, but instead calls their C-language interface in form of CBLAS and CLAPACK. In particular, we have tested Kaldi using the "Automatically Tuned Linear Algebra Software" (ATLAS) [8] library, the Intel Math Kernel Library (MKL) library, and the Accelerate Framework on OS X. It is possible to compile Kaldi with only ATLAS, even though ATLAS does not provide some necessary LAPACK routines, like SVD and eigenvalue decompositions. For those routines we use a C++ implementation of code from the "Java Matrix Package" (JAMA) [9] project.

A. Matrix and vector types

The matrix library defines the basic `Vector` and `Matrix` classes, which are templated on the floating point precision type (`float` or `double`).

```
template<typename Real> class Vector;
template<typename Real> class Matrix;
```

The `Matrix` class corresponds to the general matrix (GE) in BLAS and LAPACK. There are also special classes for symmetric matrices (`SpMatrix`) and triangular matrices (`TpMatrix`) (for Cholesky factors). Both of these are represented in memory as a "packed" lower-triangular matrix. Currently, we only support linear algebra operations with real numbers, although it is possible to extend the functionality to include complex numbers as well, since the underlying BLAS and LAPACK routines support complex numbers also.

For matrix operations that involve only part of a vector or matrix, the `SubVector` and `SubMatrix` classes are provided, which are treated as a pointer into the underlying vector or matrix. These inherit from a common base class of `Vector` and `Matrix`, respectively, and can be used in any operation that does not involve memory allocation or deallocation.

IV. Finite State Transducer (FST) library

We compile and link against an OpenFst [6], which is an open-source weighted finite-state transducer library. Both our training and decoding code accesses WFSTs, which are simply OpenFst's C++ objects (we will sometimes refer to these just as FSTs).

We also provide code for various extensions to the OpenFst library, such as a constructed-on-demand context FST (C) which allows our toolkit to work efficiently for wide phonetic context. There are also different versions of or extensions to some of the fundamental FST algorithms such as determinization, which we implement with epsilon removal and mechanisms to preserve stochasticity (discussed in Section IX); minimization, which we modify to work with non-deterministic input; and composition, where we provide a more efficient version. We provide command-line tools with interfaces similar to OpenFst's command line tools, to allow these algorithms to be used from the shell.

V. Feature Extraction

Our feature extraction and waveform-reading code aims to create standard MFCC and PLP features, setting reasonable defaults but leaving available the options that people are most likely to want to tweak (for example, the number of mel bins, minimum and maximum frequency cutoffs, etc.). The feature extraction pipeline is implemented as a series of functions, each of which output a matrix of floating point numbers and take a matrix as input, except the windowing function, which reads the waveform samples as a vector.

The windowing function can optionally dither (add random Gaussian noise to) the waveform, remove DC offset and pre-emphasize the higher frequencies. Our FFT implementation [10] works for window lengths that are not powers of 2, and we also provide an implementation of split-radix FFT for 0-padded windows whose lengths are powers of 2. We also support cepstral liftering and optional warping of the Mel filter banks using vocal tract length normalization (VTLN). Cepstral mean and variance normalization, dynamic (i.e. delta) features of arbitrary order, splicing of arbitrary number of frames to the left or right of the current frame and linear projections of such high-dimensional features using linear discriminant analysis (LDA) or heteroscedastic linear discriminant analysis (HLDA) [11] are supported at the executable layer through simple command line tools. Additionally, we support reading and writing of features in the format used by HTK [1].

VI. Acoustic Modeling

Our aim is for Kaldi to support conventional models (i.e. diagonal GMMs) and Subspace Gaussian Mixture Models (SGMMs), but also to be easily extensible to new kinds of model. Following the general design philosophy of Kaldi, the acoustic modeling code is made up of classes with very specific functionality that do not "know anything" about how they get used. For example, the `DiagGmm` class just stores the parameters of a diagonal-covariance Gaussian mixture model (together with accessors and mutators for the parameters) and provides methods for likelihood computation. Estimation of GMMs is handled by a separate class[3] that accumulates the sufficient statistics.

A. Gaussian mixture models

We support GMMs with diagonal and full covariance structures. Rather than representing individual Gaussian densities separately, we directly implement a GMM class that is parametrized by the *natural parameters*, i.e. means times inverse covariances and inverse covariances. Such an implementation is suitable for efficient (log-)likelihood computation with simple dot-products. The GMM classes also store the *constant* term in likelihood computation, which consist of all the terms that do not depend on the data vector. In other words, the constant term is the likelihood of the zero vector.

[3]In fact, different estimation classes are responsible for different estimation algorithms, e.g. ML or MMI.

B. GMM-based acoustic model

The "acoustic model" class `AmDiagGmm` represents a collection of `DiagGmm` objects, indexed by "pdf-ids" that correspond to context-dependent HMM states. Note that the acoustic model is not actually an HMM, but just a collection of densities. There are separate classes that represent the HMM structure, principally the topology and transition-modeling code and the code responsible for compiling decoding graphs, which provide a mapping between the HMM states and the pdf index of the acoustic model class. This class is implemented as a `std::vector` of type `DiagGmm`, with a slightly richer interface that supports, among other things, setting the number of Gaussian components in each pdf proportional to the occupancy of the corresponding HMM state. The classes for estimating the acoustic model parameters are likewise implemented as a `std::vector` of GMM estimators. Speaker adaptation and other linear transforms like maximum likelihood linear transform (MLLT) [12] or semi-tied covariance (STC) [13] are implemented by separate classes.

C. HMM Topology

It is possible in Kaldi to separately specify the HMM topology for each context-independent phone. The topology format allows nonemitting states, and allows the user to pre-specify tying of the p.d.f.'s in different HMM states; the main envisaged use of this is for more advanced transition modeling.

D. Speaker adaptation

We support both model-space adaptation using maximum likelihood linear regression (MLLR) [14] and feature-space adaptation using feature-space MLLR (fMLLR), also known as constrained MLLR [15]. For both MLLR and fMLLR, multiple transforms can be estimated using a regression tree [16]. When a single fMLLR transform is needed, it can be used as an additional processing step in the feature pipeline. It is also possible to only estimate the bias vector of an affine transform, or the bias vector and a diagonal transform matrix, which are suitable when the amount of adaptation data is small[4]. The toolkit also supports speaker normalization using a linear approximation to VTLN, similar to [17], or conventional feature-level VTLN, or a more generic approach for gender normalization which we call the "exponential transform" [18]. Both fMLLR and VTLN can be used for speaker adaptive training (SAT) of the acoustic models.

E. Subspace Gaussian Mixture Models

For subspace Gaussian mixture models (SGMMs), the toolkit provides an implementation of the approach described in [5]. There is a single class `AmSgmm` that represents a whole collection of pdf's; unlike the GMM case there is no class that represents a single pdf of the SGMM. Similar to the GMM case, however, separate classes handle model estimation and speaker adaptation using fMLLR.

[4]This is currently implemented only for fMLLR.

VII. Phonetic Decision Trees

Our approach for using decision trees that handle phonetic context dependency is quite generic, and is designed to efficiently support wide phonetic contexts, as well as phonesets that have been expanded to include extra information such as stress and position within a word. The conventional approach [19] is to have a single decision tree for each HMM-state of each phone, or for each phone, and to ask questions about the phones to the left and right; the questions are asked in a greedy way to maximize likelihood given a single-Gaussian model. We support a more generic range of approaches, including a single global decision tree, and decision tree roots that are shared among specified groups of phones, e.g. if the phones contain stress and position information, we might share the root among a group of such phones that corresponds to a "real" phone. To make this work, we make it possible to ask questions about the central phone and the identifier of the p.d.f. within the prototype topology: in effect, this is asking about the HMM state. The internal code for constructing and applying decision trees is very generic, and based on the abstract notion of key-value pairs (e.g., the position in a phonetic context window would be a key, and the phone at that position would be a value). The code is also designed to be easily extensible to clustering based on models other than a single Gaussian.

None of the algorithms for training and applying decision trees ever enumerate all the possible phonetic contexts; this important in order to generalize efficiently to wider-than-triphone contexts. In order to avoid making monophone models a special case, we simply treat them as context-dependent models with a phonetic-context window of 1. Note that the position of the "central phone" within the context window can be specified, which allows us to support things like left and right biphone context dependency.

The toolkit makes it possible to supply arbitrary phonetic questions for the decision tree clustering (note that for us, a question is simply a set of phones). In the conventional approach (e.g. [19]), these would be specified by a human based on linguistic knowledge. However, since our goal is to supply algorithms that are automated and generic as possible, using only publicly available resources, all the experiments we report here are with automatically derived questions. These are based on a binary tree clustering of the phones; each question corresponds to all the leaves under a particular node in this binary tree. For experiments with stress and position-dependent phones, we make the leaves of the tree correspond to sets of phones for each "real" phone, and we introduce separate questions that correspond to stress and position (this is all done at the shell-script level).

VIII. Language Modeling

Since Kaldi uses an FST-based framework, it is possible, in principle, to use any language model that can be represented as an FST. We are working on mechanisms that are able to handle LMs that would get too large when represented this way. We provide tools for converting LMs in the standard ARPA format to FSTs. In our recipes, we have used the IRSTLM toolkit [20] for purposes like LM pruning. For building LMs from raw text, users may use the IRSTLM toolkit, for which we provide installation help, or a more fully-featured toolkit such as SRILM [21].

IX. Creating Decoding Graphs

All our training and decoding algorithms use Weighted Finite State Transducers (WFSTs). In the conventional recipe [22], the input symbols on the decoding graph correspond to context-dependent states (in our toolkit, these symbols are numeric and we call them pdf-ids). However, because we allow different phones to share the same pdf-ids, we would have a number of problems with this approach, including not being able to determinize the FSTs, and not having sufficient information from the Viterbi path through an FST to work out the phone sequence or to train the transition probabilities. In order to fix these problems, we put on the input of the FSTs a slightly more fine-grained integer identifier that we call a "transition-id", that encodes the pdf-id, the phone it is a member of, and the arc (transition) within the topology specification for that phone. There is a one-to-one mapping between the "transition-ids" and the transition-probability parameters in the model: we decided make transitions as fine-grained as we could without increasing the size of the decoding graph. An advantage of having the transition-ids as the graph input symbols is that all we need in Viterbi-based model training is the sequence of input symbols (transition-ids) in the Viterbi path through the FST. We call this sequence an alignment. A set of alignments gives us all the information we need in order to train the p.d.f.'s and the transition probabilities. Since an alignment encodes the complete phone sequence, it is possible to convert alignments between different decision trees.

Our decoding-graph construction process is based on the recipe described in [22]; however, there are a number of differences. One important one relates to the way we handle "weight-pushing", which is the operation that is supposed to ensure that the FST is stochastic. "Stochastic" means that the weights in the FST sum to one in the appropriate sense, for each state (like a properly normalized HMM). Weight pushing can fail or can lead to bad pruning behavior if the FST representing the grammar or language model (G) is not stochastic, and FSTs based on backoff language models are not stochastic due to redundant paths through backoff and non-backoff arcs. Our approach is to avoid weight-pushing altogether, but to ensure that each stage of graph creation "preserves stochasticity" in an appropriate sense. Informally, what this means is that the "non-sum-to-one-ness" (the failure to sum to one) will never get worse than what was originally present in G. This requires changes to some algorithms, e.g. to determinization. There are other differences too: we minimize after removing disambiguation symbols, which is more optimal but requires changes to the minimization code of OpenFst; and we use a version of determinization that removes input epsilon symbols, which requires certain changes in other parts of the recipe (chiefly: introducing disambiguation symbols on the input of G).

The graph creation process required in test time is put together at the shell-script level. In training time, the graph creation is done as a C++ program which can be made more efficient. The aim is for all the C++ tools to be quite generic, and not to have to know about "special" things like silence. For instance, alternative pronunciations and optional silence are supplied as part of the lexicon FST (L) which is generally produced by a script.

X. Decoders

We have several decoders, from simple to highly optimized; more will be added to handle things like on-the-fly language model rescoring and lattice generation. By "decoder" we mean a C++ class that implements the core decoding algorithm. The decoders do not require a particular type of acoustic model: they need an object satisfying a very simple interface with a function that provides some kind of acoustic model score for a particular (input-symbol and frame).

```
class DecodableInterface {
 public:
  virtual float LogLikelihood(int frame, int index) = 0;
  virtual bool IsLastFrame(int frame) = 0;
  virtual int NumIndices() = 0;
  virtual ~DecodableInterface() {}
};
```

Command-line decoding programs are all quite simple, do just one pass of decoding, and are all specialized for one decoder and one acoustic-model type. Multi-pass decoding is implemented at the script level.

XI. Experiments

We report experimental results on the Resource Management (RM) corpus and on Wall Street Journal. We note that the experiments reported here should be fully reproducible, except for minor differences in WER due to differences in compiler behavior and random number generation algorithms. The results reported here correspond to version 1.0 of Kaldi; the scripts that correspond to these experiments may be found in `egs/rm/s1` and `egs/wsj/s1`, and we will provide "system identifiers" (corresponding to training runs) to help locate particular experiments in our scripts.

The scripts include all data preparation stages, and require only the original datasets as distributed by the Linguistic Data Consortium (LDC).

A. Comparison with previously published results

We first report some results intended to demonstrate that the basic algorithms included in the toolkit give results comparable to those previously reported in the literature.

Table I shows the results of a context-dependent triphone system with mixture-of-Gaussian densities; the HTK baseline numbers are taken from [23] and the systems use essentially the same algorithms. The features are MFCCs with per-speaker cepstral mean subtraction. The language model is the word-pair bigram language model supplied with the RM corpus. The WERs are essentially the same. Decoding speed was about 0.13xRT, measured on an Intel Xeon CPU at 2.27GHz. The system identifier for the Kaldi results is tri3c.

TABLE I
Basic triphone system on Resource Management: %WERs

	Test set				
	Feb'89	Oct'89	Feb'91	Sep'92	Avg
HTK	2.77	4.02	3.30	6.29	4.10
Kaldi	3.20	4.21	3.50	5.86	4.06

TABLE II
Basic triphone system, WSJ, 20k open vocabulary, bigram LM, SI-284 train: %WERs

	Test set	
	Nov'92	Nov'93
Bell	11.9	15.4
HTK (+GD)	11.1	14.5
KALDI	11.8	15.0

Table II shows similar results for the Wall Street Journal system, this time without cepstral mean subtraction. The WSJ corpus comes with bigram and trigram language models, and most of our experiments use a pruned version of the trigram language model (with the number of entries reduced from 6.7 million to 1.5 million) since our fully-expanded FST gets too large with the full language model (we are working on decoding strategies that can work with large language models). For comparison with published results, we report bigram decoding in Table II, and compare with published numbers using the bigram language model. The baseline results are reported in [24], which we refer to as "Bell" (for Bell Labs, the authors' affiliation), and a HTK system described in [25]. Note that the HTK baseline is gender dependent while ours is not, so the comparison may not be entirely fair; we have other algorithms, such as VTLN to handle gender dependency, and chose not to build this type of gender-dependent system. The system id for the Kaldi system is tri3a. Our results are slightly better than the Bell Labs results, and although HTK's are better than ours, this difference can be attributed to the gender dependency.

B. Other experiments

Here we report some more results on both the WSJ test sets (Nov'92 and Nov'93) using systems trained on just the SI-84 part of the training data, that demonstrate different features that are supported by Kaldi. Note that the triphone results for the WSJ sets are worse than those in Table II

TABLE III
Results on RM and WSJ, 20k open vocabulary, bigram LM, trained on half of SI-84: %WERs

	RM (Avg)	WSJ Nov'92	WSJ Nov'93
Triphone	3.97	12.5	18.3
+ fMLLR (spk)	3.59	11.4	15.5
+ LVTLN (spk)	3.30	11.1	16.4
Splice-9 + LDA + MLLT	3.88	12.2	17.7
+ SAT (fMLLR)	2.70	9.6	13.7
SGMM	3.32	10.4	16.4
+ spk-vecs (spk)	2.76	10.0	13.81
+ fMLLR (spk)	2.55	9.9	13.4

due to the smaller training set. We also report results on the RM task for comparison. The RM results reported here are averaged over 6 test sets–the 4 mentioned in table I together with Mar'87 and Oct'87–and so the WER with the basic triphone system is different from the average in Table I. The best result for conventional GMM system is achieved by a speaker-adaptively trained system that splices 9 frames (4 on each side of the current frame) and uses LDA to project down to 40 dimensions, together with MLLT. Comparable performance is achieved with an SGMM system trained on MFCC features (static $+\Delta+\Delta\Delta$) that uses speaker vectors and fMLLR adaptation (we have not yet tried the SGMM system on top of the better LDA-based features).

XII. Conclusions

We described the design of Kaldi, a free and open-source speech recognition toolkit. The toolkit currently supports modeling of context-dependent phones of arbitrary context lengths, and all commonly used techniques that can be estimated using maximum likelihood. It also supports the recently proposed SGMMs. Development of Kaldi is continuing and we are working on using large language models in the FST framework, lattice generation and discriminative training.

Acknowledgments

We would like to acknowledge participants and collaborators in the 2009 Johns Hopkins University Workshop, including Mohit Agarwal, Pinar Akyazi, Martin Karafiat, Feng Kai, Ariya Rastrow, Richard C. Rose and Samuel Thomas; Patrick Nguyen, for introducing the participants in that workshop and for help with WSJ recipes, and faculty and staff at JHU for their help during that workshop, including Sanjeev Khudanpur, Desirée Cleves, and the late Fred Jelinek. We would like to thank Michael Riley, who visited us in Brno to deliver lectures on finite state transducers and helped us understand OpenFst, and Henrique (Rico) Malvar of Microsoft Research for allowing the use of his FFT code. We would like to acknowledge the help with coding and documentation from Sandeep Boda and Sandeep Reddy (sponsored by Go-Vivace Inc.) and Haihua Xu. We thank Pavel Matejka (and Phonexia s.r.o.) for allowing the use of feature processing code.

We would like to acknowledge the support of Geoffrey Zweig and Alex Acero at Microsoft Research. We are grateful to Jan (Honza) Černocký for helping us organize the workshop at the Brno University of Technology during August 2010. We had excellent system support from Tomas Kašpárek, and we thank Renata Kohlová for administrative support.

During the development of Kaldi, Arnab Ghoshal was supported by the European Community's Seventh Framework Programme under grant agreement no. 213850 (SCALE); the BUT researchers were supported by the Technology Agency of the Czech Republic under project No. TA01011328, and partially by Czech MPO project No. FR-TI1/034. The JHU 2009 workshop was supported by National Science Foundation Grant Number IIS-0833652, with supplemental funding from Google Research, DARPA's GALE program and the Johns Hopkins University Human Language Technology Center of Excellence.

Finally, it is quite possible that someone else who helped us significantly was inadvertently omitted here; if so, we apologize.

References

[1] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland, *The HTK Book (for version 3.4)*. Cambridge University Engineering Department, 2009.

[2] A. Lee, T. Kawahara, and K. Shikano, "Julius – an open source real-time large vocabulary recognition engine," in *EUROSPEECH*, 2001, pp. 1691–1694.

[3] 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," Sun Microsystems Inc., Technical Report SML1 TR2004-0811, 2004.

[4] D. Rybach, C. Gollan, G. Heigold, B. Hoffmeister, J. Lööf, R. Schlüter, and H. Ney, "The RWTH Aachen University Open Source Speech Recognition System," in *INTERSPEECH*, 2009, pp. 2111–2114.

[5] D. Povey, L. Burget *et al.*, "Subspace Gaussian Mixture Models–A Structured Model for Speech Recognition," *Computer Speech & Language*, vol. 25, no. 2, pp. 404–439, April 2011.

[6] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri, "OpenFst: a general and efficient weighted finite-state transducer library," in *Proc. CIAA*, 2007.

[7] H. Soltau, G. Saon, and B. Kingsbury, "The IBM Attila speech recognition toolkit," in *IEEE Workshop on Spoken Language Technology (SLT)*, 2010, pp. 97–102.

[8] ATLAS homepage, http://math-atlas.sourceforge.net/.

[9] JAMA homepage, http://math.nist.gov/javanumerics/jama/.

[10] H. S. Malvar, *Signal Processing with Lapped Transforms*. Artech House, Inc., 1992.

[11] N. Kumar and A. G. Andreou, "Heteroscedastic discriminant analysis and reduced rank HMMs for improved speech recognition," *Speech Communication*, vol. 26, no. 4, pp. 283–297, December 1998.

[12] R. Gopinath, "Maximum likelihood modeling with Gaussian distributions for classification," in *Proc. IEEE ICASSP*, vol. 2, 1998, pp. 661–664.

[13] M. J. F. Gales, "Semi-tied covariance matrices for hidden Markov models," *IEEE Trans. Speech and Audio Proc.*, vol. 7, no. 3, pp. 272–281, May 1999.

[14] C. J. Leggetter and P. C. Woodland, "Maximum likelihood linear regression for speaker adaptation of continuous density hidden Markov models," *Computer Speech and Language*, vol. 9, no. 2, pp. 171–185, 1995.

[15] M. J. F. Gales, "Maximum likelihood linear transformations for HMM-based speech recognition," *Computer Speech and Language*, vol. 12, no. 2, pp. 75–98, April 1998.

[16] ——, "The generation and use of regression class trees for MLLR adaptation," Cambridge University Engineering Department, Technical Report CUED/F-INFENG/TR.263, August 1996.

[17] D. Y. Kim, S. Umesh, M. J. F. Gales, T. Hain, and P. C. Woodland, "Using VTLN for broadcast news transcription," in *Proc. ICSLP*, 2004, pp. 1953–1956.

[18] D. Povey, G. Zweig, and A. Acero, "The exponential transform as a generic substitute for vtln," in *Asru 2011 (submitted)*, 2011.

[19] S. J. Young, J. J. Odell, and P. C. Woodland, "Tree-based state tying for high accuracy acoustic modelling," in *Proc. 1994 ARPA Human Language Technology Workshop*, 1994, pp. 304–312.

[20] M. Federico, N. Bertoldi, and M. Cettolo, "IRSTLM: An open source toolkit for handling large scale language models," in *INTERSPEECH*, 2008, pp. 1618–1621.

[21] A. Stolcke, "Srilm-an extensible language modeling toolkit," in *Proceedings of the international conference on spoken language processing*, vol. 2, 2002, pp. 901–904.

[22] M. Mohri, F. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech and Language*, vol. 20, no. 1, pp. 69–88, 2002.

[23] D. Povey and P. C. Woodland, “Frame discrimination training for HMMs for large vocabulary speech recognition,” in *Proc. IEEE ICASSP*, vol. 1, 1999, pp. 333–336.

[24] W. Reichl and W. Chou, “Robust decision tree state tying for continuous speech recognition,” *IEEE Transactions on Speech and Audio Processing*, vol. 8, no. 5, pp. 555–566, September 2000.

[25] P. C. Woodland, J. J. Odell, V. Valtchev, and S. J. Young, “Large vocabulary continuous speech recognition using HTK,” in *Proc. IEEE ICASSP*, vol. 2, 1994, pp. II/125–II/128.