Pmtools: A Pronunciation Modeling Toolkit

Richard Sproat

AT&T Labs — Research
Florham Park, NJ, USA
rws@research.att.com

Abstract

This paper reports on a pronunciation modeling toolkit — *pmtools* — that allows one to train a weighted finite-state transducer using a Classification and Regression Tree (CART) training paradigm. Tools are provided to automatically align a pronunciation dictionary consisting of a set of words and their pronunciations, train a set of CART trees on the aligned dictionary, and compile those trees out into a special class of weighted finite-state transducer. Most of the complexity — aligning the data, labeling the data with features, and training the trees — is hidden from the user.

While some new techniques, e.g. in automatic alignment, are introduced here, the main focus of this work is to provide a toolkit to ease the development of pronunciation models using fairly standard techniques. By the time of the workshop, *pmtools* will be available free for non-commercial use.

1. Introduction

There has been a veritable cottage industry of different approaches to automatically inferring pronunciation models from labeled data; see [18, 9, 7, 11, 1, 6, 21, 2, 8], inter alia. There is also specialized software (e.g. `http://ilk.kub.nl/∼antalb/webstuff.html`) available for training such models.

One popular training paradigm for automatically deducing pronunciation models — whether grapheme-to-phoneme conversion or phoneme-to-phone conversion — is decision trees, especially Classification and Regression Trees, or CART [3]. Tree-based methods were introduced into pronunciation modeling in [4, 14, 15, 16]. One advantage of CART is that it is fairly efficient to train, and can produce relatively compact models.

One disadvantage of CART and other methods, though, is that it typically requires some work on the part of the user to massage the data into a format that can be used to train the models. In pronunciation modeling this will typically require of the user that they convert a human readable dictionary consisting of paired spellings and pronunciations into a fairly opaque set of input and output feature vectors, corresponding to each pairing of input and output symbol in a given context.

A second disadvantage is that, while it is easy to use a set of CART trees to predict pronunciations for strings, it is somewhat less straightforward to use them with recognition networks or lattices; approaches that have used them in this way such as [17] have tended to rely on somewhat ad hoc methods.

This second disadvantage has prompted investigations into *compiling* decision trees into weighted finite-state transducers (WFST's) [20], which can then be used directly as pronunciation models in ASR systems [13], or in TTS systems [19]. But the problem here is that the resulting WFST's can become very large, and in some cases compilation becomes intractable. Paradoxically, the size of transducers has prompted recent research into "re-engineering" WFST's by "compiling" them into decision trees; see [10].

This paper reports on a pronunciation modeling toolkit that attempts to address these disadvantages of CART-based pronunciation models, and their compilation into decision trees. The system, called *pmtools*, provides tools for automatically aligning a pronunciation dictionary, training a set of CART models, and compiling them into a new class of WFST, built on top of the AT&T *fsm* library [12], that uses the decision trees directly. The *tree class* WFST constructs arcs and state "on the fly", and thus obviates the need to compile the trees into a potentially large WFST.

By the time of the workshop, *pmtools*, along with documentation and examples, will be available free for non-commercial use.[1]

2. Alignment: *pmalign*

In training a string-to-string mapping such as a pronunciation model one starts with a "dictionary" that represents a sample of the string-to-string mapping that one is trying to learn. Concrete instances of "dictionaries" include, of course, pronunciation dictionaries representing words in their standard orthography and in phonetic transcription (Table 1); but also, for example, dictionaries of abbreviations and their expansions; or (in the context of speech recognition) phonemic transcriptions of sentences alongside with their observed phonetic sequence. For the sake of the current discussion we will assume we are starting with a pronunciation dictionary, and that we are attempting to infer a set of grapheme-to-phoneme rules.

Typically the first requirement in training a grapeme-to-phoneme model is to produce an *alignment* of the graphemes and phonemes. Such an alignment between the grapheme sequence ⟨nation⟩ and the phoneme sequence /nešən/, might map ⟨n⟩ to /n/, ⟨a⟩ to /e/, ⟨t⟩ to /š/, ⟨i⟩ to null, ⟨o⟩ to /ə/ and ⟨n⟩ to /n/. Such alignments might be done by hand — or more likely by some hand-constructed program (as was the case in the work reported in [16]); or they might be computed by an automatic self-organizing procedure.

One approach in the latter vein is that of Daelemans and van den Bosch [6]. This method considers, for each orthographic-phonetic word pair, all possible starting positions for aligning the two, keeping symbols of each contiguous. Thus for the pair ⟨rookie⟩ and /ruki/ we would have the following alignments, where "–" represents null:

[1]Note that the *fsm* library and the *lextools* package are already available for non-commercial use: `http://www.research.att.com/sw/tools/{fsm,lextools}`.

aaberg	AbRg
aaker	akR
aamodt	&mot
aardema	ordem&
aaron	ar&n
aarons	ar&nz
aaronson	ar&ns&n
aasen	osin
abad	ab&d
abadie	&badE
abair	AbAr
abalos	ab&lOs
abarca	&bork&
abare	Aber
abate	&botE
abbas	ab&s
abbasi	&basE
abbate	&bAt
abbatiello	ab&tEelO
. . .	. . .

Table 1: A sample from an English pronuncation dictionary of names.

(r,r)	(o,u)	(o,k)	(k,i)	(i,–)	(e,–)
(r,–)	(o,r)	(o,u)	(k,k)	(i,i)	(e,–)
(r,–)	(o,–)	(o,r)	(k,u)	(i,k)	(e,i)

Of course in any given case many of the alignments will be implausible; for example there are relatively few cases where the first phoneme of a word corresponds to the third letter, as in the third alignment. But the method depends (as does the method in *pmalign* reported below), upon the majority of specific letter-phoneme alignments being reasonable.

For each alignment computed as above, a heuristic cost is assigned depending upon how much of an offset from the left is found in the alignment. These heuristic costs are then kept in a matrix and for each word the alignment is chosen that minimizes the cost.

The alignment method we present here differs from the Daelemans and van den Bosch method in a few respects. First, while the costs in the Daelemans and van den Bosch method are to some extent ad hoc, in the present method they are based on probability estimates of the grapheme-to-phoneme alignments. Second, their method assumes that, besides substitutions, only deletions (of letters) are necessary, not insertions (of phonetic symbols). But insertions do occur: for example stress symbols in the phonetic transcription usually correspond to no orthographic symbol. The current method handles insertions, as we shall see. Finally, the Daelemans and van den Bosch is a “one-pass” method that is done purely for initial data preparation. In contrast, as we shall discuss below, our method allows for iterative improvement of the alignment, by first aligning the data, training a pronunciation model (Section 3) and realigning the data using the derived pronunciation model, retraining and so forth.

The input to *pmalign* is the textual representation of a dictionary, which should consist of a two column format with the orthography word in the first column and the pronunciation in the second column (assuming one is interested in mapping from graphemes into phonemes); the dictionary in Table 1 is in the appropriate format. Words may have multiple pronunciations, as long as these are listed on separate lines, with the corresponding spelled word in the first column.

’	⟨epsilon⟩	(orth. apostrophe deletes)
⟨epsilon⟩	stress	(stress inserts)
orthVowel	phonVowel	(orth. vow. → phon. vow.)
orthCons	phonCons	(orth. cons. → phon. cons.)
orthLiquid	phonLiquid	(orth. liquids → phon. liq.)

Table 2: Some hints for *pmalign*. Note that ⟨epsilon⟩ here denotes null. The semantics of the various classes (e.g. *phonR*) are defined by a label file that is one of the required arguments to *pmalign*.

In the initial “training” phase, the algorithm computes a simple left-to-right (alternatively right-to-left) one-for-one alignment between orthographic and phonetic symbols for each orthography-pronunciation pair. Statistics on the grapheme-phoneme matches are kept in a table, and are used to construct a single-state WFST, which we call a *map*. Each arc in this WFST is labeled with one of the observed grapheme-phoneme correspondences (as input and output label, respectively), and the cost is the negative log of the probability estimate for the correspondence, where this is computed from the maximum likelihood estimate given the observed frequency.

The initial alignment can be guided by providing a set of *hints*. This is a good thing to do if there are some reasonable general rules of thumb that could benefit the initial alignment: the hints afford a way of providing expert input to the otherwise automatic alignment process, at a minimal cost. For example, reasonable hints might be that a phonetic stress symbol corresponds to no orthographic symbol, or that a phonetic vowel is realized as an orthographic vowel; see Table 2. The hints work as follows. As the aligner traverses the input and output strings, it first looks at the input symbol and sees if there are any input hints for that symbol. If there are, and the hint is to delete (i.e., to map to epsilon), then it will consume the input symbol but not the output symbol and then move on to the next step. If the hint includes the output symbol then both will be consumed. Otherwise, it looks at the output symbol and sees if there are any hints. If there is an insertion hint for the out- put symbol (i.e., it maps *from* epsilon), the output will be consumed; otherwise if there is a match with the input, then the input will be consumed. If these steps all fail then the algorithm tries to greedily delete or insert by looking at whether there are symbols that match if one first eliminates the current input symbol by deletion, or the current output symbol by deletion. If that all fails, then the system simply forces a match between the current input and output symbols.

After the map WFST is constructed it is used to realign the dictionary, and then compute a second round of statistics and a second aligner. The realignment is computed by computing, for each word, the composition of the grapheme sequence represented as a finite-state acceptor (FSA) Γ with the map M and the phoneme sequence represented as an FSA Φ, and then computing the cheapest-cost path:

$$CheapestPath(\Gamma \circ M \circ \Phi)$$

Since the arcs of M are weighted with the unigram negative log probability estimates for the grapheme-phoneme correspondences, this is equivalent to computing

$$argmax \prod_i p(c_i)$$

for each correspondence c_i in the lattice of possible correspondences in $\Gamma \circ M \circ \Phi$.

As implied by the above discussion, this method will produce alignments that include insertions of phonetic symbols, or more specifically — cases where an output symbol corresponds to null on the input side. This is undesirable if the alignment is to be used in a training procedure such as that described in Section 3. One can, of course, easily enough train a model for insertions that predicts, given that you know you will insert, what symbol should be inserted. The problem lies in knowing when you should insert in the first place. (Note that there is no comparable problem with deletions.) One possibility is simply to pad the input with nulls between every pair of real symbols, as proposed in [10]. The solution adopted here is to concatenate output symbols whenever there is an insertion so that a given input symbol may correspond to more than one output symbol: insertion thus gets replaced with substitution with a complex symbol. Complex symbols corresponding to insertions are generated by considering the various ways in which the output symbols could be locally combined into a single symbol. To see how this works consider a case where there are two adjacent insertions:

Input	a	-	-	d
Output	A	B	C	D

There are three possible ways to reanalyze this mapping:

1. $a \rightarrow ABC, d \rightarrow D$
2. $a \rightarrow AB, d \rightarrow CD$
3. $a \rightarrow A, d \rightarrow BCD$

All of these possibilities are proposed. As before, statistics are collected over the entire dictionary, the complex symbols, along with the simple symbol-to-symbol and symbol-to-deletion pairings are added to a new map M', and M' is used to realign the dictionary as before.

The final output of *pmalign* is an aligned dictionary, which is represented as an acyclic WFST. If complex symbols were generated, an auxiliary transducer is produced that translates from those complex symbols to strings of simplex symbols.

3. Training: *pmcontext*

After one has produced an alignment for the dictionary, the next phase is to train a set of decision trees. For each input symbol (in the current discussion, a grapheme) we wish to build a tree that predicts the outputs it can have given the context in which one finds it. In the current version of *pmcontext*, the contexts are defined as fixed windows of zero or more symbols to the left, and zero or more symbols to the right; three symbols to the left and right would be a typical reasonable context.

As with previous CART-based work on speech and language, one needs to provide a featural decomposition of the symbols that one will find in the input. This is because for categorical independent variables (i.e., which of a given set of symbols is at a particular position in the context), the algorithm examines all possible ways of dividing the data. Since this is $O(2^N)$, where N is the number of symbols, this is generally too large for any reasonable set of input symbols; see [15, 16] for discussion. We therefore provide a featural decomposition; see Table 3. Given these user-provided features, *pmcontext* then automatically labels the contexts. For example, assuming a context of three on either side, the context around ⟨a⟩ in ⟨scatter⟩ — ⟨-sc_tte⟩ would be:

a	vow	Va	n/a	n/a
e	vow	Ve	n/a	n/a
i	vow	Vi	n/a	n/a
o	vow	Vo	n/a	n/a
u	vow	Vu	n/a	n/a
b	cons	n/a	Cb	n/a
c	cons	n/a	Cc	n/a
d	cons	n/a	Cd	n/a
f	cons	n/a	n/a	Cf
g	cons	n/a	Cg	n/a
h	cons	n/a	n/a	Ch
j	cons	n/a	n/a	Cj
k	cons	n/a	Ck	n/a
l	cons	n/a	n/a	Cl
m	cons	n/a	n/a	Cm
n	cons	n/a	n/a	Cn
p	cons	n/a	Cp	n/a
q	cons	n/a	Cq	n/a
r	cons	n/a	n/a	Cr
s	cons	n/a	n/a	Cs
t	cons	n/a	Ct	n/a
v	cons	n/a	n/a	Cv
w	cons	n/a	n/a	Cw
x	cons	n/a	Cx	n/a
y	cons	n/a	n/a	Cy
z	cons	n/a	n/a	Cz
pad	pad	pad	pad	pad

Table 3: Featural decomposition of English letters. Columns 2–4 are: major type (vowel or consonant); vowel identity; consonant identity group 1; consonant identity group2. Consonant identity is divided into two features since the set would otherwise be too big. *pad* is a designated pad symbol

```
[pad  pad pad pad]
[cons n/a n/a Cs]
[cons n/a Cc n/a]
____
[cons n/a Ct n/a]
[cons n/a Ct n/a]
[vow Ve n/a n/a]
```

The labeling is accomplished by composing the input side of the aligned dictionary with a transducer (actually a minimal decision tree class transducer Section 4) that maps each distinct input symbol plus context combination to a unique symbol, aligned with the corresponding output symbol. Then, for each input symbol, a tree model is constructed, consisting of a set of feature vectors, and corresponding output values. Note that we do not count the number of times each feature vector/output value pair occurs in the aligned dictionary transducer, but rather the number of times that it occurs in the corresponding word/pronunciation list: single arcs in the transducer may be shared among words and thus may correspond to several instances of the pair in the text version of the dictionary. The true number of instances of a particular pair can be computed straightforwardly from the transducer by calculating the number of distinct paths from the initial to a final state that pass through each arc.

From the tree models, trees are constructed, one for each input symbol, using the CART algorithm [3, 15, 16]. The trees are then collected and compiled into a decision tree transducer, which we describe in the next section.

4. The Decision Tree WFST Class

One important feature of the AT&T *fsm* library [12] is its extensibility. It is possible to define new classes of weighted transducers, so long as these support a set of required operations: these include returning the start state of the machine, the cost of exiting at a state, and returning the set of arcs exiting a state.

The decision tree class WFST builds its states and arcs on the fly. The states encode the context, as in an n-gram model in speech recognition (e.g., [13]). The tree class works by building an n-gram model of the context, where n is the sum of the left-context width, the right context width plus 1 for the target symbol.

The arcs are computed at each state as follows: given a request for an arc leaving state s with label l, state s is consulted to see what context c it represents. The context c is then translated into a feature vector as described in Section 3, and passed to the tree corresponding to label l. For each possible output symbol, the tree predicts a probability of that output symbol given the context, and this information is then converted into a set of arcs labeled with l on the input side, and for each output label l' on the output side, a cost of $-\log(p(l'|c,l)$.

The tree WFST is initially the size of the set of compiled tree data and so is equivalent in space efficiency to a direct use of the trees. However, as the WFST is used (i.e., composed with other WFST's), it will grow, as more contexts are seen and remembered. However, it is possible to reinitialize the WFST, thus freeing up memory. A decision tree WFST can also be converted into WFST's of other classes, or precomposed with a dictionary. A reasonable use of a tree WFST, for instance, is to compose it with a dictionary, thus producing a new dictionary that represents the mapping between a set of words in the original dictionary and their predicted pronunciations.

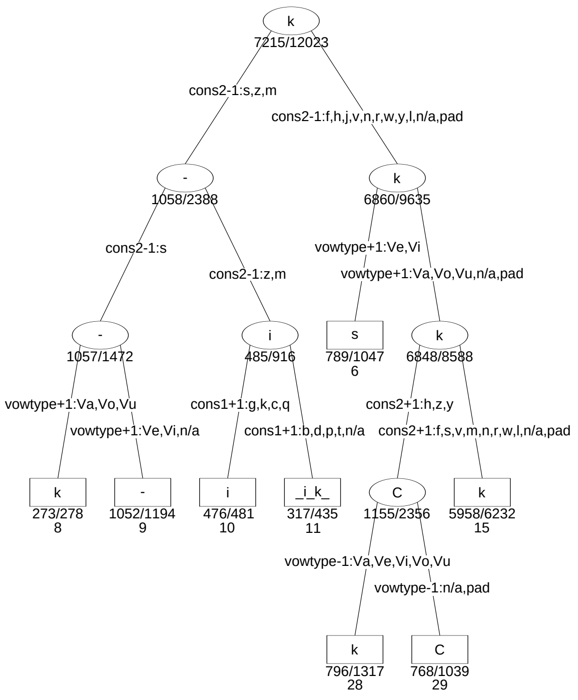

Figure 1: Pruned tree for ⟨c⟩.

5. An Example

As an example, we took a (family) name dictionary consisting of 48,980 names and their pronunciations; a fragment of this dictionary was shown in Table 1; note that this dictionary was part of the pronunciation dictionary used in various versions of the AT&T Bell Labs American English TTS system [5].

The dictionary was aligned using *pmalign* and a pronunciation model with a 7-letter context (3 to the left and 3 to the right) was built using *pmcontext*. The alignment phase took about five minutes, and the training phase an additional five minutes on an 866 Mhz Pentium III.

A typical (pruned) tree — in this case for the letter ⟨c⟩ — is shown in Figure 1. The nodes are relatively easy to interpret: so ⟨c⟩ is /s/ before ⟨e⟩ and ⟨i⟩ (terminal node 6), /č/ mostly before ⟨h⟩ (terminal node 29), and is /ɪ/ or /ɪk/ in Gaelic-derived names beginning with ⟨mc⟩ (terminal nodes 10 and 11).

As we discussed in Section 2, one can use the prediction model output by *pmcontext* as a map in *pmalign* to realign the data. In general, one would expect to get better alignments that way, given that the pronunciation model makes more intelligent use of the context around an input symbol. One can then use this new alignment to retrain, and so forth. The results of a few iterations of this process, measured on 1,000 held-out words, are shown in Figure 2, where one can observe a small, but steady decrease in phoneme error rate over the first few iterations.

6. Future Work

While we have focussed here on models of grapheme-to-phoneme transduction, *pmtools* can in principle be used for modeling any string-to-string transduction. The only requirement is that reasonable decisions can be made on the basis of local evidence (which characterizes most approaches to this prob-

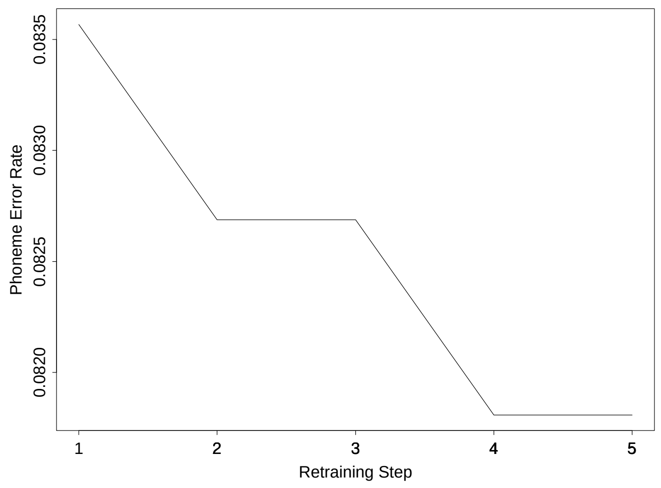

Figure 2: Results, on 1,000 held-out words, of iteratively retraining the tree model, using the resultant tree to realign the data, and then retrain. This affords a slight (0.18%) reduction in phoneme error rate.

lem), and that one can provide a reasonable small featural decomposition of the input symbols.

One major limitation in the current version of *pmtools* is that only features of the input context are available. As has been shown elsewhere (e.g. [15, 16]), it is often useful to be able to base one's prediction not only on the input context, but also on the previous output. We are planning to extend the tools to cover this in the near future.

Finally, while *pmtools* is currently only set up to use CART as a training paradigm, we envision providing a suite of possible tools so that one could train a variety of different models on the same data, and select the one that generalizes best.

7. Acknowledgments

I wish to thank Michael Riley for many discussions of this topic over many years.

8. References

[1] Adamson, M., and Damper, R. A recurrent network that learns to pronounce English text. In *Proceedings of the Fourth International Conference on Spoken Language Processing* (Philadelphia, PA, 1996), vol. 3, ICSLP, pp. 1704–1707.

[2] Black, A., Lenzo, K., and Pagel, V. Issues in building general letter to sound rules. In *Proceedings of the Third ESCA Workshop on Speech Synthesis* (Jenolan Caves, NSW, Australia, 1998), p. 1998.

[3] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. *Classification and Regression Trees*. Wadsworth & Brooks, Pacific Grove CA, 1984.

[4] Chen, F. Identification of contextual factors for pronunciation networks. In *Proceedings of ICASSP* (1990), p. S14.9.

[5] Coker, C., Church, K., and Liberman, M. Morphology and rhyming: Two powerful alternatives to letter-to-sound rules for speech synthesis. In *Proceedings of the ESCA Workshop on Speech Synthesis* (Autrans, France, 1990), G. Bailly and C. Benoit, Eds., ESCA, pp. 83–86.

[6] Daelemans, W., and van den Bosch, A. Language-independent data-oriented grapheme-to-phoneme conversion. In *Progress in Speech Synthesis*, J. van Santen, R. Sproat, J. Olive, and J. Hirschberg, Eds. Springer, New York, NY, 1997, pp. 77–89.

[7] Dedina, M., and Nusbaum, H. PRONOUNCE: a program for pronunciation by analogy. *Computer Speech and Language 5* 1996, 55–64.

[8] Fostler-Lussier, E. Multi-level decision trees for static and dynamic pronunciation models. In *Proceedings of Eurospeech 99* (1999), ISCA.

[9] Golding, A. *Pronouncing Names by a Combination of Rule-Based and Case-Based Reasoning*. PhD thesis, Stanford University, 1991.

[10] Jansche, M. Re-engineering letter-to-sound rules. In *Annual Meeting of the North American Association for Computational Linguistics* (Pittsburgh, PA, 2001).

[11] Luk, R., and Damper, R. Stochastic phonographic transduction for English. *Computer Speech and Language 10* 1996, 133–153.

[12] Mohri, M., Pereira, F., and Riley, M. Rational design for a weighted finite-state transducer library. *Lecture Notes in Computer Science 1436* 1998.

[13] Pereira, F., and Riley, M. Speech recognition by composition of weighted finite automata. In *Finite-State Language Processing*, E. Roche and Y. Schabes, Eds. MIT Press, Cambridge, MA, 1997.

[14] Randolph, M. A data-driven method for discovering and predicting allophonic variation. In *Proceedings of ICASSP* (1990), p. S14.10.

[15] Riley, M. A statistical model for generating pronunciation networks. In *Proceedings of the Speech and Natural Language Workshop* (1991), DARPA, Morgan Kaufmann, p. S11.1.

[16] Riley, M. Tree-based modeling for speech synthesis. In *Talking Machines: Theories, Models, and Designs*, G. Bailly and C. Benoit, Eds. Elsevier, Amsterdam, 1992, pp. 265–273.

[17] Riley, M., Byrne, W., Finke, M., Khudanpur, S., Ljolje, A., McDonough, J., Nock, H., Saraclar, M., Wooters, C., and Zavaliagkos, G. Stochastic pronunciation modelling from hand-labelled phonetic corpora. *Speech Communication 29* 1999, 209–224.

[18] Sejnowski, T., and Rosenberg, C. Parallel networks that learn to pronounce English text. *Complex Systems 1* 1987, 145–168.

[19] Sproat, R., Ed. *Multilingual Text to Speech Synthesis: The Bell Labs Approach*. Kluwer Academic Publishers, Boston, MA, 1997.

[20] Sproat, R., and Riley, M. Compilation of weighted finite-state transducers from decision trees. In *Association for Computational Linguistics, 34th Annual Meeting* (Santa Cruz, CA, 1996), Association for Computational Linguistics, pp. 215–222.

[21] van den Bosch, A. *Learning to pronounce written words. A study in inductive language learning*. PhD thesis, University of Maastricht, 1997.