Indexing Weighted Lattices

Andrew Fowler

Topics in IR

11/6/2012

Spoken Document Retrieval (SDR)

- Same end goal as standard document retrieval
- Big caveat: Data is not stored as text
- Made possible recently by the presence of cheap storage and large audio corpora
- Existing IR techniques can still work, with modification (inverted indices)
- Appeared as a track in several TRECs

A Possible Easy Solution

- Just take the **1-best ASR output**
- Works well for low-WER scenarios (broadcast news, etc.)
- Performance plummets (i.e. low recall) in high-noise environments (teleconference speech, recorded meetings, etc.)
- We have to consider the whole lattice

A Better, More Difficult Solution

- Can we expand **every** possible path through each ASR lattice, then do standard IR on that text? No.
  - Computationally infeasible; explodes fast
  - Does not preserve weight information
- But, we can still create an index of the entire ASR lattice
  - Not too big
  - Weight information kept

What Do We Want?

- Given a query **x**, we want to quickly get
  - The *doc id of every lattice* in which x is a potential realization
  - The *relative weights* of each of these hits (useful for pruning, etc.)
  - Efficient handling of OOVs
- In other words: a more sophisticated, lattice-aware index which is robust to high-WER environments

Paper 1: Saraclar & Sproat (2004)

- **Core idea**: Create an *extended index* that keeps more info than just doc id
- Given an arc *a* with label *l* in an ASR lattice, the index keeps the following:
  - Lattice number *L[a]* (equivalent to doc id)
  - Input-state *k[a]* of arc
  - Probability mass *f(k[a])* leading to input state
  - Probability *p(a|k[a])* of arc itself
  - Index of next state

Paper 1: Using the index

- To get **documents**, just look up all lattices in the (inverted) index for the given query/label
- To get **weights**, we calculate probabilities from the lattice *by arc*
  - Normalize all lattices with weight-pushing first (ensures probability of set of all paths from any arc to the final sums to 1)

Paper 1: Index (continued)

- Calculate the probability "of the set of all paths containing that arc"

$$p(a) = \sum_{\pi \in L : a \in \pi} p(\pi) = f(k[a])p(a|k[a])$$

- Construct a "count" *C(l|L)* for a given label *l* in a lattice *L*

$$\sum_{a \in \mathcal{I}(l) : L[a] = L} f(k[a])p(a|k[a])$$

Paper 1: Index (continued)

  - We can use this "count" to threshold our results (think of it as a "lattice-based confidence measure")
- To search for multi-label expressions:
  - Seek for each label in the expression individually
  - For each boundary between labels, **join** output states of first word with input states of second word
- Note: Special case of unweighted single path lattice results in a **traditional inverted index**

Paper 2: Allauzen et al. (2004)

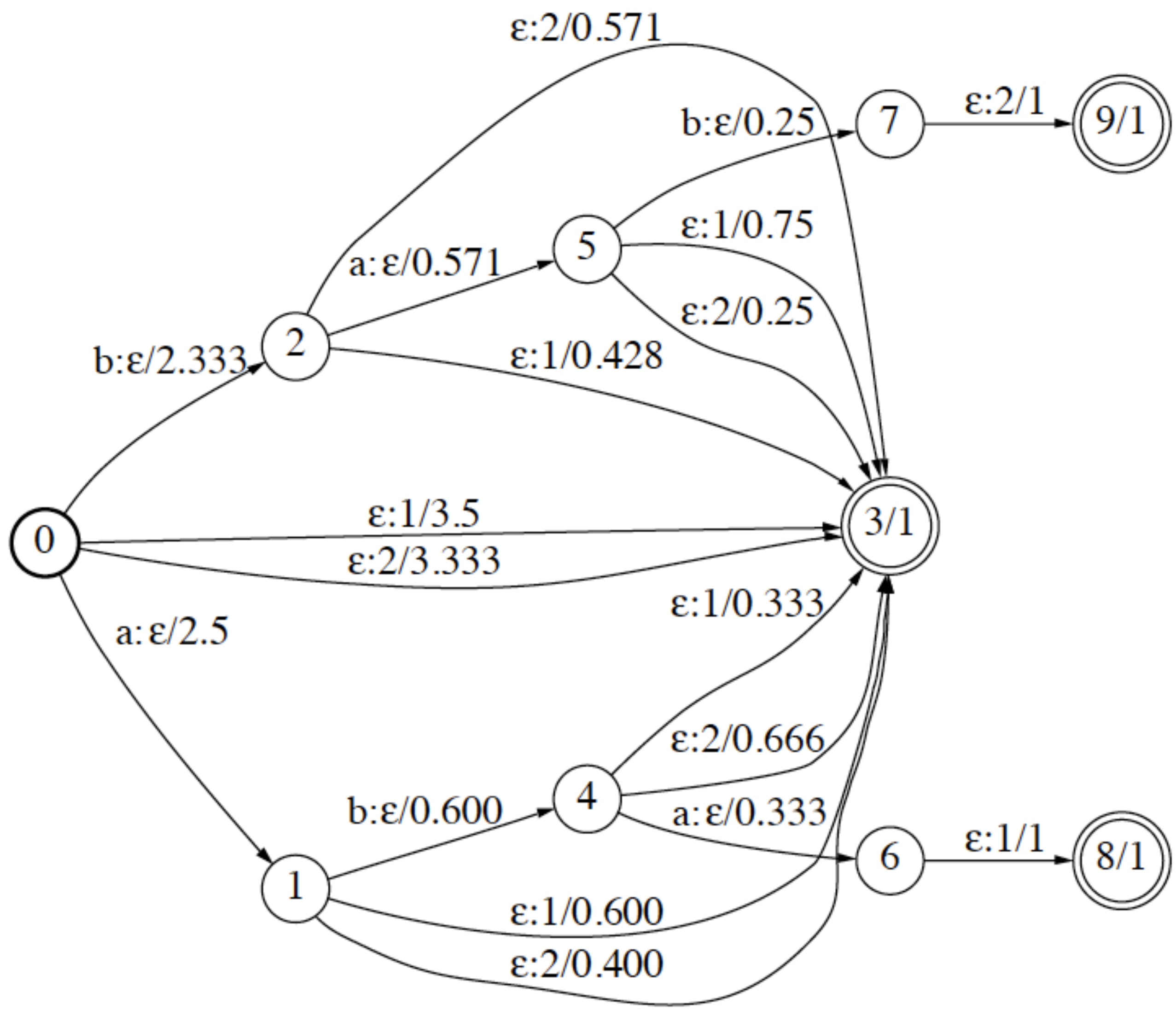
- Generalization of Paper 1 to indexing **any** weighted automaton
- Allows for more sophisticated queries (boolean, regular expressions, etc.)
- **Core idea**: The *index itself is an FST* that outputs the matching document IDs and the weights/ scores of the matches
- Given search factor x, index emits all indices where x appears, and the negative log of the expected count of x

Paper 2 Method

- For each “document” with weighted automaton A:
  - Use weight pushing to convert A to B
  - Transform all arcs to emit epsilon
  - Create new unique start state
  - Create new unique final state
  - Add new transitions from new start state to original states (new weight: shortest distance)
  - Add new transitions from original states to new final state (new weight: shortest distance)

Paper 2 Method (continued)

- Optimize the new transducer (weighted epsilon removal, weighted determinization, minimization over log semiring
- Take the union of all FSTs thus created (one for each document)
- Determinize (ensures unique initial state and no two transitions leaving a state have same input label)


ε:2/0.571
b:ε/0.25
7
ε:2/1
9/1
5
ε:1/0.75
a:ε/0.571
ε:2/0.25
b:ε/2.333
2
ε:1/0.428
ε:1/3.5
0
3/1
ε:2/3.333
ε:1/0.333
a:ε/2.5
ε:2/0.666
4
b:ε/0.600
a:ε/0.333
6
ε:1/1
8/1
1
ε:1/0.600
ε:2/0.400

Paper 2 Method (running a query)

- Read query **x** into the machine
- Return set of all transitions with input epsilon
- This set contains all documents/automata where x is found, and their associated weights

Paper 2 Advantages

- All expressed in terms of automata, so you can use existing, highly optimized algorithms
- More sophisticated queries are possible; anything expressible by an arbitrary weighted automaton
  - Boolean queries
  - Certain regular expressions
  - Just take the two weighted transducers and perform composition

Paper 2 Advantages

- General framework means highly flexible:
  - Pronunciation dictionary (OOVs)
  - Vocabulary restriction
    - Ignore some words, i.e. non named entities
  - Reweighted (TF-IDF)
  - Classification (labeling)
  - Length restriction (more on this later)
- Shown to be equivalent to Paper 1 for that specific task, but much more generalizable (also potentially smaller footprint with pruning)

Paper 3: Chelba & Acero (2005)

- **Core idea**: Traditional TD-IDF assumes independence of terms. This is not true.
- Word positions, and therefore N-grams, are important
- A “hit” currently might look like this:
  `(doc id, position)`
- But we really need this:
  `(doc id, position, posterior probability)`

Paper 3 Method

- Since these “soft hits” are from inside a lattice, computing the posterior probability is the hard part
- Turns out you can use a modified forward-backward algorithm to sum over all identical words in the same position
- The resulting structure is called a Position Specific Posterior Lattice (PSPL)

Paper 3 Method (continued)

- Separate documents into segments
- Create a PSPL for each segment
- Given a query, count 1-grams up to N-grams
- Build an inverted index from every document/ segment

Paper 3 Discussion

- Outperforms 1-best, like other systems
- Lossy (?)
- Query corpus somewhat arbitrary
- Not obvious from paper (to me) how to quickly accumulate the n-gram counts
- N-gram weights arbitrary

The OOV problem and sub-word units

- Since OOVs can be common in ASR systems, using word-based lattices exclusively is not optimal
- Phone-based lattices exist, but have worse overall performance
- Solution: Combine them. Three ways:
  1. Create index for words and phones, search separately then combine results
  2. Use word index for in-vocabulary, phone index for OOVs
  3. Use phone index only if/after word index returns zero results

Problems with length

- Words with short pronunciations in the phone index (when used) can result in false alarms
  - Solution: filter out queries with too few phones
  - Results in better precision
- We can also impose limits on the maximum string lengths in the index
  - Allows for smaller indices
  - May degrade performance for long queries
  - Search procedures may require modification

Questions