Finding Good Sequential Model Structures using Output Transformations

Edward Loper
Department of Computer & Information Science
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104
edloper@cis.upenn.edu

Abstract

In Sequential Viterbi Models, such as HMMs, MEMMs, and Linear Chain CRFs, the type of patterns over output sequences that can be learned by the model depend directly on the model's structure: any pattern that spans more output tags than are covered by the models' order will be very difficult to learn. However, increasing a model's order can lead to an increase in the number of model parameters, making the model more susceptible to sparse data problems.

This paper shows how the notion of *output transformation* can be used to explore a variety of alternative model structures. Using output transformations, we can selectively increase the amount of contextual information available for some conditions, but not for others, thus allowing us to capture longer-distance consistencies while avoiding unnecessary increases to the model's parameter space. The appropriate output transformation for a given task can be selected by applying a hill-climbing approach to held-out data. On the NP Chunking task, our hill-climbing system finds a model structure that outperforms both first-order and second-order models with the same input feature set.

1 Sequence Prediction

A *sequence prediction task* is a task whose input is a sequence and whose output is a corresponding sequence. Examples of sequence prediction tasks include part-of-speech tagging, where a sequence of words is mapped to a sequence of part-of-speech tags; and IOB noun phrase chunking, where a sequence of words is mapped to a sequence of labels, `I`, `O`, and `B`, indicating whether each word is inside a chunk, outside a chunk, or at the boundary between two chunks, respectively.

In sequence prediction tasks, we are interested in finding the most likely output sequence for a given input. In order to be considered likely, an output value must be consistent with the input value, but it must also be internally consistent. For example, in part-of-speech tagging, the sequence "preposition-verb" is highly unlikely; so we should reject an output value that contains that sequence, even if the individual tags are good candidates for describing their respective words.

2 Sequential Viterbi Models

This intuition is captured in many sequence learning models, including Hidden Markov Models (HMMs), Maximum Entropy Markov Models (MEMMs), and Linear Chain Conditional Random Fields (LC-CRFs), by including terms corresponding to pieces of output structure in their scoring functions. (Sha and Pereira, 2003; Sutton and McCallum, 2006; McCallum et al., 2000; Alpaydin, 2004)

Each of these *Sequential Viterbi Models* defines a set of scoring functions that evaluate fixed-size pieces of the output sequence based on fixed-size pieces of the input sequence.[1] The overall score for

[1]For HMMs and MEMMs, the local scores are negative log probabilities. For LC-CRFs, the local scores do not have any direct probabilistic interpretation.

Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pp. 801–809, Prague, June 2007.

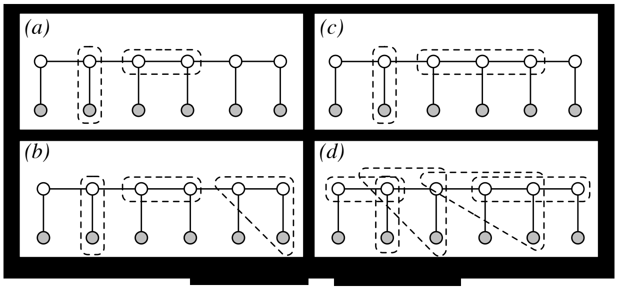

Figure 1: **Common Model Structures**. *(a)* Simple first order. *(b)* Extended first order. *(c)* Simple second order. *(d)* Extended second order.

an output value is then computed by summing the scores for all its fixed-size pieces. Sequence prediction models can differ from one another along two dimensions:

1. **Model Structure:** The set of output pieces and input pieces for which local scoring functions are defined.

2. **Model Type:** The set of parametrized equations used to define those local scoring functions, and the procedures used to determine their parameters.

In this paper, we focus on model structure. In particular, we are interested in finding a suitable model structure for a given task and training corpus.

2.1 Common Model Structures

The model structure used by classical HMMs is the "simple first order" structure. This model structure defines two local scoring functions. The first scoring function evaluates an output value in the context of the corresponding input value; and the second scoring function evaluates adjacent pairs of output values. Simple LC-CRFs often extend this structure by adding a third local scoring function, which evaluates adjacent pairs of output values in the context of the input value corresponding to one of those outputs. These model structures are illustrated in Figure 1.

Because these first order structures include scoring functions for adjacent pairs of output items, they can identify and reject output values that contain improbable subsequences of length two. For example, in part-of-speech tagging, the sequence "preposition-verb" is highly unlikely; and such models will easily learn to reject outputs containing that sequence. However, it is much more difficult for first order models to identify improbable subsequences of length three or more. For example, in English texts, the sequence "verb-noun-verb" is much less likely than one would predict based just on the subsequences "verb-noun" and "noun-verb." But first order models are incapable of learning that fact.

Thus, in order to improve performance, it is often necessary to include scoring functions that span over larger sequences. In the "simple second order" model structure, the local scoring function for adjacent pairs of output values is replaced with a scoring function for each triple of consecutive output values. In extended versions of this structure typically used by LC-CRFs, scoring functions are also added that combine output value triples with an input value. These model structures are illustrated in Figure 1. Similarly, third order and and fourth order models can be used to further increase the span over which scoring functions are defined.

Moving to higher order model structures increases the distance over which the model can check consistency. However, it also increases the number of parameters the model must learn, making the model more susceptible to sparse data problems. Thus, the usefulness of a model structure for a given task will depend on the types of constraints that are important for the task itself, and on the size and diversity of the training corpus.

3 Searching for Good Model Structures

We can therefore use simple search methods to look for a suitable model structure for a given task and training corpus. In particular, we have performed several experiments using hill-climbing methods to search for an appropriate model structure for a given task. In order to apply hill-climbing methods, we need to define:

1. The search space. I.e., concrete representations for the set of model structures we will consider.

2. A set of operations for moving through that search space.

3. An evaluation metric.

In Section 4, we will define the search space using transformations on output values. This will allow us to consider a wide variety of model structures without needing to make any direct modifications to the underlying sequence modelling systems. Output value transformations will be concretely represented using Finite State Transducers (FSTs). In Section 5, we will define the set of operations for moving through the search space as modification operations on FSTs. For the evaluation metric, we simply train and test the model, using a given model structure, on held-out data.

4 Representing Model Structure with Reversible Output Transformations

The common model structures described in Section 2.1 differ from one another in that they examine varying sizes of "windows" on the output structure. Rather than varying the size of the window, we can achieve the same effect by fixing the window size, but transforming the output values. For example, consider the effects of transforming the output values by replacing individual output tags with pairs of adjacent output tags:

$$y_1, y_2, \ldots, y_t \rightarrow$$
$$\langle \mathrm{START}, y_1 \rangle, \langle y_1, y_2 \rangle, \langle y_2, y_3 \rangle, \ldots, \langle y_{t-1}, y_t \rangle$$

E.g.:

```
  I   O   O   I   I   B   I  →
 OI  IO  OO  OI  II  IB  BI
```

Training a first order model based on these transformed values is equivalent to training a second order model based on the original values, since in each case the local scoring functions will be based on pairs of adjacent output tags. Similarly, transforming the output values by replacing individual output tags with triples of adjacent output tags is equivalent to training a third order model based on the original output values.

Of course, when we apply a model trained on this type of transformed output to new inputs, it will generate transformed output values. Thus, the transformation must be reversible, so that we can map the output of the model back to an un-transformed output value.

This transformational approach has the advantage that we can explore different model structures using off-the-shelf learners, without modifying them. In particular, we can apply the transformation corresponding to a given model structure to the training corpus, and then train the off-the-shelf learner based on that transformed corpus. To predict the value for a new input, we simply apply the learned model to generate a corresponding transformed output value, and then use the inverse transformation to map that value back to an un-transformed value.

Output encoding transformations can be used to represent a large class of model structures, including commonly used structures (first order, second order, etc) as well as a number of "hybrid" structures that use different window sizes depending on the content of the output tags.

Output encoding transformations can also be used to represent a wide variety of other model structures. For example, there has been some debate about the relative merits of different output encodings for the chunking task (Tjong Kim Sang and Veenstra, 1999; Tjong Kim Sang, 2000; Shen and Sarkar, 2005). These encodings differ in whether they define special tags for the beginning of chunks, for the ends of chunks, and for boundaries between chunks. The output transformation procedure described here is capable of capturing all of the output encodings used for chunking. Thus, this transformational method provides a unified framework for considering both the type of information that should be encoded by individual tags (i.e., the encoding) and the distance over which that information should be evaluated (i.e., the order of the model). Under this framework, we can use simple search procedures to find an appropriate transformation for a given task.

4.1 Representing Transformations as FSTs

Finite State Transducers (FSTs) provide a natural formalism for representing output transformations. FSTs are powerful enough to capture different orders of model structure, including hybrid orders; and to capture different output encodings, such as the ones considered in (Shen and Sarkar, 2005). FSTs are efficient, so they add very little overhead. Finally, there exist standard algorithms for inverting

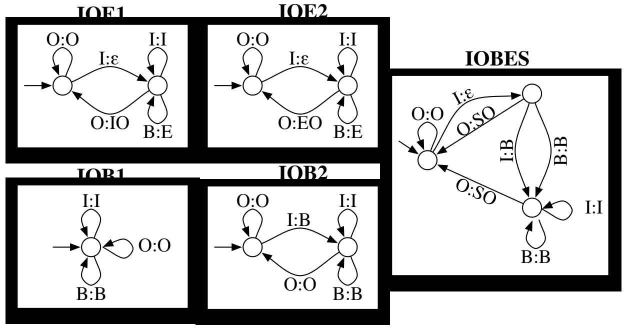

Figure 2: **FSTs for Five Common Chunk Encodings**. Each transducer takes an `IOB1`-encoded string for a given output value, and generates the corresponding string for the same output value, using a new encoding. Note that the `IOB1` FST is an identity transducer; and note that the transducers that make use of the `E` tag must use ϵ-output edges to delay the decision of which tag should be used until enough information is available.

and determinizing FSTs. [2]

4.1.1 Necessary Properties for Output-Transformation FSTs

In order for an FST to be used to transform output values, it must have the following three properties:

1. The FST's inverse should be deterministic.[3] Otherwise, we will be unable to convert the model's (transformed) output into an untransformed output value.

2. The FST should recognize exactly the set of valid output values. If it does not recognize some valid output value, then it won't be able to transform that value. If it recognizes some invalid output value, then there exists an transformed output value that would map back to an invalid output value.

3. The FST should not modify the length of the output sequence. Otherwise, it will not be possible to align the output values with input values when running the model.

In addition, it seems desirable for the FST to have the following two properties:

4. The FST should be deterministic. Otherwise, a single training example's output could be encoded in multiple ways, which would make training the individual base decision classifiers difficult.

5. The FST should generate every output string. Otherwise, there would be some possible system output that we are unable to map back to an un-transformed output.

Unfortunately, these two properties, when taken together with the first three, are problematic. To see why, assume an FST with an output alphabet of size k. Property (5) requires that all possible output strings be generated, and property (1) requires that no string is generated for two input strings, so the number of strings generated for an input of length n must be exactly k^n. But the number of possible chunkings for an input of length n is $3^n - 3^{n-1} - 3^{n-2}$; and there is no integer k such that $k^n = 3^n - 3^{n-1} - 3^{n-2}$.[4]

We must therefore relax at least one of these two properties. Relaxing the property 4 (deterministic FSTs) will make training harder; and relaxing the property 5 (complete FSTs) will make testing harder. In the experiments presented here, we chose to relax the second property.

4.1.2 Inverting the Transformation

Recall that the motivation behind property 5 is that we need a way to map *any* output generated by the machine learning system back to an untransformed output value.

As an alternative to requiring that the FST generate every output string, we can define an extended inversion function, that includes the inverted FST, but also generates output values for transformed values that are not generated by the FST. In particular,

[2]Note that we are not attempting to learn a transducer that generates the output values from input values, as is done in e.g. (Oncina et al., 1993) and (Stolcke and Omohundro, 1993). Rather, we we are interested in finding a transducer from one output encoding to another output encoding that will be more amenable to learning by the underlying Sequential Viterbi Model.

[3]Or at least determinizable.

[4]To see why the number of possible chunkings is $3^n - 3^{n-1} - 3^{n-2}$, consider the IOB1 encoding: it generates all chunkings, and is valid for any of the 3^n strings except those that start with `B` (of which there are 3^{n-1}) and those that include the sequence `OB` (of which there are 3^{n-2}).

in cases where the transformed value is not generated by the FST, we can assume that one or more of the transformed tags was chosen incorrectly; and make the minimal set of changes to those tags that results in a string that *is* generated by the FST. Thus, we can compute the optimal un-transformed output value corresponding to each transformed output using the following procedure:

1. Invert the original FST. I.e., replace each arc $\langle S \rightarrow Q[\alpha : \beta] \rangle$ with an arc $\langle S \rightarrow Q[\beta : \alpha] \rangle$.
2. Normalize the FST such that each arc has exactly one input symbol.
3. Convert the FST to a weighted FST by assigning a weight of zero to all arcs. This weighted FST uses non-negative real-valued weights, and the weight of a path is the sum of the weights of all edges in that path.
4. For each arc $\langle S \rightarrow Q[x : \alpha] \rangle$, and each $y \neq x$, add a new arc $\langle S \rightarrow Q[y : \alpha] \rangle$ with a weight one.
5. Determinize the resulting FST, using a variant of the algorithm presented in (Mohri, 1997). This determinization algorithm will prune paths that have non-optimal weights. In cases where determinization algorithm has not completed by the time it creates 10,000 states, the candidate FST is assumed to be non-determinizable, and the original FST is rejected as a candidate.

The resulting FST will accept all sequences of transformed tags, and will generate for each transformed tag the un-transformed output value that is generated with the fewest number of "repairs" made to the transformed tags.

5 FST Modification Operations

In order to search the space of output-transforming FSTs, we must define a set of modification operations, that generate a new FST from a previous FST. In order to support a hill-climbing search strategy, these modification operations should make small incremental changes to the FSTs. The selection of appropriate modification operations is important, since it will significantly impact the efficiency of the search process. In this section, I describe the set of FST modification operations that are used for the experiments described in this paper. These operations were chosen based our intuitions about what modifications would support efficient hill-climbing search. In future experiments, we plan to examine alternative modification operations.

5.1 New Output Tag

The *new output tag* operation replaces an arc $\langle S \rightarrow Q[\alpha : \beta x \gamma] \rangle$ with an arc $\langle S \rightarrow Q[\alpha : \beta y \gamma] \rangle$, where y is a new output tag that is not used anywhere else in the transducer. When a single output tag appears on multiple arcs, this operation effectively splits that tag in two. For example, when applied to the identity transducer for the IOB1 encoding shown in Figure 2, this operation can be used to distinguish O tags that follow other O tags from O tags that follow I or B tags – effectively increasing the order of the model structure for just O tags.

5.2 Specialize Output Tag[5]

The *specialize output tag* operation is similar to the *new output tag* operation, but rather than replacing the output tag with a new tag, we "subdivide" the tag. When the model is trained, features will be included for both the subdivided tag and the original (undivided) tag.

5.3 Loop Unrolling

The *loop unrolling* operation acts on a single self-loop arc e at a state S, and makes the following changes to the FST:

1. Create a new state S'.
2. For each outgoing arc $e_1 = \langle S \rightarrow Q[\alpha : \beta] \rangle \neq e$, add add an arc $e_2 = \langle S' \rightarrow Q[\alpha : \beta] \rangle$. Note that if e_1 was a self-loop arc (i.e., $S = Q$), then e_2 will point from S' to S.
3. Change the destination of loop arc e from S to S'.

By itself, the *loop unrolling* operation just modifies the structure of the FST, but does not change

[5] This operation requires the use of a model where features are defined over (input,output) pairs, such as MEMMs or LC-CRFs.

the actual transduction performed by the FST. It is therefore always immediately followed by applying the *new output tag* operation or the *specialize output tag* operation to the loop arc e.

5.4 Copy Tag Forward

The *copy tag forward* operation splits an existing state in two, directing all incoming edges that generate a designated output tag to one copy, and all remaining incoming edges to the other copy. The outgoing edges of these two states are then distinguished from one another, using either the *specialize output tag* operation (if available) or the *new output tag* operation.

This modification operation creates separate edges for different output histories, effectively increasing the "window size" of tags that pass through the state.

5.5 Copy State Forward

The *copy state forward* operation is similar to the *copy tag forward* operation; but rather than redirecting incoming edges based on what output tags they generate, it redirects incoming edges based on what state they originate from. This modification operation allows the FST to encode information about the history of states in the transformational FST as part of the model structure.

5.6 Copy Feature Forward

The *copy feature forward* operation is similar to the *copy tag forward* operation; but rather than redirecting incoming edges based on what output tags they generate, it redirects incoming edges based on a feature of the current input value. This modification operation allows the transformation to subdivide output tags based on features of the input value.

6 Hill Climbing System

Having defined a search space, a set of transformations to explore that space, and an evaluation metric, we can use a hill-climbing system to search for a good model structure. This approach starts with a simple initial FST, and makes incremental local changes to that FST until a locally optimal FST is found. In order to help avoid sub-optimal local maxima, we use a fixed-size beam search. To increase the search speed, we used a 12-machine cluster to evaluate candidate FSTs in parallel. The hill climbing system iteratively performs the following procedure:

1. Initialize `candidates` to be the singleton set containing the identity transducer.

2. Repeat ...

   (a) Generate a new FST, by applying a random modification operation to a randomly selected member of the `candidates` set.

   (b) Evaluate the new FSTs, and test its performance on the held-out data set. (This is done in parallel.)

   (c) Once the FST has been evaluated, add it to the `candidates` set.

   (d) Sort the `candidates` set by their score on the held-out data, and discard all but the 10 highest-scoring candidates.

   ... until no improvement is made for twenty consecutive iterations.

3. Return the candidate FST with the highest score.

7 Noun Phrase Chunking Experiment

In order to test this approach to finding a good model structure, we applied our hill-climbing system to the task of noun phrase chunking. The base system was a Linear Chain CRF, implemented using Mallet (McCallum, 2002). The set of features used are listed in Figure 1. Training and testing were performed using the noun phrase chunking corpus described in Ramshaw & Marcus (1995) (Ramshaw and Marcus, 1995). A randomly selected 10% of the original training corpus was used as held-out data, to provide feedback to the hill-climbing system.

7.1 NP Chunking Experiment: Results

Over 100 iterations, the hill-climbing system increased chunking performance on the held-out data from a F-score of 94.93 to an F-score of 95.32. This increase was reflected in an improvement on the test data from an F-score of 92.48 to an F-score

Feature	Description
y_i	The current output tag.
y_i, w_{i+n}	A tuple of the current output tag and the $i+n$th word, $-2 \leq n \leq 2$.
y_i, w_i, w_{i-1}	A tuple of the current output tag, the current word, and the previous word.
y_i, w_i, w_{i+1}	A tuple of the current output tag, the current word, and the next word.
y_i, t_{i+n}	A tuple of the current output tag and the part of speech tag of the $i+n$th word, $-2 \leq n \leq 2$.
y_i, t_{i+n}, t_{i+n+1}	A tuple of the current output tag and the two consecutive part of speech tags starting at word $i+n$, $-2 \leq n \leq 1$.
$y_{i+n-1}, t_{i+n}, t_{i+n+1}$	A tuple of the current output tag, and three consecutive part of speech tags centered on word $i+n$, $-1 \leq n \leq 1$.

Table 1: **Feature Set for the CRF NP Chunker**. y_i is the i^{th} output tag; w_i is the i^{th} word; and t_i is the part-of-speech tag for the i^{th} word.

System	F_1 (Held-out)	F_1 (Test)
Baseline (first order)	94.93	92.48
Second order	95.14	92.63
Learned structure	95.32	92.80

Table 2: **Results for NP Chunking Experiment**.

of 92.80.[6] As a point of comparison, a simple second order model achieves an intermediate F-score of 92.63 on the test data. Thus, the model learned by the hill-climbing system outperforms both the simple first-order model and the simple second-order model.

Figure 3 shows how the scores of FSTs on held-out data changed as the hill-climbing system ran. Figure 4 shows the search tree explored by the hill-climbing system.

[6]The reason that held-out scores are significantly higher than test scores is that held-out data was taken from the same sections of the original corpus as the training data; but test data was taken from new sections. Thus, there was more lexical overlap between the training data and the held-out data than between the training data and the testing data.

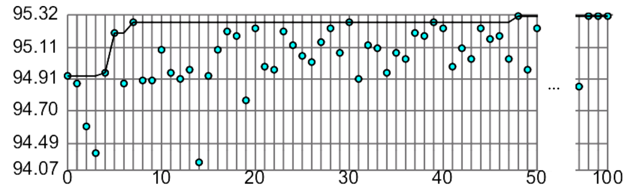

Figure 3: **Performance on Heldout Data for NP Chunking Experiment**. In this graph, each point corresponds to a single transducer generated by the hill-climbing system. The height of each transducer's point indicates its score on held-out data. The line indicates the highest score that has been achieved on the held-out data by any transducer.

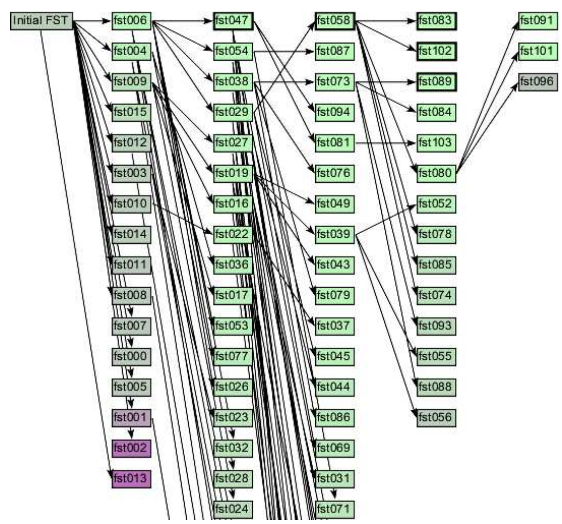

Figure 4: **Hill Climbing Search Tree for NP Chunking Experiment** This tree shows the "ancestry" of each transducer tried by the hill climbing system. Lighter colors indicate higher scores on the held-out data. After one hundred iterations, the five highest scoring transducers were `fst047`, `fst058`, `fst083`, `fst102`, and `fst089`.

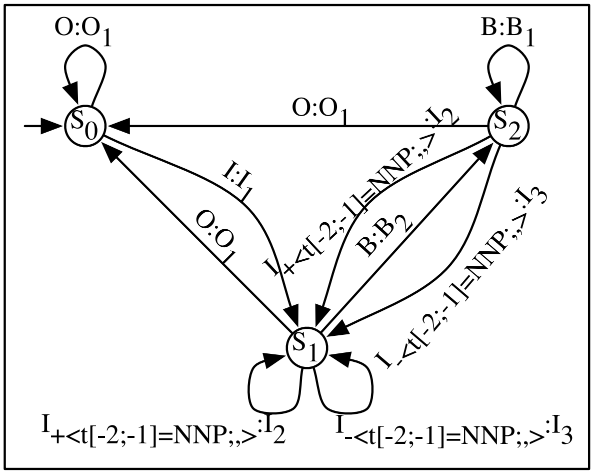

Figure 5: **Final FST**. The highest-scoring FST generated by the hill-climbing algorithm, after a run of 100 iterations. For a discussion of this transducer, see Section 7.1.1.

7.1.1 NP Chunking Experiment: The Selected Transformation

Figure 5 shows the FST for the best output transformation found after 100 iterations of the hill-climbing algorithm. Inspection of this FST reveals that it transforms the original set of three tags (I, O, and B) to six new tags: I_1, I_2, I_3, O, B_1, and B_2.

The first three of these tags are used at the beginning of a chunk: I_1 is used if the preceding tag was O; B_1 is used if the preceding tag was B; and B_2 is used if the preceding tag was I. This is similar to a second order model, in that it records information about both the current tag and the previous tag.

The next tag, O, is used for all words outside of chunks. Thus, the hill-climbing system found that increasing the window size used for O chunks does not help to learn any useful constraints with neighboring tags.

Finally, two tags are used for words that are inside a chunk, but not at the beginning of the chunk: I_2 and I_3. The choice of which tag should be used depends on the input feature that tests whether the current word is a comma, and the previous word was a proper noun (NNP). At first, this might seem like an odd feature to distinguish. But note that in the Wall Street Journal, it is quite common for proper nouns to include internal commas; but for other nouns, it is fairly uncommon. By dividing the I tag in two based on this feature, the model can use separate distributions for these two cases. Thus, the model avoids conflating two contexts that are significantly different from one another for the task at hand.

8 Discussion

Sequential Viterbi Models are capable of learning to model the probability of local patterns on the output structure. But the distance that these patterns can span is limited by the model's structure. This distance can be lengthened by moving to higher order model structures, but only at the expense of an increase in the number of model parameters, along with the data sparsity issues that can arise from that increase. Therefore, it makes sense to be more selective about how we extend the model structure. Using reversible output transformations, it is possible to define model structures that extend the reach of the model only where necessary. And as we have shown here, it is possible to find a suitable output transformation for a given task by using simple search procedures.

9 Acknowledgements

We gratefully acknowledge the support of the National Science Foundation Grant NSF-0415923, Word Sense Disambiguation, the DTO-AQUAINT NBCHC040036 grant under the University of Illinois subcontract to University of Pennsylvania 2003-07911-01, NSF-ITR-0325646: Domain-Independent Semantic Interpretation, and Defense Advanced Research Projects Agency (DARPA/IPTO) under the GALE program, DARPA/CMO Contract No. HR0011-06-C-0022.

References

Ethem Alpaydin, 2004. *Introduction to Machine Learning*, chapter 7. The MIT Press.

Andrew McCallum, Dayne Freitag, and Fernando Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. In *Proc. 17th International Conf. on Machine Learning*, pages 591–598. Morgan Kaufmann, San Francisco, CA.

Andrew Kachites McCallum. 2002. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

José Oncina, Pedro García, and Enrique Vidal. 1993. Learning subsequential transducers for pattern recognition tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:448–458, May.

Lance Ramshaw and Mitch Marcus. 1995. Text chunking using transformation-based learning. In David Yarowsky and Kenneth Church, editors, *Proceedings of the Third Workshop on Very Large Corpora*, pages 82–94, Somerset, New Jersey. Association for Computational Linguistics.

Fei Sha and Fernando Pereira. 2003. Shallow parsing with conditional random fields. In *Proceedings of HLT-NAACL*, pages 134–141.

Hong Shen and Anoop Sarkar. 2005. Voting between multiple data representations for text chunking. In *Advances in Artificial Intelligence: 18th Conference of the Canadian Society for Computational Studies of Intelligence*, May.

Andreas Stolcke and Stephen Omohundro. 1993. Hidden Markov Model induction by Bayesian model merging. In C. L. Giles, S. J. Hanson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufman, San Mateo, Ca.

Charles Sutton and Andrew McCallum. 2006. An introduction to conditional random fields for relational learning. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press. To appear.

Erik Tjong Kim Sang and Jorn Veenstra. 1999. Representing text chunks. In *Proceedings of EACL'99*, Bergen. Association for Computational Linguistics.

Erik Tjong Kim Sang. 2000. Noun phrase recognition by system combination. In *Proceedings of BNAIC*, Tilburg, The Netherlands.