Large Scale Inference of Deterministic Transductions: Tenjinno Problem 1

Alexander Clark

Department of Computer Science, Royal Holloway, University of London, Egham, Surrey TW20 0EX

Abstract. We discuss the problem of large scale grammatical inference in the context of the Tenjinno competition, with reference to the inference of deterministic finite state transducers, and discuss the design of the algorithms and the design and implementation of the program that solved the first problem. Though the OSTIA algorithm has good asymptotic guarantees for this class of problems, the amount of data required is prohibitive. We therefore developed a new strategy for inferring large scale transducers that is more adapted for large random instances of the type in question, which involved combining traditional state merging algorithms for inference of finite state automata with EM based alignment algorithms and state splitting algorithms.

1 Introduction

The Tenjinno competition [14] is a grammatical inference competition where the problem tasks, though from synthetic data, are designed to approximate the problem of machine translation. The problem consists of inferring a transduction from one symbol sequence (the input sequence) to another symbol sequence (the output sequence). Once inferred the transduction must be applied to some new input data to generate a predicted output sequence. These sequences are then submitted to a web-based oracle. If all of the output sequences are exactly correct then the oracle will say so; otherwise, the oracle merely states that the data is incorrect. No other feedback is given. The problems were generated according to some random process which was unspecified.

The first problem, which we study here, is a deterministic sequential transducer. The key factor here is simply the size of the problem; the input alphabet size was 1001, and the output alphabet was 1000. The problem clearly had many thousands of states, and the amount of training data was 100,000 pairs, with 10,000 test strings. Thus the problem is very substantial; indeed of the same size as real world problems in machine translation, though current SMT systems train on many millions or even billions of sentences.

The outline of our approach is as follows. We started by discarding the output data and considering the input data alone. This will be a regular language, and given the parameters of the problem, the distinguishability of the underlying deterministic finite state automaton would be high. We accordingly constructed the prefix tree acceptor for the data, creating an automaton with over 1,000,000

Y. Sakakibara et al. (Eds.): ICGI 2006, LNAI 4201, pp. 227–239, 2006.

states and then used a state merging algorithm, with appropriate optimisations, to construct a more compact automaton with less than 20,000 states. We then used a novel alignment algorithm based on the Expectation-Maximization algorithm to attach an output string to each transition, which also identified some errors (incorrectly merged states) which were then split, and the aligment corrected, until we had a correctly aligned transducer with 20,000 states that generated the training data correctly. Using the information now from the outputs on the transition, we could merge the automaton more freely. A final phase used the test data itself, to merge states to increase the coverage to include all of the test data.

In the rest of the paper, we start by defining our notation, Section 2, and then describe approaches to the inference of finite state models both for languages and for transductions, Section 3. We then discuss the Tenjinno problem, Section 4 and then present the algorithm that we developed to solve the problem Section 5. We discuss the alignment algorithm in Section 7 in some detail, and finish with some remarks about the competition design and the issues about applying theoretical algorithms to large scale problems.

2 Notation

We are interested in transductions between finite strings. We have a finite input alphabet Σ, and a finite output alphabet Ξ. We write Σ^* for the free monoid generated by Σ and Ξ^* similarly. We write λ for the empty string in both monoids, since this will not be confusing. We write $|u|$ for the length of a string, and uv, or occasionally $u + v$ for the concatenation of elements of Σ^* . We will use letters a, b for elements of Σ and also for elements of Σ^* of length 1. We write $u \sqsubseteq v$ if u is a contiguous subsequence (factor) of v , i.e.$\exists, l, r \in \Sigma^*$ s.t. $lur = v$. For a string u we will write u_i for the ith letter in the string, i.e. $u = u_1 u_2 \dots u_{|u|}$ we will write $u[i : j]$ for the substring $u_{i+1} \dots u_j$, so $u[i : i] = \lambda$, $u[0 : |u|] = u$.

A transduction is a subset T of $\Sigma^* \times \Xi^*$. Here we are interested in transductions that are deterministic and unambiguous. A transduction T is unambiguous if $(u, v) \in T, (u, v') \in T$ implies $v = v'$.

The transductions in the Tenjinno competition are of various formal types, problem 1 being a deterministic finite state transducer.

A deterministic finite state automaton A is a tuple $\langle Q, \Sigma, q_0, q_f, \tau \rangle$, where

- Q is a finite set of states,
- Σ is the alphabet, a finite set of symbols,
- $q_0 \in Q$ is the single initial state,
- $q_f \notin Q$ is the final state,
- $\tau : Q \times \Sigma \to Q \cup \{q_f\}$ is a partial function that defines the transitions.

We extend the transition function τ to Σ^* in the normal way. This automaton then defines a language $L(A) = \{u \in \Sigma^* | \tau(q_0, u) = q_f\}$. The definition we use here means that the language is prefix free: i.e. if $u \in L(A)$ and $uv \in L(A)$ then $v = \lambda$.

A key idea here is that of the signature of a state. The signature of a state $q \in Q$ is defined as $\text{sig}(q) = \{\sigma \in \Sigma | \exists q' \in Q : \tau(q, \sigma) = q'\}$ - i.e. the set of letters that label transitions out of the state q.

A deterministic finite state transducer T is a DFA together with an output alphabet Ξ and a function $\gamma : Q \times \Sigma \to \Xi^*$ where the domain of γ is the same as the domain of τ. We extend γ to Σ^*, and then this transducer defines a partial function $\phi_T : \Sigma^* \mapsto \Xi^*$ where $\phi(u) = v$ if and only if $\tau(q_0, u) = q_f$ and $\gamma(q_0, u) = v$.

The inference problem is to infer a transducer T, given a set of pairs of strings from $\Sigma^* \times \Xi^*$ which we will write $X = (u_1, v_1), \ldots (u_n, v_n)$, Then given some new input data $Y = x_1, \ldots x_{n'}$ we predict values of $\phi_T(x'_i)$ which are then compared to an oracle. No feedback is given except when there is an exact match.

3 Inference of Finite State Models

There are basically two methods of learning finite state automata. The first, which learns DFAs uses a state merging algorithm starting from a prefix tree acceptor [3,6]. This can be proven to be correct under various assumptions. The second, which can learn some non-deterministic finite state automata, uses a randomly initialised Hidden Markov Model, together with an iterative statistical optimisation algorithm to find a local optimum – normally the Expectation Maximisation (EM) algorithm [7,2] is used because of its simplicity, elegance, and rapid convergence, but other methods of non-convex optimisation can be used, or gradient ascent. These optimisation problems are in general hard to solve exactly ([1]), and these algorithms will only find a local optimum rather than the global one.

Analogously with these two methods, there are two standard algorithms for learning finite state transducers. The first, OSTIA, [12] essentially uses a state merging algorithm on the transducer, together with an algorithm for shifting the outputs to an appropriate spot. While it is provably correct in the identification in the limit framework, it is not necessarily going to produce satisfactory results on smaller data sets. The second method, analogous to the use of HMMs, uses stochastic finite state transducers, also sometimes known as Pair Hidden Markov Models [8], with again a training algorithm based on the EM algorithm. This has been demonstrated to be effective for learning some simple transductions [4].

4 Problem Analysis

Problem 1 appeared to have been generated randomly. Accordingly we would expect the out-degrees of the states to be distributed binomially. As a first step we examined the histogram of signatures of the prefix tree acceptor which is shown in Table 1.

There are in principle three sources of information: the input language (the domain of the transduction), the output language, and the relationship between

Table 1. Signature histograms. This shows the numbers of different signatures of various sizes, of the various automata produced. For each automaton, we give the total number of distinct signatures of a given size n, and the total number of states with signatures of that size. We give the results for three automata: the initial prefix tree acceptor, the automaton formed after merging, and the final correct one. Note for the PTA, that there is only one empty signature, and we have exactly 95000 states with this – one for each training string. There are 1001 different signatures of size 1: one for each letter in the input alphabet.

$\lvert \mathrm{sig}(q) \rvert$	PTA		Merged		Final	
	n	N	n	N	n	N
0	1	95000	1	6520	1	1
1	1001	860039	841	2090	5	5
2	15920	41278	894	942	23	23
3	7580	12756	2059	2362	1923	1923
4	4764	6259	3906	3907	3904	3904
5	1955	2250	2089	2089	2089	2090
6	77	84	81	81	81	81
7	2	2	4	4	4	4
8	0	0	0	0	0	0
total	31300	1017668	9875	17995	8030	8031

the two of them. The large input alphabet means that the occurrence of a transition gives a huge amount of information, and allows us to identify pairs of states in the prefix tree that were generated by the same state in the transducer. Thus given that there is a very easily detectable structure in the input language, and that the inference of deterministic finite state languages is now a mature field, it seems appropriate to start by trying to find a compact DFA that generates the input language. The input and output languages are both regular but very different in structure. For example, the input language has 4 symbols that can appear as the first symbol in a word, whereas the output language has 922 (out of a possible 1000!). The reason for this enormous difference is that the output language is not "deterministic" – if we consider the generative process, there will be numerous transitions that only generate the empty string – thus the transitions between the states are not detectable from the symbols. Accordingly, learning the output language requires very different techniques from learning the input language, and after a few preliminary experiments, we decided to focus on learning the input language. However, given the large size of the problems, and the limited amount of data, we could not merge states only when we were very sure that they were equivalent, and thus it was inevitable that some errors would occur in the learning of the automaton. When learning an automaton from positive data only, it is very difficult to detect over-generalisation. When learning a transducer, the situation is very much easier: the output strings give you a great deal of information as to which transitions are being taken. Obviously in this case, we have some problems because so many of the transitions output the empty string: but enough of the states generate proper output strings, that we can detect problems.

Existing algorithms are not suitable for this approach: non-deterministic learning algorithms require a global optimisation that is quadratic in the number of states, which is not feasible on this scale of problem, and the OSTIA algorithm is sensitive to the order in which states are merged, and in this case we need to merge where the data permits, rather than in a fixed order.

5 Algorithm

We now describe the algorithm we applied to this problem.

- We shuffled the data set randomly and split the data into a training set of 95,000 pairs, and a validation set of 5,000 pairs, which we used to detect bugs and errors.
- We then removed the output data.
- On this input data we then constructed the prefix tree acceptor, and then ran a very efficient, aggressive state merging algorithm based on the signatures of the states, until we had an automaton, shown as Merged in Table 1.
- We then used an EM based alignment algorithm, with initialisation based on co-occurrence statistics. This is described in Section 7 below. This algorithm was also used to split states that had erroneously been merged earlier. This produces a small automaton that correctly models the training data.
- Using the additional information given by the output labels, we continued merging, but using a similarity measure that was sensitive to possible misalignments of the outputs.
- The final phase used the input strings from the test data to drive the state merging algorithm, until all of the test data was accepted by the transducer.

We will now describe the various phases of the algorithm in more detail.

6 Initial Merging

The initial merging algorithm followed standard techniques for DFA inference. We computed the prefix tree acceptor, and then merged states that were similar according to a variety of similarity measures, and then recursively determinised the results. Given the large size of the automata, with over 10^6 states, the similarity measures were selected based on the ease with which efficient indices could be constructed.

After looking at the PTA signature histogram in Table 1 we decided to start by merging states that shared 3 elements of their signature. To do this we created an index that stored for every 3-tuple of input symbols the set of states whose signatures are a superset of that tuple.

$$T(\sigma_1, \sigma_2, \sigma_3) = \{q \in Q : \{\sigma_1, \sigma_2, \sigma_3\} \subseteq sig(q)\} \tag{1}$$

Then for every tuple above a threshold, we merge all of those states. We use two heuristics to control rampant overgeneralisation. First we assumed an upper

bound on the cardinality of the signatures – if we have a state with more than 8 outgoing states then we assume this is an error, and secondly we have to maintain the prefix-free property. If that is violated then we *know* we have an error. In that case, we undo the merge, and continue. This reduces the automaton size to about $500,000$ states. We then switch to looking at merging states whose signatures share only two symbols using classes of the form:

$$P(\sigma_1, \sigma_2) = \{q \in Q : \{\sigma_1, \sigma_2\} \subseteq sig(q)\} \tag{2}$$

together with a measure of similarity based on the recursive computation of similarity.

Finally we resorted to merging states with signature of width 1. We looked for states that had not been merged; i.e. states that generate only a single string. For a state q such that $|\{w|\tau(q,w) = q_f\}| = 1$, and such that if $\tau(q', a) = q$ then $|\operatorname{sig}(q')| > 1$, we iterate along the unique path of states from q, and stop when we can find another state q'' such that $\tau(q'', w) = q_f$, and such that $|w| > k$ for some fixed threshold k. If there is such a state, then we merge q and q'. This helps only marginally with the coverage of the automaton, but it is very important with the alignment algorithm, since it forces the output strings away from the ends of the strings.

6.1 Similarity Computation

We compute a recursive similarity measure between the nodes, based on the number of symbols that they overlap with.

$$\begin{aligned} s_1(q, q') &= 1 \text{ if } q = q' \\ &= -\infty \text{ if only one of } q, q' \text{ is } q_f \\ &= |sig(q) \cap sig(q')| \text{ otherwise} \end{aligned} \tag{3}$$

$$s_n(q, q') = \sum_{a \in sig(q) \cap sig(q')} (1 + s_{n-1}(\tau(q, a), \tau(q', a))) \tag{4}$$

When we merge states such that $s_1(q, q') = 2$ we also require that $s_2(q, q')$ is larger than some threshold: we manually tweaked these thresholds at each iteration to get good performance. The definition in line 2 of Equation 3 has the effect of enforcing the prefix-free property: if we merged an accepting state with a non-accepting one, this would violate this property.

7 EM Based Alignment Algorithm

We now consider the alignment algorithm. We are given a DFA, and a set of training pairs, and we wish to attach to each transition of the DFA an element of Ξ^* such that the resulting transducer will model the training transduction correctly. Note first of all that this involves solving a set of word equations. Let $v_{q,a}$ be the output attached to the transition from q labelled with the letter a.

If we translate the input data into the sequence of transitions generated, e.g $u = u_1 u_2 \dots u_l$, this will produce the output v. This is an equation $v_{\delta(q_0,\lambda),u_1} + v_{\delta(q_0,u_1),u_2} + \dots v_{\delta(q_0,u_1 \dots u_{l-1}),u_l} = v$. We will have one of these equations for each training pair (so 100,000 equations) and one variable for each transition. We only align when the number of transitions is much less (say 20,000) than the number of equations. If the structure of the automaton is correct, we can simply solve these equations, by mapping them into linear equation(s), for example, using a Parikh map [13], or simply having variables representing the length of each string and using standard techniques for solving very large sparse linear systems. However in most cases, we will have made an error somewhere, and thus there will not be an exact solution. On the other hand, large parts of the automaton may be correct and thus will have solutions. We solve this by considering a relaxation of the problem. We associate with each transition not a unique output, but a probability distribution over a set of strings. We then adjust the parameters of the distributions to maximise the probability of the observed outputs. If the automaton is correct, then we hope to converge on a solution which will give every output string a probability of one, given the input string – i.e. all of the distributions will eventually converge to putting all of the probability mass on a single string. If the automaton has been merged incorrectly, then there will be transitions which cannot be assigned a single string consistently, and at convergence the distribution for that transition will generate more than one string (the support of the distribution will have cardinality greater than one). We then use this as a diagnostic test for splitting states of the automata.

The output distribution is a multinomial distribution over a set of strings. For every transition $t = (q, \sigma, q')$, we define a finite set of strings $O_t \subset \Xi^*$, where $O_t = \{o_1, \dots o_n\}$. We then define a multinomial distribution over this set using a set of parameters $\alpha_1, \dots \alpha_n$ where each α_i is the probability that this transition will generate the corresponding string o_i, and $\sum_i \alpha_i = 1$.

Given a maximum length L for the output strings, for a string $v \in \Xi^*$ define the set of all substrings of v of length at most L by $Sub_L(v) = \{w : |w| \leq L \wedge w \sqsubseteq v\}$.

For a transition $t = (q, a)$ we can define the set of all training pairs such that this transition is taken.

$$C_{q,a} = \{(u, v) \in X | \exists l, r \in \Sigma^*, u = lar \wedge \tau(q_0, l) = q\} \tag{5}$$

$$O_{q,a} = \bigcap_{(u,v) \in C_{q,a}} Sub_L(v) \tag{6}$$

Intuitively this is the set of all substrings that occur in every output string associated with this transition. Note that $\lambda \in O_{q,a}$, so the cardinality is always at least 1. Thus we know that, if the structure of the automaton is correct, the true output associated with that transition will be in the set $O_{q,a}$. Note the relationship to the insights of [15].

Given this set, we initialise the probabilities with the following computation where $\alpha(q, a, w)$ is the parameter for the transition (q, a) labelled with $w \in \Xi^*$,

where Z is a normalisation constant.

$$\alpha(q,a,w) = \frac{1}{Z}\left(\frac{|\{(u,v) \in X | w \sqsubseteq v\}|}{|C_{q,a}|} + \epsilon\right) \tag{7}$$

We add a small random fluctuation ϵ to break symmetry.

Computational note. Calculating this efficiently requires computing a suffix array for the set of strings $v_1, \ldots v_n$, and then for each transition (q,a) computing a bit vector representation of the set of $C_{q,a}$; given these two data structures we can then compute all of the α parameters rapidly.

We can illustrate this with a simple example: the string `zs bq` occurs 23 times in the output strings of the training data. In each of these 23 cases the symbol `hgc` occurs in the input strings. Thus *a priori* it is quite likely that the transition generating `hgc` might be generating the string `zs bq`. Accordingly we would give this one a high probability.

7.1 Iterative Optimisation

For each transition we have a multinomial distribution over a finite subset of Ξ^*. We will write these as $P(v|(q,a))$ Given an input string u, the probability of the transducer generating v is given by the sum over all possible alignments between the input and output strings. We write $l = |u|$ and $m = |v|$, and take the alignment variable at the beginning and end to be fixed, $i_0 = 0$ and $i_l = m$.

$$P(v|u) = \sum_{i_0=0}^{0} \sum_{i_1=i_0}^{m} \sum_{i_2=i_1}^{m} \cdots \sum_{i_l=m}^{m} \prod_{j} P(v[i_{j-1} : i_j]|(\tau(q_0, u[0:j]), u_j)) \tag{8}$$

The calculation of this sum with exponentially many terms can be performed efficiently using modifications of standard dynamic programming techniques, which we outline below.

We now want to maximise the following expression for the log likelihood.

$$\mathcal{L} = \sum_{i=1}^{n} \log P(v_i|u_i) \tag{9}$$

We do this using the EM algorithm. We compute the expected number of times that each string is generated by the transition and then normalise. This requires a two-dimensional DP trellis as is used for IOHMMs or PHMMs [10]. Since the recursions used are non-standard, because the state is fixed we give them here. We compute the forward and backward probabilities for a pair of strings $u_1 \ldots u_m \rightarrow v_1 \ldots v_n$ as follows.

$$\begin{aligned} f(0,0) &= 1 \\ f(0,j) &= 0 \text{ if } j > 0 \\ f(i,j) &= \textstyle\sum_{k=0}^{j} f(i-1,k)\alpha(\tau(q_0, u[0:i-1]), u_i, v[k:j]) \text{ if } i > 0 \end{aligned}$$

$$
\begin{aligned}
b(l,m) &= 1 \\
b(l,j) &= 0 \text{ if } j < m \\
b(i,j) &= \textstyle\sum_{k=j}^{l} b(i+1,k)\alpha(\tau(q_0,u[0:i]),u_i,v[j:k]) \text{ if } i < l
\end{aligned}
$$

Given these forward and backward probabilities we can compute the expectation of a particular output being produced, given a particular pair at a particular point where $\tau(q_0, u[0:i]) = q, a = u[i:i+1]$ and where $w = v[j:k]$:

$$P((q,a) \to w | u \to v) = \frac{f(i,j)\alpha(q,a,w)b(i+1,k)}{P(v|u)} \tag{10}$$

This is the E-step; the M step just sets the values of α and normalises. This is computationally quite intensive, requiring over an hour of computation and 2 Gigabytes of memory. We run this algorithm to convergence. Ideally this would converge to a log likelihood of 0, which would indicated that every transition was deterministically assigned a particular output.

However, at convergence the log likelihood is -700 (for 95,000 strings). Thus though the vast majority of transitions were unambiguously assigned an output string, there were 11 transitions that did not converge to unambiguous distributions. This was because there were two strings (or sets of strings) that needed to have different outputs. An advantage of this algorithm is that we can identify the exact location of the errors, and thus it is possible to split those states, so that we can have a unique output for every transition.

7.2 Splitting Erroneous States

The alignment algorithm above will not always produce a solution – for example if we merge the whole automaton much too aggressively then all of the transitions might only be able to output λ. In that case this approach would not work, and one would have to relax the requirement that all of the output strings lie in the intersection defined above.

We can now identify problematic states in the transducer. Given a transition (q,a) that we cannot assign an output string to deterministically, we take two pairs of training strings (u_1,v_1) and (u_2,v_2) such that the Viterbi output for that transition (i.e. the one generated by the single most likely alignment) is different. Thus we will have two pairs in the training set $(m_1ar_1, s_1w_1t_1)$ and $(m_2ar_2, s_2w_2t_2)$ such that $\tau(q_0,m_1) = \tau(q_0,m_2) = q$ and such that we want the transition to generate w_1 in the first case and w_2 in the second case. To do this we need to split states so that $\tau(q_0,m_1) \neq \tau(q_0,m_2)$.

We find the smallest i such that $m_1 = j_1k_1$, $m_2 = j_2k_2$, $|k_1| = |k_2| = i$, and $\tau(q_0,j_1) \neq \tau(q_0,j_2)$. We then split all of the i states $\tau(q_0, m_1[0:|j_1|+1])$ to $\tau(q_0, m_1[0:|j_1|+i]) = \tau(q_0,m_1)$.

To split a state q, we take the set of all transitions that end in q, $I_q = \{(q',a)|\tau(q',a) = q\}$, and for each element of I_q we create a separate state $q_{(q,a)}$. We then change every transition in I_q so that $\tau(q',a) = q_{(q,a)}$. The outgoing transitions of all of the new states are identical to that of q. The end result of these manipulations is that the two transitions can now be assigned the correct outputs.

At this point we have a smallish automaton that correctly transduces the training data, partially transduces the validation set, and has low coverage on the test data. Our goal now is to merge, while maintaining the transduction on the training set, until we cover the test set.

8 Merge/Split Algorithm

A transducer is onward if for every state, $(|\{(q', a)|\tau(q', a) = q\}| = 1)$ the outgoing transitions do not have a nonzero common output prefix. We define the output prefix of a state q to be the longest common prefix of all outputs generated starting from that state, i.e. the longest common prefix of the set of strings $\{v \in \Xi^* | \exists u \in \Sigma^*, \text{ s.t. } \tau(q, u) = q_f \wedge \gamma(q, u) = v\}$. It is simple to make a transducer onward, by identifying states which only have one incoming transition, and moving all non trivial output prefixes onto them. In our case we only considered states with in-degree one. More formally given a state q with only one incoming transition (q', a) such that $\tau(q', a) = q$, and such that there is a non empty string $u \in \Xi^+$ for all letters $b \in \Sigma$, such that $\tau(q, b)$ is defined, $\exists v \in \Xi^*$ such that $\gamma(q, b) = uv$.

We then use a state merging algorithm that uses the additional information from the output symbols.

When we merge incompatible arcs we push the residue onto arcs, just as is done in the OSTIA algorithm. This fails if we hit an incoming arc, or if we hit the edge. The latter is fatal, but with the other we can again split nodes. This happened 5 times. When we merge two states q, q' which both have a transition a, such that $\gamma(q, a) = uv$ and $\gamma(q', a) = uv'$ where v and v' start with different letters, and if the two states $\tau(q, a), \tau(q', a)$ have no other incoming arcs, we can then push the residues v and v' onto the outgoing arcs of the states $\tau(q, a), \tau(q', a)$, which will then give the two transitions the same output strings $\gamma(q, a) = \gamma(q', a) = u$, so that they can be merged. If however, there are other transitions into the same state say $\tau(p, b) = \tau(q, a)$, we can't do this. In this case we can split this node $\tau(p, b)$ into different ones, so that there is only one incoming transition into each node, and they can be pushed succesfully.

9 Final Phase

At this point the transducer was still not accepting all of the test input strings. Accordingly we used the test input data itself to drive the merging algorithm. For each string x_i in the test data that is not accepted by the transducer, we identified the longest prefix of x_i that was accepted. Take the longest l_i such that $x_i = l_i a r_i$, where $l_i, r_i \in \Sigma^*, a \in \Sigma$, such that $\tau(q_0, l_i) = q$. There is by construction no transition (q, a). We thus define $\text{sig}^+(q) = \text{sig}(q) \cup \{a\}$, using our knowledge of the test data to augment the information available. We then search for a state in the automaton with a signature that includes this one. Since $|\text{sig}^+(q)| \geq 2$, in each case we found a state that included this. In a couple of cases, there was more than one such state, in which case we selected the one with largest count. At

the end of this process, we had constructed an automaton that correctly modelled the training data and accepted the test strings. The final signature histogram is shown in Table 1, which shows the expected binomial distribution.

10 Implementation

The programs written for this project were all implemented in Java, based on an existing code base. We experimented with a number of other techniques beyond those described here. We started by writing an exploratory data analysis tool, that allowed us to examine parts of the transducers being generated, starting with the onwward sequential transducer constructed directly from the training data. We extended this to a simple command line interpreter, that allowed validation, storing of intermediate results, and other related computations. Since the data sets were large, the time for loading and storing the data sets and the large automata/transducers generated in the early phases of the algorithm was very substantial. Important productivity gains were achieved by keeping the data sets resident in memory.

The efficient implementation of the algorithms described in this paper required the use of a number of specialised algorithms. We focussed on applying algorithms with good theoretical efficiency rather than on constant factor optimisations.The transitions between states were represented, given the large alphabet size, using a sorted linked lists. Merging between states was performed using a union-find algorithm. Since we would sometimes make mistakes in the merging, by trying to merge incompatible nodes, we made this non-destructive, using a graph unification based algorithm originally developed for unification based parsing [9], that uses generational dereferencing. This allowed "undo" operations, when merging operations had undesirable consequences, without the prohibitive expense of copying entire transducers. Since all of the output strings are substrings of the output strings of the training data, we represented these lazily, without copying. This meant that taking substrings was a constant time/space operation, which gives an important efficiency gain – as used in algorithms for linear time construction of suffix trees [11]. These were implemented as immutable value classes. We also used suffix arrays extensively – we constructed these naively, rather than from suffix trees, but given the comparatively short length of the strings, this was efficient enough.

11 Discussion

First, in many respects learning a transducer is easier than learning an automaton; the output strings allow one to identify overgeneralisation during the state merging process. Thus we can say that large random instances of deterministic sequential transducers can be efficiently learned. Again the presence of a large alphabet turns out to be a great help for solving large grammatical inference problems [5].

Competitions of this sort are undoubtedly a good way of pushing forward the state of the art. The real question then is as to what direction the state of the

art should be pushed in. The two elements of this competition were first of all the requirement for exact identification, which is perhaps misconceived, since it rules out the sort of approximation techniques that will be useful in dealing with real world problems. The large size of the problems is very useful – first of all it rules out manual or semi-manual approaches. Indeed when developing the algorithms here, the automata are so large that actually inspecting them is out of the question or even misleading. One is forced to look at the automata through statistical summaries of particular properties, such as the signature histograms shown above. The large size of the problems means that the context free inference problems are extremely computationally intensive. A lot of the algorithms that might work, such as inversion transduction grammars [16,17] have prohibitively high, albeit polynomial, complexity.

The exercise was ultimately succesful. A number of lessons can be drawn from this exercise. First, large random instances can be solved accurately from small amounts of data. The number of states in the final automaton was over 5,000, and the number of string pairs in the training data was only 100,000. So the amount of data is less than quadratic. An important issue is that inevitably errors will occur. Mechanisms should be put in place for detecting and correcting errors.

Finally, very large scale state merging algorithms are practical using efficient data structures and indices. In our case, the use of a customised index, which was carefully tuned, allowed a state merging algorithm to run on a PTA with over a million states, very rapidly on a standard workstation. However in real world applications, random instances are less interesting. Naturally occurring instances tend not to have such clean properties (but then we do not have to get the problem exactly right). In particular, finite-state approximations to machine translation tasks will tend to have multiple states that are very similar, and thus will require a more refined approach.

There are a number of ways in which this work could be improved. Firstly, it would clearly be preferrable to combine the alignment and state merging phases; one way of doing this would be to annotate each transition with some vector of posterior probabilities, and incorporate this into the similarity computation.

Acknowledgements

I would like to thank the organisers of the Tenjinno competition.

References

1. N. Abe and M. K. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9:205–260, 1992.
2. L. E. Baum and T. Petrie. Statistical inference for probabilistic functions of finite state markov chains. *Annals of Mathematical Statistics*, 37:1559–1663, 1966.
3. R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In R. C. Carrasco and J. Oncina, editors, *Grammatical Inference and Applications, ICGI-94*, number 862 in LNAI, pages 139–152, Berlin, Heidelberg, 1994. Springer Verlag.

4. Alexander Clark. Memory-based learning of morphology with stochastic transducers. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 513–520, 2002.
5. Alexander Clark. Learning deterministic context free grammars in the Omphalos competition. *Machine Learning*, 2006. to appear.
6. Alexander Clark and Franck Thollard. Partially distribution-free learning of regular languages from positive samples. In *Proceedings of COLING*, Geneva, Switzerland, 2004.
7. A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society Series B*, 39:1–38, 1977.
8. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of proteins and nucleic acids*. Cambridge University Press, 1998.
9. Martin C. Emele. Unification with lazy non-redundant copying. In *Meeting of the Association for Computational Linguistics*, pages 323–330, 1991.
10. F.Casacuberta. Probabilistic estimation of stochastic regular syntax-directed translation schemes. In R.Moreno, editor, *VI Spanish Symposium on Pattern Recognition and Image Analysis*, pages 201–297. AERFAI, 1995.
11. Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
12. J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:448–458, 1993.
13. R. J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.
14. Brad Starkie, Menno van Zaanen, and Dominique Estival. Tenjinno machine translation competition. `http://www.ics.mq.edu.au/~tenjinno/`, 2006.
15. J. M. Vilar. Improve the learning of subsequential transducers by using alignments and dictionaries. In *Proceedings of ICGI*, pages 298–311, 2000.
16. Dekai Wu. Stochastic inversion transduction grammars, with application to segmentation, bracketing, and alignment of parallel corpora. In *IJCAI-95*, pages 1328–1335, Montreal, August 1995.
17. Hao Zhang and Daniel Gildea. Stochastic lexicalized inversion transduction grammar for alignment. In *Proceedings of the 43rd Annual Conference of the Association for Computational Linguistics (ACL-05)*, 2005.