Final Report: Natural Language Generation in the Context of Machine Translation

Jan Hajič[1]
Martin Čmejrek[2]
Bonnie Dorr[3]
Yuan Ding[4]
Jason Eisner[5]
Dan Gildea[6]
Terry Koo[7]
Kristen Parton[8]
Gerald Penn[9]
Dragomir Radev[10]
Owen Rambow[11]
with contributions by[12]
Jan Cuřín
Vladislav Kuboň
Ivona Kučerová

October 28, 2005

[1]Team leader, Charles University, Prague, Czech Republic
[2]Charles University, Prague, Czech Republic
[3]Affiliate team member, University of Maruland, USA
[4]University of Pennsylvannia, USA
[5]Johns Hopkins University, USA
[6]Affiliate team member, University of Pennsylvannia, USA
[7]MIT, USA
[8]Stanford Univerity, USA
[9]University of Toronto, Canada
[10]University of Michigan, USA
[11]University of Pennsylvania, USA
[12]All from Charles University, Prague, Czech Republic

Abstract

Abstract

Text......

Chapter 1

Introduction

Jan Hajič

1.1 Section Name

Text......

Chapter 2

Summary of Resources

Jan Hajič, Martin Čmejrek, Jan Cuřín, Vladislav Kuboň

2.1 Section

Text......

Chapter 3

The Generation System MAGENTA

Jason Eisner, Jan Hajič, Dan Gildea,
Yuan Ding, Terry Koo, Kristen Parton

3.1 System Architecture

by Jan Hajič and Jason Eisner

3.1.1 Section Name

Text......

3.2 Synchronous Tree Transducers

by Jason Eisner

3.2.1 Section

Text......

3.3 The Elementary Tree Model

by Jan Hajič

3.3.1 Section

Text......

3.4 The Proposer

by Yuan Ding

3.4.1 Section

Text......

3.5 Using Global Tree Information: Preposition Insertion

by Terry Koo

Prepositions are elided in the TR but exist in the AR. This section describes our implementation of a classifier that indicates where prepositions must be inserted into the TR. Because the system operates on the tree as a whole, it has access to global tree information and can make more accurate decisions than the Proposer. Although we only discuss preposition insertion in this section, the techniques we present are applicable to other tree transformations such as insertion of determiners or auxiliary verbs.

The remainder of this section is split into four subsections. The first two subsections present, respectively, an overview of the C5.0 data mining tool, which we used to train our classifier, and the general technique of using classifiers to capture global tree information. The next subsection presents our preposition insertion implementation, describing the operation of our classifier, how we prepared its testing and training data, how we improved its performance, and how it performs. The final subsection concludes with suggestions for future work.

3.5.1 Description of C5.0

C5.0 is a data mining tool that generates classifiers in the form of decision trees or rule sets. We present a brief overview of the C5.0 data mining tool. The reader already familiar with C5.0 may skip this sub-section.

The training data for C5.0 is set of *cases*, where each case consists of a *context* and a *classification*. The context can be made of any number of variables which can be real numbers, dates, enumerated discrete values, or functions on the other variables (such as `area := width*height` or `young := age < 20`). The classification is the category to which C5.0 should assign this case. C5.0 has a number of training options, a few of which we describe below.

Discrete Value Sub-setting C5.0 can make decisions based on subsets of discrete values instead of individual discrete values. On a data set with many discrete values, discrete value sub-setting reduces fragmentation of the training data.

Adaptive Boosting C5.0 can run several trials of classifier generation on given sets of training and test data. The first classifier generated may do well but will probably make errors on some regions of the test data. C5.0 takes these errors into account and trains the second classifier with an emphasis on these error-prone data. This process continues iteratively until some specified number of trials is complete or the accuracy of the classifier becomes either very high or very low.

Misclassification Costs C5.0 allows misclassification costs to be specified individually. This is useful in when one kind of error is more grave than another; for instance, when diagnosing a potentially fatal disease, we would rather misdiagnose a healthy patient as sick than vice versa. By specifying a high cost for misclassifying `sick` as `healthy` and a low cost for misclassifying `healthy` as `sick`, we can instill a bias in the classifier C5.0 produces.

When using a C5.0 classifier, the context is presented as input, and the classifier attempts to deduce the correct classification. Source code is available for a free C program called `sample`, which takes a trained C5.0 decision tree or rule set and classifies new cases with it.

3.5.2 Using Classifiers to Capture Tree Information

We now discuss the general technique of using classifiers to aid our tree transduction. Recall that our tree transduction operates by transforming small tree fragments at a time and then composing those fragments into whole trees. Unfortunately, the TR tree fragments only provide local information limited by the size of the fragment. This lack of information makes it difficult for the Proposer to choose the correct AR transformation.

In actuality, the entire TR tree is available to the transducer, but we choose not to transduce the entire tree at one go for data sparseness and computational efficiency reasons. However, some TR to AR transformations depend on long-range relationships or other data which are lost when the tree is broken into small fragments. We would like to capture the relevant long-range information in some local form that the Proposer can easily access.

Our approach is to preprocess the TR tree with a classifier, attaching arbitrary informative labels to individual nodes. When the Proposer receives a tree fragment, it reads the labels off of the nodes in the fragment, incorporating the labels' information in its decisions.

3.5.3 The Preposition Insertion Classifier

The classifier operates by labeling individual nodes of the TR tree. The possible labels are `nothing`, indicating that no preposition should be inserted above this node, or `insert_X`, where `X` represents a preposition to insert above the labeled node. `X` can either be a single preposition such as "of" or a compound preposition such as "because_of". In the case of a compound preposition such as “because_of”, “of” should be inserted as the parent to the labeled node and “because” should be inserted as the parent of “of”.

Unfortunately, we were unable to integrate the preposition insertion classifier into the tree transduction system, due to time constraints. However, the classifier itself is fully implemented and has been tested as an independent module.

The remainder of this subsection is broken into three parts which describe, respectively, how we created the training data for our classifier, how we attempted to improve its performance, and how the classifier performs.

Preparing the Training Data

We now discuss how we prepared training data for our preposition insertion classifier. To begin, we have only trained and tested the classifier on data from Input 1. This is because Input 1 has a large amount of data, covering all 24 WSJ sections, and was the earliest available Input of this size. Naturally, we did not train or test the classifier on the parts of the WSJ which were reserved for testing.

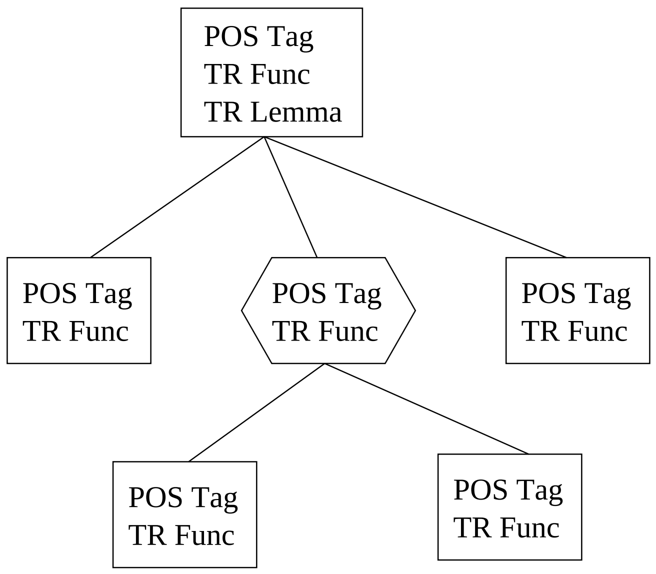

Figure 3.1: Inputs to the classifier when classifying current node (hexagon).

The classifier’s inputs, displayed in Figure **??**, are the attributes of the node to be classified and a few of the surrounding nodes. Specifically, we pass the POS tag and TR functor of the node being classified, its parent node, its left and right brother nodes, and its leftmost and rightmost children. In addition, the TR lemma of the parent node is also included. We would ideally include more TR lemmas but doing so increases the size of the problem drastically and causes C5.0 to exit with an out of memory error.

One caveat about our inputs is that the POS tags in the classifier's input are not full POS tags — at transduction time, only simple POS tags are available. This simple POS tag can distinguish between basic categories such as noun, verb, and adjective, but does not make distinctions within these categories, such as `VBZ` and `VBD`. We simulate this by using only the first character of the full POS tag in the input to the classifier.

To create the data, we used a Perl script that traverses a TR tree, creating a single case entry for each node. The contextual information is gathered in fairly straightforward fashion, by examining the parent, left and right brother, and leftmost and rightmost children of the current node. The classification is a bit more tricky to find, however.

Since the TR trees from Input 1 are automatically generated from English surface text, they contain the original prepositions, marked as hidden nodes. The hidden prepositions are also attached at a different position in the TR than the AR; if a preposition is attached above node X in the AR tree, it is attached as the leftmost child of X in the TR tree. As a second check, we examine the corresponding AR tree and make sure that the hidden prepositions in the TR tree appear in the AR tree as parents of the current node, with AR functor `AuxP` and POS tag `IN`. These two quick checks will accept some subordinating conjunctions as prepositions, but this is reasonable since the two phenomena are closely related. Finally, the prepositions that pass both checks are concatenated with `insert`, forming the current node's classification. If no prepositions were found, the classification `nothing` is used.

In the previous paragraph, we have left out two important mechanisms for reasons of clarity. These mechanisms are conjunction and apposition hopping and skipping. We first describe conjunction and apposition hopping.

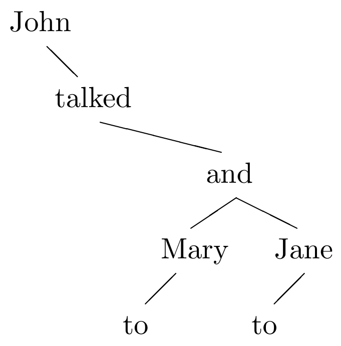

Figure 3.2: TR tree for "John talked to Mary and to Jane"

Suppose that we always looked at the literal parent node when gathering context information. In the TR of a sentence such as "John talked to Mary and to Jane" (see Figure ??), the parent of "Mary" and "Jane" is the conjunction "and". However, "and" provides no information about the relationship between "talked" and "Mary" or "Jane" represented by the preposition "to". We should have ignored "and" and looked at the parent of "and", "talked". Rather, "and" is simply a grouping node and the true parent, with respect to the preposition "to", is "talked". Through an identical mechanism, appositions also exhibit this spurious parenthood. To fix these errors, we look for the parent by hopping upward until we find a node that is neither a conjunction nor an apposition.

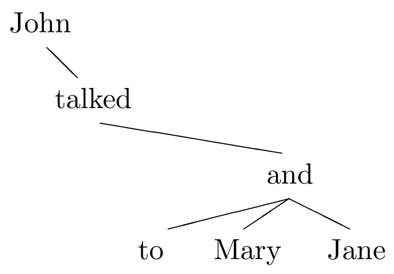

Figure 3.3: TR tree for "John talked to Mary and Jane"

Conjunction and apposition skipping complements hopping. In the sentence "John talked to Mary and Jane" (see Figure ??), "and" would be classified `insert_to` by our naive data creation script. However, the "and" is not the actual target of the preposition "to". Rather, the targets are "Mary" and "Jane". We solve this problem by simply skipping over conjunction and apposition nodes without creating any data for them; we let conjunction and apposition hopping generate the appropriate data when "Mary" and "Jane" are processed. We must, however, ensure that the correct prepositions are detected when "Mary" and "Jane" are processed. Thus, we alter the hopping process to check for prepositions attached to any of the nodes on the path to the true parent. In the TR of the sentence "John talked to Mary and Jane", "to" is attached to "and", which is between "talked" and both "Mary" and "Jane". This means that "Mary" and "Jane" are processed as if they had the preposition "to" inserted above them, even though they do not. Note that this makes the TR trees of Figures ?? and ?? identical with respect to the training data they produce. Since the two sentences are nearly identical in terms of meaning, and certainly interchangeable, we have decided that this is acceptable behavior.

Improving Performance

We now discuss how we improved the performance of our classifier. Our efforts have focused on increasing *insertion recall*, which we define as the number of correct preposition insertions the classifier makes divided by the total number of prepositions in the test set. We emphasize insertion recall because the classifier, when integrated, will make suggestions to the Proposer. Ideally, these suggestions always include the correct answer. We focus only on insertions because our system attains perfect recall on `nothing`, for trivial reasons which we will explain later.

Our first performance problems derive from the high prior probability of the classification `nothing`, which makes up roughly 9/10 of all classifications. On the other hand, there are roughly 330 classifications corresponding to preposition insertions, all sharing the remaining 1/10 of the probability space. This makes the classifier tentative about inserting prepositions; if there is any uncertainty about whether to insert a preposition, the classifier will generally opt for `nothing` since it is right 9 times out of 10. Unfortunately, this tentativeness causes poor insertion recall.

We first attempted to deal with this problem by using C5.0's differential misclassification costs feature to favor insertions and disfavor `nothing`. However, we made little improvement with this approach and actually performed worse in some cases. We also attempted to use the adaptive boosting, hoping that C5.0 could detect that it was classifying many prepositions insertions as `nothing` and compensate for that in successive trials. However, attempts to train using adaptive boosting also met with failure: C5.0 aborted the training every time because the scores of the classifiers became too low.

The solution we eventually arrived at was to train on data that consisted only of preposition insertion cases. Our intuition was that if the `nothing` cases were causing confusion, then we should remove them. The classifier we trained on the preposition-only data is able to get significantly higher insertion recall than the classifier we trained on all the data, even though both classifiers are trained on the same preposition insertion cases. The obvious disadvantage is that a classifier trained on preposition-only data will never make a classification of `nothing`. We solve this problem by modifying the classifier to return two classifications: `nothing` and the preposition insertion determined by the decision tree. This explains why, as we mentioned above, we have perfect recall on `nothing`.

Notice that we are now returning multiple suggested classifications. Although this increases the number of possibilities that the tree transduction must evaluate, the increased insertion recall more than offsets the computational disadvantage. The natural step at this point is to alter our decision tree so that it returns multiple classifications.

Our first and most primitive application of this idea is what we call *N Thresholding*. To explain, when the normal classifier evaluates an input, it creates confidence values for each of the possible classifications and returns the classification with the highest confidence as its decision. A classifier using N Thresholding is the same except it returns the best N_{thresh} classifications. This method is able to get high insertion recall, but only when N_{thresh} is large. Furthermore, when N_{thresh} is large, there are many cases where the decision tree's highest or second highest confidence classification is correct, but the remaining classifications are still returned as dead weight. Manual study of the confidence values of in these cases shows that most of the confidence mass is concentrated in these one or two highest-ranked classifications. Additionally, when the correct answer is in one of the lower-ranked classifications, the confidence mass is generally more spread out. These two observations lead us to our next thresholding technique.

The next kind of thresholding we developed is what we call *C Thresholding*, where C stands for "confidence". Instead of returning a fixed number of classifications, C Thresholding can return a variable number. C Thresholding operates by picking the smallest set of M classifications such that the sum of the confidence values of the M classifications is greater than fC_{total}. f is an adjustable parameter and C_{total} is the sum of confidence values for all classifications. Thus, if the confidence mass is concentrated in one or two classifications, the threshold will probably be passed using just those one or two classifications. If, on the other hand, the confidence mass is spread evenly, more classifications will be returned. This behavior follows the trends we have observed.

Note that although $C_{total} \leq 1$, it is not necessarily 1; hence the need to multiply f by C_{total} to create the threshold value. We set a hard limit on how many classifications can be returned, which is also an adjustable parameter. Thus far, however, we have used the hard limit 15 for all of our classifiers, and anticipate that changing the hard limit would not have any profound effects. For comparison to the N Thresholding classifier on a given set of cases, we calculate the "effective N_{thresh}" of a C Thresholding classifier as the average number of classifications suggested.

Unfortunately, when compared at equal values of N_{thresh}, the C Thresholding classifier does not perform much better than the N Thresholding classifier. This is strange, since manual study of insertion cases shows that C Thresholding consistently returns fewer classifications than N Thresholding.

The problem lies in the `nothing` cases. Since the classifier has been trained on preposition-only data, it has no knowledge of what to do with `nothing` cases. When it is presented with the input for a `nothing` case, it will produce a fairly even spread of very low confidences. As we have mentioned earlier, this causes the C Thresholding technique to return multiple classifications. After more manual study, we made the key observation that C_{total} is small for these `nothing` cases. This leads us to our next and final thresholding technique.

Our final thresholding technique is what we call *Aggressive C Thresholding*. This is identical to C Thresholding except the threshold value is $f(C_{total})^2$. Preposition insertion cases will have C_{total} reasonably close to 1, so the effect of squaring it is small and the number of classifications returned is

N Thresholding		C Thresholding		Aggressive C	
N_{thresh}	**Recall (%)**	**Eff.** N_{thresh}	**Recall (%)**	**Eff.** N_{thresh}	**Recall (%)**
2	70.1568	2.8719	74.2187	**2.0697**	**71.1497**
3	76.3963	**2.9981**	**74.6136**	2.0992	71.6462
		3.1038	75.2680	2.1158	71.7026
		3.2714	77.4343	2.1493	73.4514
		3.3517	78.2241	2.1666	73.9366
		3.5324	78.8334	2.1702	73.9479
		3.7341	79.5555	2.1799	74.0494
4	80.5935	**4.0257**	**80.1647**	2.2271	74.5797
		4.4184	82.9065	2.2812	76.5655
5	82.9967	4.8659	84.1137	2.3397	76.7799
		5.3083	84.7004	2.3583	76.8927
6	84.8810	**5.9295**	**86.4380**	2.4066	77.2199
7	86.2011	6.6144	87.7581	2.4884	77.9871
8	87.2955	7.7973	89.0105	2.6519	78.8559
9	87.9838	**9.0266**	**89.8793**	**2.9678**	**80.1647**
10	88.6720	**10.1724**	**90.2516**	3.2876	80.8643
11	89.1459	12.5654	90.6916	3.5069	81.3156

Figure 3.4: Number of Classifications Returned and Percent Recall for the three thresholding methods. Values in both C Thresholding columns are **bold**-ed when their effective N_{thresh} is close to an integer. For easier comparison, the N Thresholding values have been moved next to their closest companions in the C Thresholding column.

similar to C Thresholding. However, for `nothing` classifications, C_{total} will be small, and squaring it will force the threshold value very low. Therefore, in these `nothing` cases, Aggressive C Thresholding returns far fewer classifications, usually only 1, than normal C Thresholding. Aggressive C Thresholding thus combines the accuracy of C Thresholding on preposition insertion cases with high economy on `nothing` cases.

Our current best classifier is trained on preposition-only data and uses Aggressive C Thresholding to modulate its suggestions.

Results

We now present the results of testing our preposition insertion classifier. All of the results we include are of a classifier trained on preposition-only data, and using various thresholding techniques.

Figures **??** and **??** do not show any clear winner between N Thresholding and C Thresholding. Both, however, show that Aggressive C Thresholding has a clear advantage over the other two. The Aggressive C Thresholding classifier can classify with 80% insertion recall, using only about 3 classifi-

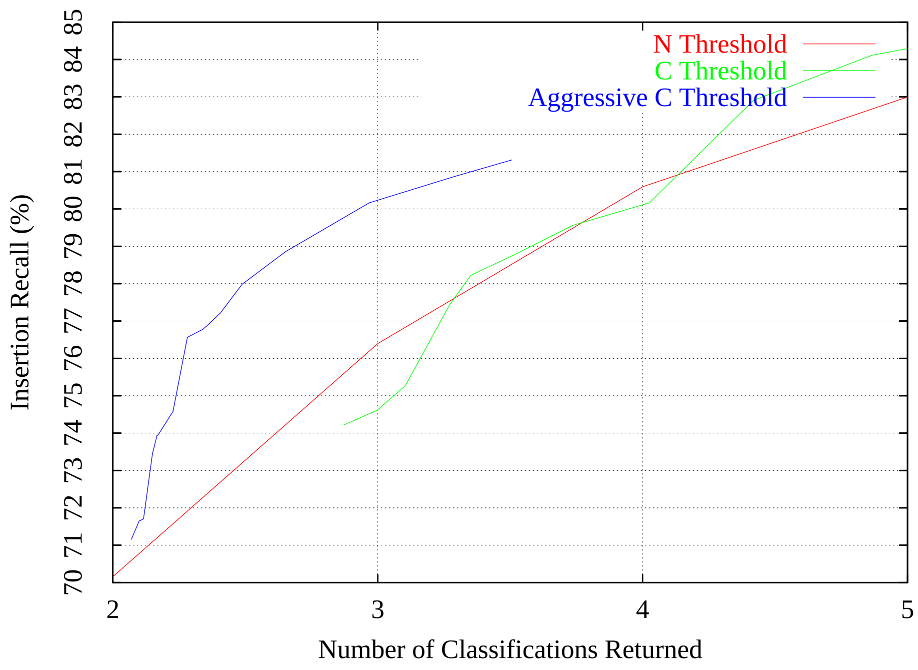

Figure 3.5: Percent Recall vs. Number of Classifications Returned for the three thresholding methods.

cations per case. Of course, this performance can be improved upon, but it is nevertheless an impressive start.

3.5.4 Future Work

There are a number of tasks we should undertake following this research. Clearly, the first task is integrating the above classifier with the Proposer. This will include training the classifier again on the other Inputs. Once integrated, we can evaluate the impact that the classifier has on overall English generation performance. From there, we will create more classifiers if the technique turns out to be useful, which we expect to be the case.

We should also improve the performance of the preposition insertion classifier. For instance, recall that we use only one TR lemma in the input because C5.0 cannot allocate enough memory. We would like to group TR lemmas, which have over 30,000 different values, into a smaller number of categories. Instead of directly using the TR lemmas as input, we can use the derived categories. Provided that the number of categories is small enough, we could use the categorized TR lemmas of multiple nodes.

Another improvement would focus on the creation of training data. Recall that we accept some subordinating conjunctions as prepositions. If we were able to separate prepositions from subordinating conjunctions, we could create a more uniform set of training data, which in turn might lead to a more accurate decision tree.

Finally, we could look at using several different classifier generators, with the hope that one might provide us significantly better performance.

To conclude, we have shown the effective of the general technique of using classifiers to capture tree information. We have also created a working classifier which achieves high recall on preposition insertions. We hope that in the future the preposition insertion classifier and other classifiers like it will be integrated into the tree transduction.

3.6 The Word Order Language Model

by Dan Gildea

3.6.1 Section

Text......

3.7 Punctuation and English Morphology

by Kristen Parton

3.7.1 Section

Text......

Chapter 4

The Generation System ARGENT

Dragomir Radev

4.1 Section name

Text......

Chapter 5

Towards a Full MT System: English and Czech Tectogrammatical Parsing

Gerald Penn, Owen Rambow, Bonnie Dorr and Ivona Kučerová

5.1 Czech Tectogrammatical Parsing by xxx

by Gerald Penn

5.1.1 Section

Text......

5.2 Using xxx for English Tectogrammatical Parsing

by Owen Rambow, Bonnie Dorr and Ivona Kučerová

5.2.1 Section

Text......

Chapter 6

Evaluation

Terry Koo and Jan Hajič

6.1 Overview

We evaluated our systems with IBM's BLEU evaluation metric. Because BLEU scores are a relative measure, we also created baseline and upper bound systems to use as reference points. Additionally, we created a competitor system using GIZA++, so that we could compare our performance to that of a good word-to-word system.

The remainder of this chapter describes, in order, the BLEU evaluation metric, the baseline system, the upper bound system, the GIZA++ system, the evaluation mechanics, and the results.

6.2 BLEU

We used IBM's BLEU metric because of its convenience and accuracy. BLEU is automatic, so it does not require expensive and time-consuming human evaluations. Additionally, BLEU has been shown to correlate with the judgments of humans [**?**].

We obtained a Perl implementation of the baseline BLEU metric, as described by [**?**]. The following paragraphs describe the operation of the baseline BLEU metric as we have used it.

BLEU has two inputs: the candidate translation and a set of reference translations. First, the candidate and reference translations are broken into sentences. Each sentence is then processed into sets of 1-grams, 2-grams, 3-grams and 4-grams. Each of the candidate sentence's n-grams are checked for inclusion in the union of the reference sentences' n-grams, and matching n-grams are tallied. To guard against overly repeated n-grams (i.e. "the the the the the"), a given n-gram is prevented from matching more times than the maximum number of times it appears in any of the reference sentences. For each level of n-gram, the number of matching n-grams in the entire candidate translation is divided by the total number of n-grams in the candidate translation. This yields four numbers which are the *modified n-gram precision* for 1-grams through 4-grams.

With the above scheme, shorter sentences can get higher modified n-gram precision scores than longer ones, since the number of times an n-gram can match is bounded. At the extreme, one can imagine a sentence composed of a single 4-gram that matches the reference; this would get perfect scores for all n-gram levels. To counteract the advantage of short sentences, BLEU uses a *brevity penalty*. The brevity penalty is a decaying exponential in c/r, where c is the length of the candidate translation in words, and r is the length of a best-matched reference translation. Specifically, r is the sum of the lengths of the reference sentences which are closest in length to their corresponding candidate sentence. The brevity penalty multiplied by the geometric mean of the four modified n-gram precision scores gives the final BLEU score.

We made one modification to the BLEU implementation, which was to add sentence ID's. Initially, we had trouble with dropped sentences because the original Perl code assumed the sentences in the candidate and reference translations were in the same order. A dropped sentence would change the candidate translation's order and cause an incorrect evaluation. Our modification allows BLEU to use sentence ID's to match sentences in the candidate and reference translations, rather than depending on a rigid ordering.

The number and variety of reference translations used affects the BLEU score and its accuracy. There may be many ways to correctly translate a given sentence, but the BLEU metric will only acknowledge the words and phrasing found in the reference translations. The more reference translations there are, and the more variety there is among them, the better the BLEU score can recognize valid translations.

In our own evaluations, we used 5 reference translations. Each reference contained roughly 500 matching sentences selected from WSJ sections 22, 23, and 24. Of the five references, one was the original Penn Treebank sentences and the other four were translations to English of a Czech translation of the Penn Treebank, done by four separate humans. We feel this gives us a good level of coverage in our reference translations.

6.3 Upper Bound System

The upper bound for the performance of our system would be translation by humans. Accordingly, our upper bound comparison system is composed of the references translations themselves.

To evaluate the references, we held out each reference in turn and evaluated it against the remaining four, averaging the five BLEU scores at the end. For the purposes of a meaningful comparison, all of the results we present were created using the same 5-way averaging.

6.4 Baseline System

Each of the four Inputs is associated with its own baseline. Each baseline, however, operates off the same principle: output the TR lemmas of each input TR tree in some order.

For Inputs 1 and 3, the TR lemmas are output in a randomized order. These Inputs are automatically generated from surface text and the TR trees capture the true surface word ordering. However, our generation system for these Inputs views the input TR trees as unordered. Thus the baseline is similarly deprived of word order information.

For Inputs 2 and 4, the TR lemmas are output in the order they appear in the TR tree. These inputs are, respectively, the manually annotated and Czech transfer TR trees. Their word orderings are meaningful; the manually annotated TR captures the deep word order while the Czech transfer TR captures the Czech word ordering. Our generation system can make use of this input word ordering. Therefore, the baseline should also be able to take advantage of this.

6.5 GIZA++ System

The GIZA++ system was created by Jan Cuřín and trained on 30 million words of bilingual text. This system is a representative of the word-to-word machine translation systems with which our own system will compete.

6.6 Evaluation Mechanics

Our 500 reference sentences were split into two test sets: a devtest set which we used to evaluate our system during its development, and an evaltest set which remained untouched until its use in the final evaluation at the conclusion of the workshop.

The evaluation itself is orchestrated by a number of `sh` and Perl scripts. The scripts allowed individual evaluations to be run as well as batch evaluations, and created a HTML webpage and GIF bar graph as output.

6.7 Results

	Base 1	**Gen 1**	**Base 2**	**Gen 2**	**Upper**
BLEU	0.04184	0.24416	0.16022	0.2365	0.53366
4-Gram	0.01146	0.10286	0.08038	0.09894	0.3385
3-Gram	0.02608	0.20238	0.17814	0.2016	0.46396
2-Gram	0.07564	0.41364	0.37008	0.41038	0.62766
1-Gram	0.76004	0.88762	0.7609	0.88116	0.8441
Brev	0.64902	0.82612	0.63484	0.8117	0.99396
	Base 3	**Gen 3**	**Base 4**	**Gen 4**	**GIZA++**
BLEU	0.04142	0.24324	0.05862	0.0479	0.18954
4-Gram	0.01146	0.10152	0.0181	0.00594	0.06884
3-Gram	0.026	0.20078	0.0408	0.02722	0.1365
2-Gram	0.07194	0.4132	0.12028	0.13192	0.28172
1-Gram	0.76194	0.8872	0.48054	0.6013	0.62328
Brev	0.65042	0.82758	0.7252	0.80064	0.9402

Figure 6.1: Final evaluation results for all systems.

Figures **??** and **??** display the final evaluation results in chart and graph form. In each of these, "Base N" is the baseline for Input N, and "Gen N" is the generation system for Input N. The BLEU scores as well as the four modified n-gram precision scores and the brevity penalties are displayed.

The system as it currently stands can outperform the baselines for Inputs 1, 2, and 3, but it is still below the baseline for Input 4, the full MT. It is also well below the GIZA++ system's performance This is unfortunate, but as there are still many components missing from our generation system, we feel it is acceptable.

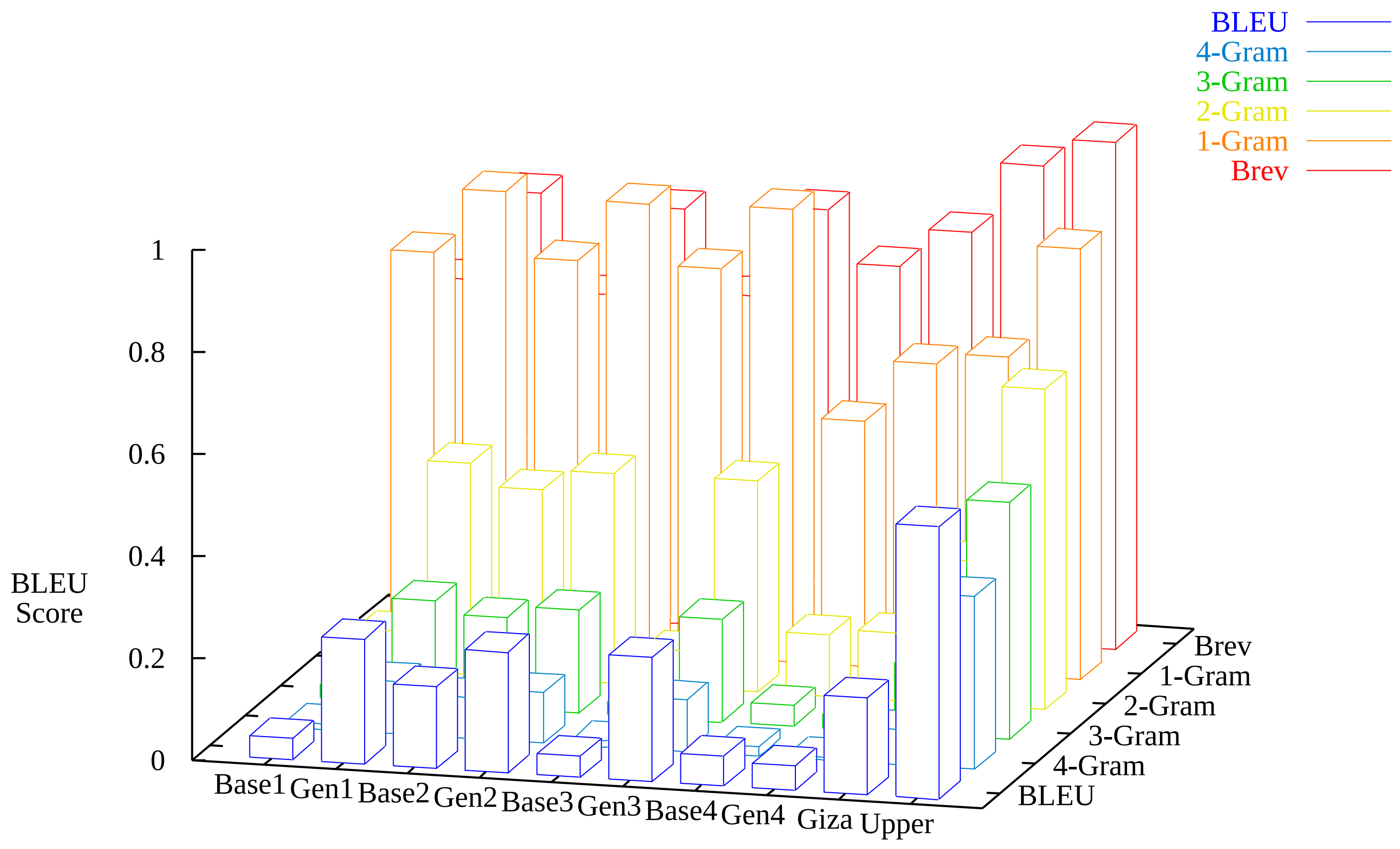

Figure 6.2: Graph of final evaluation results for all systems.

Bibliography

[Papineni 2001] Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu (2001). “IBM Research Report Bleu: a Method for Automatic Evaluation of Machine Translation,” IBM Research Division Technical Report, RC22176 (W0109-022), Yorktown Heights, New York.

[Och 2000] Franz Josef Och, Hermann Ney. “Improved Statistical Alignment Models”. *Proc. of the 38th Annual Meeting of the Association for Computational Linguistics*, pp. 440-447, Hong Kong, China, October 2000.

Chapter 7

Conclusions and Future Directions

Jan Hajič and Jason Eisner

7.1 Section Title

Text......