Rapid prototyping with Limited Resources in an existing Transfer Based Machine Translation system

Simon Zwarts,
Centre for Language Technology,
Division of Information and Communication Sciences,
Macquarie University, Sydney, Australia
szwarts@ics.mq.edu.au

and

Alon Lavie
Language Technologies Institute,
Carnegie Mellon University
Pittsburgh, PA, United States of America
alavie@cs.cmu.edu

Abstract

We describe the rapid development of a Dutch-English Machine Translations system in an existing transfer-based framework. Constructions of Transfer-based Machine Translation prototypes are known for requiring a lot of human labour. In this approach we focus on building a prototype really fast with minimal human labour. This divides the project in two tasks: deriving a suitable lexicon and building an acceptable parallel grammar. The construction of the Lexicon is done mainly by using results of statistical training on a parallel corpus. The grammar is built by hand, because the used Framework, the Avenue framework can work with partial parses the focus is on building a grammar which recognises as much constituents as possible, but not necessarily full parses of every sentence. Performance results show this system can keep up with previous built prototypes in the Avenue Transfer-Based project.

1 Introduction

Rapid prototyping in the transfer based machine translation systems has done before (Lavie et al., 2004) and (Lavie et al., 2003). But unlike previous projects we try to limit the time of prototyping even further to six weeks manpower. Furthermore we do this by the use of existing, publicly available resources. This means a considerably amount of time of the total project has to be spend on building a bilingual lexicon and finding the right grammatical functions for this lexical items. Since there are to our knowledge no publicly available bilingual Dutch-English word lists, which include part of speech, this has to be created during the project.

The prototype is build in the Avenue Project (Probst et al., 2003).

First we describe this transfer engine shortly, then we address some issues with the Dutch Language and the Dutch English language pair. After that we describe the actual project which involves building the Lexicons and the Grammar within certain restrictions. Finally evaluation metrics are addressed to see how this prototypes works out and how it holds up against similar projects.

2 Avenue Transfer Engine

The Avenue project was designed with the idea to handle languages with limited resources. When we want to handle languages with fewer resources, we need to have a system which is robust enough to handle spare and partial information. The first thing to give us this ro-

```plaintext
(0 0 "that" 1 "dat" "@prep")
(1 1 "is" 1 "is" "@verb")
(1 1 "has" 1 "is" "@auxv")
(0 1 "that is" 1 "dat is" "@s")
(2 2 "a" 1 "een" "@det")
(2 2 "an" 1 "een" "@det")
(3 3 "asset" 1 "pluspunt" "@noun")
(2 3 "a asset" 1 "een pluspunt" "@np")
(2 3 "an asset" 1 "een pluspunt" "@np")
(1 3 "is a asset" 1 "is een pluspunt" "@vp")
(1 3 "is an asset" 1 "is een pluspunt" "@vp")
(0 3 "that is a asset" 1 "dat is een pluspunt" "@s")
(0 3 "that is an asset" 1 "dat is een pluspunt" "@s")
```

Figure 1: Lattices created by the transfer engine
bustness is the transfer engine. This operates on a parallel grammar. In this grammar we immediately specify for every rule in the grammar how the target side needs to be constructed. We want to keep this information together so we do not first have to do a parse on the sentence and then try to do the transfer afterwards on the parse result. Because we keep parsing and transfer at the same place we immediately get the transfer for all the different non terminals we parse, so that even when we do not have a full parse we have different chunks of the sentence in the place where the grammar could work. Figure 2 gives shows what the result is of this process and how the different lattices are stored. The parser uses a chart parser, each chart with its transfer is called a lattice. We will come back to this.

The notation for each of these rules is defined as:

\[ SNT : : TNT \rightarrow [NT1 \ NT2] \rightarrow [NT3 \ NT4] \]

This rule describes a Source non terminal \textit{SNT} which exists of a non terminal \textit{NT1} and \textit{NT2}. Then on the target language side a phrase with the non terminal \textit{TNT} can be created which has two other non terminals. In our case at the lowest level these non terminals will be part of speech. It is also necessary to know the alignment between the source and the target side, otherwise the target side cannot be constructed. For each non terminal at the target side we need to specify a proper alignment at the source language side. The non terminals on the source side are labeled with \( x_1 \ x_2 \ldots \) and \( y_1 \ y_2 \ldots \) on the target side. In our example when we want to align \( NT3 \) with \( NT1 \) and \( NT4 \) with \( NT2 \) we write:

\[ (x_1 :: y_1) \]

\[ (x_2 :: y_2) \]

The alignment were we reverse order would be when \( x_1 \) is aligned to \( y_2 \) and \( x_2 \) to \( y_1 \). This approach benefits us especially in rapid prototyping: over a time span of only a couple of weeks it is almost impossible to write a grammar with a wide coverage in the source language for complete sentences, but even with simple rules we can cover most of the sentence parts.

This formalism also allows both an automatic grammar derivation approach, or one where the grammar is built manually. Because the grammar is human readable these approaches can mixed.

The transfer engine can handle unification theory, although this is only used in the case of the verb particles, see section refGrammar. The second important part of Avenue is the decoder, this uses the output of the transfer engine, the lattices from figure 2. In this example, the sentence \textit{dat is een pluspunt} generates two translations. The decoder takes these different lattices and using a trigram language model it tries to maximise the likelihood of having a fluent sentence in the target language. In our example it can be seen that there are two translations with a complete span, in this case it is the job of the language model to decide whether “that is an asset” or “that is an asset” should be chosen. Quite likely in real life data there are no complete parses, so the decisions have to be made on the different parts of the sentences, sticking lattices together to form a complete translation. In the previously mentioned rapid prototype experiments for Hebrew and Hindi, a more extensive explanation is provided of how translations are generated in the Avenue project. More information can also be found in (Probst et al., 2003).

3 The Dutch and English Language

Dutch is a language which is interesting for a rapid prototyping project because it is relatively close to English, but there are still a lot of syntactical divergences which need to be solved. In this section we describe the major differences with English which need to be addressed. There are many more differences, but these fall out of the scope of a rapid prototype design, like for example the famous case where Dutch shows cross dependencies. This is so rare in real life data that it is not interesting to address.

In the case where Dutch grammar has more restrictions than English we can take the easy way and just lose information, for example Dutch has genders for the nouns while English does not, so we can easily lose the gender information. Because in most cases Dutch has more restrictions this direction is probably easier in the transfer based paradigm than going from English to Dutch.

The biggest reordering challenge is the fact of Dutch being almost verb final. Typically in Dutch sentences all verbs in a sentence except for the first one, accumulate at the end of the sentence, while in English they are normally positioned after the subject. This requires reordering in most sentences over big distances.
We turn to the use of verbs in most clauses, Dutch turns to a complete verb final sentence. For example: 
Hij zei dat we weg moeten gaan
He told that we away must go
He told that we should leave
Finally verb reordering is required when the Dutch sentence starts with an adverbial clause. In this case in Dutch the verb comes before the subject, while English keeps the traditional SVO ordering.

Another verb related problem in Dutch is the verb particles. In Dutch verb particles are much more common than in English and typically the verb gets separated from its particle, however they form together one word. Because the verb particle tends to go to the complete end of the sentence, this usually requires very long dependencies, something in which MT systems usually do not excel. In English there is a tendency not to make long dependencies over long Noun Phrases with for example the verb particle (the so called Heavy Noun Phrase shift). In Dutch the verb particle cannot move close to the verb when the noun phrases tends to get quite long. In the chosen domain it is not uncommon to see a noun phrase with several attached prepositional phrases.

Another problem with Dutch and with most Germanic languages is the case of compound nouns. Multiple nouns in Dutch run together and for MT to do a good job segmentation should be done on the compound nouns, or the most common compound nouns should be stored as individual entries. Due to lack of time we only addressed this marginally.

Negations in English take an auxiliary verb, where as in Dutch usually only the word for “not” appears.

The last big problem to mention here is how Dutch and English deal with different tenses, especially the past tense. In most cases where English chooses the past tense, Dutch chooses the present perfect. This issue is not addressed in this project and given the domain of political texts this is probably not a major issue, however this might be a point of concern for an extension of the project.

4 Restrictions
The first restriction we will set for the project is a time-limitation. We want to see how well a system will perform which is extremely rapidly prototyped. Therefore we wanted to analyze the project after six weeks. Because of this time restriction other restrictions had to be set. We choose to put some restrictions on the lexicon. We choose not to do any morphological analysis and see if we can get away with this. Typically both English and Dutch do not heavily rely on morphology and since English has less morphology than Dutch we do not have to introduce any morphology.

For Dutch this means the only real problem which remains is the compound noun problem. Some other morphological issues are handled incidently, but at the end this is resolved on a lexical level, which means each stem with different morphology is treated as a different lexical item. Unlike previous projects run in the Avenue project, like for example the Hebrew-to-English system (Lavie et al., 2004), there is no special morphology component present in this system. See section 5 for details on these incidental morphological fixes.

5 Building the Lexicons
The transfer engine takes the lexicon on a format where it requires entries to have a Part of Speech on the both language sides and to have exactly one translation per entry. (On both sides the transfer engine accepts multiple words as a single token). So each different translation needs its own entry. A typical lexicon item would look like: 
VERB::VERB | : \["zijn"] -> \["are"]
which translates zijn in Dutch to one of the correct inflections of to be in English. Other appropriate translations must have a new entry. It is possible in the Avenue engine to annotate words with unification constraints. This however is not used in the Dutch/English lexicons we have developed (accept to distinguish possessive pronouns from non-possessive pronouns, but this could have been done with a separate POS too, and in the small test with verb particles which is discused below), so this feature is not discussed here.

To develop our prototype we want to use existing, publicly available resources. Because to our knowledge there are no publicly available Dutch English word lists with part of speech
Table 1: Created Lexicons

<table>
<thead>
<tr>
<th>Name</th>
<th>Manner</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexicon - cross</td>
<td>automatic</td>
<td>124166</td>
</tr>
<tr>
<td>Top500</td>
<td>manual</td>
<td>724</td>
</tr>
<tr>
<td>Incidental</td>
<td>manual</td>
<td>380</td>
</tr>
<tr>
<td>Morphology</td>
<td>semi-automatic</td>
<td>1698</td>
</tr>
<tr>
<td>verbparticles</td>
<td>manual</td>
<td>477</td>
</tr>
<tr>
<td>Lexicon - last</td>
<td>automatic</td>
<td>66609</td>
</tr>
</tbody>
</table>

indication we have to build this ourselves. At the end we want to build as much from the lexicon possible by automatic means. The core of our lexicon is made by use the Europarl Corpus (Koehn, 2005), which is trained with the GIZA++ tool (Och and Ney, 2000). We created six different lexicons which will be discussed here. In table 1 the different Lexicons are listed, along with the information on whether they were created automatically and how many entries are available in this lexicon.

The Lexicon - cross lexicon is the main part of the lexicon, this lexicon is created by taking the probability files from the GIZA++ results. For each entry here we apply a certain threshold. A lot of the GIZA++ word translations with low probabilities are not translations at all and in transfer-based MT system the items probably only hurt. Furthermore to ensure good translations we require that for the word pair selected a probability above the threshold also occurs in the training data with the Source and Target Language reversed. In the transfer based paradigm we are of course interested in which grammatical role these word function, so basically we want to derive the Part of Speech for the different words. We make the working assumption that words in Dutch and English will have the same POS. Now to assign POS tags to the words we have selected, we use the MBT tagger (Daelemans et al., 2003), on the Dutch side this tagger is trained on the WOTAN-Lite tagset (van Halteren et al., 2001) with which we tagged the complete corpus so all the occurring words have a known POS-tag. Similarly, on the English side, we tagged the data with the WSJ-tagset of the Penn Treebank. Not only do these both tagset have different tags, they are also much more specific than we need for our use. We only need to know the function of the words in the sentence. For example for us it is not necessary to know whether a noun is plural or singal, in our grammar it will have the same behaviour. Therefore we do not want to use the full size of both Tagsets. This also benefits us in our rapid prototyping because our grammars only have to address fewer tags. The WOTAN-Lite tagset has about 175 different tags and the WSJ tagset about 35. We reduce the size of both Tagsets to the following tags: NOUN VERB DET PRON ADJ ADV PREP CD. Now for every word pair selected from the GIZA++ output above the threshold, we take the intersection between the tags it has on the English side and on the Dutch side. For each tag in this intersection we add an entry in this particular lexicon. Setting the threshold at 0.01 this results in almost 125k entries.

The Top500 lexicon was created because it was observed that the previous lexicon was far from perfect and was lacking some basic entries, either because there was no agreement for that specific tag, or (and this reason is especially true for the top 500 words) the words are used a lot and might have very different translations pushing some important words below the threshold. Therefore, counting on Zipf’s Law we decided to make a small manual translation of the top 500 most occurring words on the Dutch language. Staying in our domain, we counted the words in our Europarl corpus and the top500 words are selected. Also the tags AUXV and CONJ were added here, because as a human we can distinguish which verbs are and can be used as auxiliary verbs and which words can used as conjunctions. Some of the words in this top 500 are ambiguous and can have more translations and POS. Therefore there are over 700 entries in this top 500.

The Incidental lexicon is a manually created corpus to fill all the Lexical gaps in the development test-set. This is interesting because it will show how the MT system will operate when all words are covered in contrast with the final test set where words still can have no translation. See section 7 to see how these differences worked out. This is a minor part of the overall Lexicons and does not play an important role.

The Morphology lexicon is created to address some morphological features which really was holding back the overall performance. Most morphology is picked up by the Lexicon -
cross lexicon where different morphologies have different entries. However a typical result of using the GIZA++ word probabilities only, and ignoring the fertility probabilities is that words are one on one aligned. Sometimes morphology causes other alignments. Especially the case was addressed of the comparative and the superlative. In Dutch this is done with morphology in English for longer words this done by more and most. From the GIZA++ word probabilities all words were selected which had the right Dutch suffix and which were aligned to more or most in English and where added as a token in English with more and most respectively added at front.

Another typical mistake which occured frequently in the development set, could be solved by transforming some words from ADJ to ADV. In Dutch, words were selected with an ADJ tag and ending in -sch. On the English side a -ly suffix was added and the entry was added with the appropriate POS. Because both things did not result in a massive lexicon it was possible to browse them by hand and removing wrong entries. There are much more morphologies which could have been addressed, but because of the time limitation these were omitted. However, the showed framework can be used to use some morphologies and add them in this way. So far these two issues seemed to be the most important things to address given the time.

The verbparticles lexicon is created to address the case of the verb particles in Dutch. It was observed in the development test data that there were quite some errors attributable to verb particles. These are especially a problem because they often require very long dependencies to recognise them. In Dutch verb particles are much more common than in English, and the particles often ‘jump’ over the entire sentence.

A Dutch verb like voorzitten (PREP:voor+VERB:zitten, in English before+sit which means to chair (a meeting) ) often splits up where the verb part zitten is inflected and occurs right after the subject and the preposition part occurs at the end of the sentence. For example:

ik zit de europese vergadering voor I sit the european meeting before
I chair the european meeting

Some of these verbs (like this one) occur quite frequently in the selected domain. Now this problem is partly a grammatical problem and partly a lexical problem, because the two parts together form one lexical item. This problem is solved by making entries for the verb particles which translate to an empty token, and the verb which translate to the correct translation. Now here we use coordination of the Avenue transfer engine were we require that this translation is only valid when the verb find its verb particles. On the lexical side this looks like:

VERB::VERB :| ["zit"] -> ["chair"]
verbparticle=voor
while the normal, without the verb particle entry, still remains:

VERB::VERB :| ["zit"] -> ["sit"]

Now in the Grammar we have to ensure that this verb actually does find its verb particles, see section 6.

This lexicon was manually created where the most frequent verbparticle-verbs (over 60 verbs) are selected and put in the lexicon with their different inflections in different tenses, only when the verb and its particle are split. There are cases where they do stay together, for example in Dutch in the present perfect they stay together. In these cases where do stay together it is assumed the correct translation is already in the Lexicon - cross lexicon. Finally this approach for the verb particle lexicon results in almost 500 entries.

The Lexicon - last lexicon is created for the final test. In the development set there are no more lexical gaps present, because they are filled in the incidental lexicon. However it is possible that the final test set does have words which are never seen before. This lexicon is created to extend the lexicon as much as possible, to reduce the lexical gaps in the final test set as much as possible. But because we have never seen the final test before it is always possible that there still are some rare words which are missing even after this lexicon. To make the coverage is big as possible we want to add at least one translation for each word, from the word probability list which not already does occur in one of the previous lexicons. This lexicon is created by taking again all the word pairs from the GIZA++ word probabilities. Here we select the highest ranking translation an add that word with the most
likely Dutch Tag. Because we do not change any entry for already existing words (which could happen if we say lower the threshold in the process for Lexicon - cross were we can get multiple entries), performance cannot go down. We only assure that our coverage is as wide as our training data allows us. This lexicon has about 66k entries.

6 Grammatical issues

The Transfer engine of the Avenue project expects a parallel grammar operating on both the Source and the Target side. Research on automatic acquiring grammars is still work in progress and is not in the state where it could be easily used for rapid prototyping. Therefore it was decided to handcraft the grammar. This is possible because the decoder can handle partial parses, which means still reasonable results can be achieved without a full parse but with partial parses of the sentence. In the time span set for this project it is by all means not possible to write a grammar which does wide coverage full parsing, therefore we focused on getting the grammar parts working in the case where Dutch and English grammar are different.

Because the entire focus of the project is rapid prototyping, only the most frequent occurring grammatical differences can be addressed. For Dutch-English the first important syntactical divergence to address is the feature of Dutch of being almost verb final.

A traditional Phrase Structure analysis is used on the Dutch side, this means that reordering of the verb has to take place in the VerbPhrase part. A non terminal is created which recursively tries to match as much verbs as possible in verb clusters at the Dutch sentence. Rules operating on this Verb clusters try to reorder them to correct position in English.

Example for a transitive verb:

\[
\text{VP} : \text{VP} : \text{[AUXV NP VERBC]} \rightarrow \text{[AUXV VERBC NP]}
\]

This will match sentences in Dutch like:

Ik zou dat doen

I would that do

and reorders it to:

I would do that

The important factor is that the NounPhrase has to be matched completely. If the transfer engine fails to create a full parse of the NounPhrase the transfer engine cannot reorder the fragments. Some considerable time of the project therefore was spent to extend the coverage of NounPhrases as far as possible. This means also recognising NounPhrases which are glued together with commas and “and”s. These can grow quite large and because the transfer engine is trying to list all the different possibilities with different translation (an exponential increase with the sentence length), this takes quite a performance hit when it concerns overall time of translation.

In a lot of clauses Dutch turns to a verb final language. To tackle this problem some special Clause non terminals were created, in which the re-ordering can be done.

For example:

\[
\text{SCLAUSE} : \text{SCLAUSE} : \text{[NP NP VERB]} \rightarrow \text{[NP VERB NP]}
\]

\[(x1::y1)\]

\[(x2::y3)\]

\[(x3::y2)\]

The “problem” is that this re-ordering only work is the grammatical parts (in the case of the example the NounPhrases) are recognised. For most rules where local re-ordering is done it is not so important to get a wide grammatical coverage on sentences, however in this case, because there are very long dependencies at work, it is vital the grammars coverage is sufficient. Therefore again on the case of the NP we need wide coverage on real data.

Another syntactical difference happening a lot in the chosen domain, is how English deals with negations. The transfer engine can handle rules with lexical items instead of nonterminals. This feature is used in to create negations. For the Dutch words to create negations rules are created to add the auxiliary verb in English.

For example:

\[
\text{VP} : \text{VP} : \text{[VERB NP "niet"]} \rightarrow \text{["do not" VERB NP]}
\]

We count on the language model here to pick the right English inflection of the verb, but luckily English has not much morphology on the verbs.

When examining the development test set it was noted that another cause for quite some errors was the case of the verb particles in Dutch, the lexical side solution is discussed in the previous section. Here we used the unification rules of the transfer engine to indicate that the verb is still waiting for its verb particle. This results in two different kind
of rules. In the normal rules we require that there cannot be a verbparticle coming:

\[ \text{VP} : \text{VP} : [\text{VERB NP}] \rightarrow [\text{VERB NP}] \]

\((x_1 \text{ verbparticle}) = \text{UNDEFINED}*)

Now for the verb particles we only have to check that the verb and the particle indeed match and if that is the case the lexical entry for the verb will result in the correct translation of both the verb and its particle:

\[ \text{VP} : \text{VP} : [\text{VERB NP \ VERBPART}] \rightarrow [\text{VERB NP}] \]

\((x_1 \text{ verbparticle}) = c (x_2 \text{ verbparticle}))\]

Finally the grammar contained the basic (di-/in-)transitive rules and the discussed items. Some rules for declarative, sentences questions and how to handle sentences after adverbial clauses. Most of the rules were built trying to improve the development scores.

The final grammar contains 94 rules, of these rules 63 are rules with only non terminals. The other 31 rules are lexical anchored, for example on “niet” (not) to capture negation, and some them capture idioms used a lot in the development test set.

7 Evaluation

The evaluation was done on the select domain of political text of the European union, often these sentences are long and complicated, probably more than day to day sentences. In the selected development and final test data the average sentences length is very close to 30 words per sentence. Furthermore the sentence alignment between the Dutch text and the English text was done by the tools provided by the Europarl corpus which make statistical decisions, which means, and which was observed in the development test set, that not all the parts of the sentences are aligned properly. This was not corrected. The result of this is that the automatic metric probably give lower results because the references text does not always corresponds with the source text.

During the project heavy development was done the on the development test set. This was a set of 150 selected sentences which were not used for the creation of the lexicon. After building all the lexicons except the last one the Lexicon - last lexicon, there was an average one unknown word a sentence. For this development set these lexical gaps were filled manually. Another advantage the development test set has over the final test set, is that some of the more idiomatic expressions are solved in the grammar. This means that the development test result give a nice indication in which direction the final test result will go if the project would be extended with more human labour. The results on the development set are show in table 2. We chose to use the BLEU(Papineni et al., 2002), the F-Measure evaluation (Turian et al., 2003) and the METEOR evaluation (Banerjee and Lavie, 2005) metrics, which were run with the default settings, which means ngram matching for BLEU up to 4 and the exponent set for F-Measure to 2.

At the end of the project, we translated the final test set. These are 1000 sentences which do not occur in either the training or the development set, and run the BLEU, the F-Measure evaluation and the METEOR evaluation metrics. The results are listed in table 3. Because only one reference text is available these metrics are run one only one reference text, this is by all means not ideal, and could possible lower the scores. The total test set was a 1000 sentences. For the metrics, again like the development test set, the default setting was used: For BLEU up to 4-grams, and for the F-measure an exponent equal to 2.

We like to compare this results with previous projects in the Avenue project. Because we use different (and very likely an easier) language pair and because we use quite a different domain for translation, these comparisons are always very hard to make and “hard” BLEU, F-Measure or Meteor numbers do not necessarily make a fair comparison. However on a rough basis they do indicate whether this project is completely failed or whether we do get some encouraging results. Because we assume we have an easier language pair than Hebrew-English our goal is

<table>
<thead>
<tr>
<th>Metric</th>
<th>Precision</th>
<th>Recall</th>
<th>Penalty</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLEU</td>
<td>0.229</td>
<td>0.975</td>
<td>0.164</td>
<td></td>
</tr>
<tr>
<td>F-Measure</td>
<td>0.204</td>
<td>0.202</td>
<td>–</td>
<td>0.203</td>
</tr>
<tr>
<td>Meteor</td>
<td>0.515</td>
<td>0.513</td>
<td>0.134</td>
<td>0.445</td>
</tr>
</tbody>
</table>

Table 2: Development Set Results

\(^{1}\)For a time indication this took 15h45m for the transfer engine and 22h00m for the decoder, so that the entire 1000-sentence test was translated in 37h45m. This was done on a 1.5Ghz Athlon machine
<table>
<thead>
<tr>
<th>Metric</th>
<th>Precision</th>
<th>Recall</th>
<th>Penalty</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLEU</td>
<td>0.201</td>
<td>—</td>
<td>1</td>
<td>0.136</td>
</tr>
<tr>
<td>F-Measure</td>
<td>0.179</td>
<td>0.186</td>
<td>—</td>
<td>0.182</td>
</tr>
<tr>
<td>Meteor</td>
<td>0.481</td>
<td>0.498</td>
<td>0.155</td>
<td>0.419</td>
</tr>
</tbody>
</table>

Table 3: Final Results

to perform as least as good as that MT system measure by the different metrics. Keeping in mind that this prototype was created with less human labour than the previous projects. The final BLEU scores reported for the Hebrew-English MT system was 0.101 with a manually build Grammar. On that project a total of four person-months development was spend. Now as mentioned a different language and a different domain really skew the the automatic metrics, but the difference is big enough to claim that the chosen approach for this system worked quite well in the Avenue project.

8 Conclusions
We have shown how to bootstrap rapidly a MT system in the Transfer Based MT paradigm. For this the lexicon was created using statistical tools. The grammar is mainly a handwritten grammar, but the entire project was finished in 6 weeks human labour. The results are at least comparably with previous rapid prototyping projects done in the Avenue project and possibly better. There are still a lot of unaddressed issues, so if this prototype is taken as a base system it is expected higher score are achievable.

References


