Extraction and Translation of Multi-Word Number Expressions

Anil Kumar Singh

Language Technologies Research Centre
International Institute of Information Technology
Hyderabad, India
anil@research.iiit.net

Abstract. This paper describes a tool for extracting multi-word number expressions, calculating their numerical values, and then generating them into another language, thus translating the expressions in that language. It is based on the fact that such expressions in many languages follow a simple recursive pattern. By changing the language configuration files of our tool, extraction and translation can be done for all the languages which use this pattern. We present the results of testing this tool on seven languages and twenty four language pairs. During testing, the analysis part (finding the numerical value) worked correctly for all the languages for which testing was done. Generation worked correctly for five languages but requires some modification for two languages due to the fact that word forms in languages like Telugu depend on the context. We also discuss some issues which remain and suggest solutions for some of these issues.

Keywords: Number expressions, multilingual processing, information extraction, machine translation.

1 Introduction

One important kind of information that can be extracted from documents for tasks such as question-answering, summarization, etc. are number expressions, i.e., numbers written in words (*one million two hundred thousand three hundred and forty*). It is easy to extract numbers in digits, but not so easy if they are in words. Finding the boundaries of these expressions is a simple task (section-6), but finding their numerical values (section-7) is somewhat more difficult. The ability to do the converse of this, i.e., generating number expressions from numbers in digits (section-8) can also be useful for tasks like machine translation.

Analysis of number expressions shows that these expressions follow a simple recursive pattern (section-5). This pattern is applicable to a large number of languages, including all the major Indian languages. Based on this idea, we implemented an extraction and translation tool for multi-word number expressions. This tool marks the boundaries of such expressions, calculates the numerical

values denoted by them, and can also translate them into another language by generating number expressions in the target language. During testing (section-10), it worked perfectly for five languages including English and Hindi, except for some cases which can be handled with minor modifications.

The results also show that languages like Telugu have some variations which are not completely covered by the tool as implemented so far. Analysis works correctly even for these languages, but generation is partly wrong in the sense that, though the structure of the generated expressions is correct, the word forms may not be correct. This is because for languages like Telugu there are more than one word forms and which one will be used depends on the context.

The tool doesn't depend on any other NLP tools (taggers, parsers) or language resources (dictionaries). The only resource used is a small language configuration file (figure-6).

2 Related Work

The book by Corbett Greville [1] on number expressions in natural languages addresses linguistic issues, but does not directly address the problem being considered in this paper. In speech synthesis, there has been some work [4] on text normalization which sometimes includes conversion of numerals to words. Radzinski [3] discusses the structure of number words and its repercussions for the formal complexity of the set of such words. He has also discussed the case of Chinese for which our approach may not be applicable. The most relevant work is the the IBM-sponsored open source ICU libraries[1] for Java and C/C++. This library, which is a successor to the `java.text.*` for internationalization (part of the Java Development Kit or JDK), contains a class called RuleBasedNumberFormat. This class handles the 'analysis' of number expression, i.e. conversion of number expression to its numeric value. It also handles ordinals, decimal numbers, amounts, etc. One standard for annotation (manual or automatic) of general number expressions is NUMEX[2]. However, it does not handle generation and translation. Part of the section on the linguistic issues (section-11) is based on the book on Telugu grammar by Krishnamurti and Gwynn [2].

3 Overview

The problem of extracting and translating number expressions can be divided into three parts:

- Filtering, i.e., finding the boundaries of the expressions (section-6)
- Calculating the numerical values (section-7)
- Generating the expressions in the target language (section-8)

[1] `http://www.icu-project.org/`
[2] `http://cs.nyu.edu/cs/faculty/grishman/NEtask20.book_1.html`

For extraction, only the first two stages are required. The third stage is needed for translation. The solution to the problem is based on the fact that most languages use a simple recursive pattern (section-5) for number expressions. This pattern can be used to find the numerical value as well as for generating the expressions from a numerical value. The method used for each of these stages has been described in the following sections.

4 Number Words

Number expressions are formed from *number words*. A *number word* can be any word that can occur in a number expression. Thus, in addition to 'real' *number words* like *one*, *twenty*, etc., the list of *number words* will include words like *and*, *decimal*, *point*, *plus*, *minus*, etc. The 'real' *number words* and their numerical values have to be specified for numbers from zero to ninety nine, and also for hundred, thousand, million, etc. For some languages like English, words for numbers from twenty one to twenty nine, thirty one to thirty nine, etc. may not be essential. Such numbers can be formed from other words if the values of twenty, thirty, forty, etc. are provided. But this happens very differently in different languages. For example, in French, eighty is *quatre vingt* (four twenty). In Hindi there is a unique (single) word for numbers from one (*eka*[3]) to hundred (*sO*). It is, therefore, better to provide values up to ninety nine. Hundred, of course, has to be provided as a *special word* (see the next section).

5 Terminology

In this section we define some terms that we have used earlier or will be using later. They are either for different kinds of number words or for numbers which make up the value of a number expression. The terms are:

- **Minor numbers**: Numbers from 1 to 99.
- **Minor words**: Strings (made up of one or more words) for *minor numbers*, e.g., *one*, *thirty*, *fifty-seven*.
- **Major numbers**: Numbers like 1000, 1000000.
- **Major words**: Strings for *major numbers*, e.g., *thousand*, *million*.
- **Special numbers**: 0 or 100.
- **Special words**: Strings for 0, 100, decimal, 'and', etc. These are things which have to be handled as special cases.
- **Number words**: A super-set consisting of *minor*, *major* and *special* words.
- **Central number**: The number (a *major*, a *minor* or a *special* number) at the top of the hierarchy for a given number. For example, for a number like 5567457, the central number will be 1000000. For 107, it will be 100. For 27 it will be 27.

[3] To represent text in Indian languages, we have used one version of the WX notation in which capitalization roughly means aspiration for consonants and longer length for vowels. In addition, 'w' represents 't' as in French *entre* and 'x' means something similar to 'd' in French *de*, hence the name.

– **Central word**: The string (a *number word*) representing the *central number*. For a number expression like *three million two hundred*, it will be *million*.

Note that a *number word* can actually consist of more than one 'words' (*twenty seven*). This is why we have defined it (and other types of number words) in terms of strings rather than 'words'. These strings will be provided in the language configuration file (figure-6).

6 The Pattern of Number Expressions

Number expressions in many languages have a pattern which can be used for finding their numerical values. Languages differ a great deal in the way they express numbers up to one hundred. This difference is so much that we can simplify the problem by saying that from zero to ninety nine, numbers are expressed by unique strings. There are unique strings for the other *number words* as well (we will discuss later how this assumption has to be modified to take care of languages like Telugu). Moreover, there are some special words (more accurately, strings consisting of one or more words) for numbers like zero, hundred, thousand, million, etc. The numbers above ninety nine can be expressed by using these numbers and fitting them into the pattern. Since this pattern is recursive in nature, any large number can be expressed by applying the pattern recursively. The main idea here is that number expressions for values greater than one hundred have a recursive structure as given below:

expression = left_part central_word right_part

The left and right parts also have the same structure. For example, consider the following expression:

(a) *one hundred twenty three million four hundred fifty six thousand seven hundred eighty nine*

It can be broken up as:

(b) *[one hundred twenty three]* million *[four hundred fifty six thousand seven hundred eighty nine]*

The left part can then be broken up as:

(c) *[one]* hundred *[twenty three]*

And the right part as:

(d) *[four hundred fifty six]* thousand *[seven hundred eighty nine]*

The left and right parts of (d) can be further broken up as:

(e) *[four]* hundred *[fifty six]*

(f) *[seven]* hundred *[eighty nine]*

```
Given: A sentence sen and
 the subroutine ToNumber
Task: Find all number expressions
 in the sentence

Trim the sentence and tokenize
 it into an array wds;

while(i < length(wds) ) {
 undefine mwe;
 j = length(wds);
 while(i <= length(wds) && j >= 0 && i < j
  && (not defined mwe && mwe != 'NaN')
 {
  token = '';
  for(k = i; k < j; k++) {
   token = token . wds[k] . ' ';
  }
  j = j - 1;
  Trim the token;
  mwe = ToNumber(token);
 }
 if(defined mwe && mwe != 'NaN') {
  Replace the elements k = i to k = j from
   wds by token and store the value
   of token, i.e., mwe;
  if(i == j) { i++; else { i = j;}
 }
 else i++;
}
```

Fig. 1. Method for Filtering Number Expressions

7 Filtering

The method used for filtering or identifying the number expressions is quite simple. We start with the whole sentence and check whether all the words in it are number words. If yes, we check whether it is a valid number expression by using the method for calculating its numerical value (section-7). If it is not a valid number expression, one word at a time is then removed from the right end of the sentence and the remaining part is checked, till we reach the first word at the left end. Then this process is repeated on the whole sentence minus the first word on the left end. Then two words from the left end of the sentence are removed, and so on. The algorithm for doing this is given in figure-1 using pseudo perl. The subroutine *ToNumber* returns the numerical value of a valid number expression (figure-2).

```
Given: An expression exp and the subroutines
 FormNumber and GetNumberFromWord
Task: If it is a number expression,
 return its numerical value

sub ToNumber(exp) {
 Tokenize exp into an array wds, taking
  care of multi-word 'number words';

 Check the first word for being
  a sign (+ or −) word;

 for(i = 0; i < length(wds); i++) {
  if(wds[i] is not a number word)
  { return undef }
  else {
   add GetNumberFromWord(wds[i])
    to an array nums;
  }
 }

 return sign * FormNumber(nums);
}
```

Fig. 2. Finding the Numerical Value

8 Finding the Numerical Value

Given the structure of number expressions, it is easy to come up with a formula for calculating the value denoted by the expression:

*value = value(left_part) * value(central_word) + value(right_part)*

So, for finding the numerical value, we first find the *number word* that denotes the highest value. This is the word with respect to which we delimit the left and right parts. The numerical value of this word is known (from the language configuration file). The same will be done for the left and the right parts. The recursion will end at *minor words* like 'three', 'fifty seven', etc.

The algorithm for finding the numerical value of a number expression is given in figure-2. It is in the form of a subroutine *ToNumber*, which uses another subroutine called *FormNumber*, (given in figure-3). The subroutine *GetNumberFromWord* basically returns the numerical value of a *number word* as given in the language configuration file.

9 Generation in the Target Language

The method described above for finding the numerical values of number expressions can also be used in a very similar way for generation. Numbers correspond-

```
Given: An array of numbers in digits
Task: Calculate the numerical value
 based on the recursive pattern

sub FormNumber(nums) {
 maxNumberIndex = index of the of
  the highest number;

 Fill array left with numbers on
  the left of maxNumberIndex;
 Fill array right with numbers on
  the right of maxNumberIndex;

 if(left not empty) { l = FormNumber(left); }
 else { l = 1; }

 if(right not empty) { r = FormNumber(right); }
 else { r = 0; }

 return ( (l * nums[maxNumberIndex]) + r);
}
```

Fig. 3. The Subroutine *FormNumber*

ing to *number words* can be directly generated since their values are known. For higher numbers, we first find the *central number*, i.e., a number that can be used as the center around which left and right parts can be put.

The word for this central number is known, since it will be a *number word*. The left part is obtained by dividing the given number by the central number, and the remainder gives the right part. Note that we are dealing here only with integers. Number expressions containing fractions or a decimal will be considered later. The core method will remain the same even for those cases.

The algorithm for generation in the target language is given in figure-4. It is in the form of a subroutine *ToWords*, which uses another subroutine called *FormWords*, (given in figure-5). The subroutine *GetWordFromNumber* returns the *number word* for a *major, minor* or *special number* as given in the language configuration file. The '.' sign is used for concatenation as in perl.

10 Implementation

The tool has been implemented in Java as well as perl. The configuration files ensure that the tool can be easily adapted for different languages without chang-

Given: The number *num* and the subroutine *FormWords*
Task: If it is a number, generate a number expression

sub *ToWords(num)* {
Check the sign (+ or −) of the number;

return *FormWords(num)* after adding the sign word, if required;
}

Fig. 4. Generation in the Target Language

ing the code. Implementation was first done in Java and then the tool was ported to perl. Since the perl implementation is also an object-oriented one (as a perl extension), it was easy to do the porting. Both the implementations consist of three classes:

- *NumberWord*: For representing a number word with its alternative forms. For example, in Telugu, 'one' can be expressed as *oka* or *okaTi*, depending on the context.
- *NumberFilter*: For identifying the number expressions and annotating their numerical values.
- *NumberTranslator*: For translating a number expression from one language to another.

Most of the actual functionality is in the *NumberFilter* class, including all the algorithms given in this paper. The *NumberTranslator* uses one *NumberFilter* object (configured for the source language) to identify a number expression and find its numerical value, and then another *NumberFilter* object (configured for the target language) to generate the expression in the target language.

A *NumberFilter* uses a language configuration file (figure-6) which contains the following information:

- Language name and code
- Case sensitivity (yes or no): In our case this was required for the WX notation used for Indian languages
- Special words (for zero, hundred, 'and', plus, minus, separator and decimal)
- Major words (thousand, million, trillion)
- Number words from one to ninety nine
- XML tag and attribute names for annotating, say *nlp_number*, *val* and *transval*

11 Testing

For testing, we prepared a list of sentences containing number expressions for 21 different type of numbers. This list was used to test calculation of the numerical value and generation (i.e., translation), as well as automatic annotation of

```
Given: The number num and the subroutine
 GetWordFromNumber
Task: If it is a number, generate a number
 expression based on the recursive pattern

sub FormWords(num) {
 if(num >= 0 && num <= 100) {
  return the number word for num;
 }

 cenNum = the number that
  can be a 'central number';

 left = int(num / cenNum);
 right = int(num % cenNum);

 if(left == 0) { l = ''; lsep = ''; }
 else { l = FormWords(l); lsep = ' '; }

 if(right == 0) { l = ''; rsep = ''; }
 else { r = FormWords(r); rsep = ' '; }

 cenWord = GetWordFromNumber(cenNum);
 return (l . lsep . cenWord . rsep . r);
}
```

Fig. 5. The Subroutine FormWords

number expressions. Such lists were prepared for English, Hindi, Telugu, Bengali, Oriya, French and Marathi. Although these lists are handcrafted, the criterion for preparing them was to include all possible standard ways of expressing number expressions. Therefore, the evaluation described in this paper has the limitation that it does not cover non-standard cases. It should be noted that these languages belong to four different families: Indo-Germanic (English), Indo-Aryan (Hindi, Bengali, Marathi and Oriya), Romance (French) and Dravidian (Telugu). This strengthens our belief that most major languages use the same basic pattern for number expressions.

Testing of extraction and annotation of number expressions with *NumberFilter* was done for all these languages. Similarly, testing of translation with *NumberTranslator* was done for 24 language pairs. These pairs were formed by taking a list of sentences containing number expressions in four languages (English, Hindi, Oriya and Telugu) and translating them into all the other six languages.

A sample of the output (for translation) for English-Hindi and Hindi-Telugu is given in figure-7. The results of testing are given in table-1.

Analysis (finding the numerical value of a number expression) worked correctly for all the languages and language pairs. Generation worked correctly for 18 language pairs. For 3 language pairs (Oriya with others), generation was ac-

Language: English	en
Case Sensitivity	0
AltWord Separator	/
...	...
Special Words	
0	zero
-	minus
100	hundred
...	...
Major Words	
1000	thousand
1000000	million
...	...
Minor Words	
1	one
2	two
...	...
31	thirty one/thirty-one
32	thirty two/thirty-two
...	...

Fig. 6. Language Configuration File

ceptable to the native user, but could be made more appropriate based on the context. For 3 other language pairs involving Telugu, the generated expression had the correct structure, but the word forms in it were not correct.

12 Some Issues

The results of testing indicate that there are some issues that require more work. These are summarized below.

12.1 Language-Specific Variations

As indicated previously, in some languages like Telugu, the word forms used in number expressions depend on the context. For example, the root form for 'one' in Telugu is *oka*. But it has different forms depending on the features of the objects being counted:

- *okaDu:* 'one man' (male)
- *okawe:* 'one woman' (female)
- *okaTi:* 'one thing' (inanimate)

It can be observed from figure-7 that the structure of the generated number expressions is correct even for Telugu, only the word forms are incorrect.

A similar thing happens for Oriya, but the generated output based on the root form was rated as 'acceptable' by native speakers of Oriya. For example,

English-Hindi

i gave him <nlp_number val="118" trans="eka sO atTAraha">hundred eighteen</nlp_number> rupees

i gave him <nlp_number val="1316" trans="eka hajZAra wIna sO solaha">one thousand three hundred sixteen</nlp_number> rupees

i gave him <nlp_number val="3000200" trans="wIsa lAKa xo sO">three million two hundred</nlp_number> rupees

i gave him <nlp_number val="123456789" trans="bAraha karodZa cOzwIsa lAKa Cappana hajZAra sAwa sO navAsI">one hundred twenty three million four hundred fifty six thousand seven hundred eighty nine</nlp_number> rupees

i gave him <nlp_number val="-4005" trans="riNAwmaka cAra hajZAra pAnca">minus four thousand and five</nlp_number> rupees

Hindi-Telugu

mEMne use <nlp_number val="118" trans="okaTi nUru paxxenimixi">eka sO atTAraha</nlp_number> rupaye xiye

mEMne use <nlp_number val="1316" trans="okaTi veyyi mUDu nUru paxahAru">eka hajZAra wIna sO solaha</nlp_number> rupaye xiye

mEMne use <nlp_number val="3000200" trans="mupPai lakRa reMDu nUru">wIsa lAKa xo sO</nlp_number> rupaye xiye

mEMne use <nlp_number val="123456789" trans="panneMDu kOTi mupPai nAlugu lakRa yABai Aru veyyi EDu nUru enaBai wommixi">bAraha karodZa cOzwIsa lAKa Cappana hajZAra sAwa sO navAsI</nlp_number> rupaye xiye

mEMne use <nlp_number val="-4005" trans="riNAwmaka nAlugu veyyi aixu">riNAwmaka cAra hajZAra pAnca</nlp_number> rupaye xiye

Fig. 7. Sample Output

'two' in Oriya has the forms *xui, xuiti* and *xuijaNa*. However, even if *xuiti* is more appropriate in a particular context, *xui* is acceptable and is used by native speakers. This is why in table-1 the language pairs having Oriya as the target language have A's (acceptable).

The cases of Oriya and Telugu show that our assumption that *number words* are represented by unique strings needs to be modified to state that '*number words* are represented by one or more *alternative* unique strings, one of which is selected depending on the context'.

We currently provide the list of all the possible forms in the language configuration file. This makes correct analysis possible even for Telugu-like languages. For generation, we are picking up the first word form from the list, say *okaTi* for 'one', which may not be the correct one.

To solve this problem, all the word forms for a number word as well as the features (such as animacy) with which they can be used, should be provided in the language configuration file. From there, they can be read into *NumberWord* objects. The input to the tool can then be sentences annotated with features like animacy and gender (at least for each word that is a noun). Based on the features, the correct word form can then be selected for generation.

	English	Hindi	Marathi	Bengali	French	Oriya	Telugu
English	-	C	C	C	C	A	G
Hindi	C	-	C	C	C	A	G
Oriya	C	C	C	C	C	-	G
Telugu	C	C	C	C	C	A	-

C's	Everything correct	18
A's	Analysis correct, generation acceptable	3
G's	Analysis correct, generation requires modification (structure is correct but word forms or suffixes are not)	3
W's	Analysis correct, generation wrong (structure is wrong)	0
X's	Analysis wrong, therefore, generation also wrong	0

Table 1. Results for 24 Language Pairs

12.2 Ambiguous 'And'

The word 'and' can occur inside as well as outside a number expression. Normally, this doesn't cause any problem because we are ignoring 'and' inside a number expression. But if we have a case like the following:

"Are both *one hundred and seven* **and** *two hundred and eleven* prime numbers?"

Here the second 'and' which occurs between the two number expressions is actually a conjunction, but our tool will take it to be a part of a number expression and it will treat the two expressions as one, thus causing an error. We are working on this problem.

12.3 Tokenization

The tool currently tokenizes with space as the separator. It is possible to improve tokenization by taking care of punctuations etc. This could be done either by normalizing the sentence before being given to the tool, or by modifying the tokenization method used inside the tool.

12.4 Fractions and Decimal

Our tool at present doesn't take care of number expressions having fractions and decimal such as the following:

"The weight of the book is two hundred and five point seven grams."

"Her age is twenty one and a half."

Since the language configuration file has the word for decimal, it won't be difficult to take care of number expressions with a decimal. The expression can be divided into two parts with respect to the decimal and the two parts can be evaluated and added. A little more work will be needed for expressions like:

"The weight of the book is two hundred and five point *seven five* grams."

In such a case, seven and five have to be read as digits of a number and a number representing the part after the decimal can be easily formed from them, but the problem is that the same thing (the part after the decimal) can be written in another way as *seventy five*. We will have to determine which of the ways is being used for the part after the decimal.

Fractions like 'half' can be handled by adding words for them in the language configuration file and slightly modifying the code to use them. However, it will be slightly more difficult to handle cases like the following in a language-independent way:

"Her age is twenty one and three quarters."

13 Conclusion

We described a tool for identifying multi-word number expressions, calculating their numerical values, and translating them into another language. It was tested on English, Hindi, Marathi, Bengali, French, Oriya and Telugu for extraction, and on 24 language pairs formed by these languages for translation. The results show that analysis was correct for all seven languages. Generation was correct for 18 language pairs, acceptable for 3 pairs, and requires modification of the generated expression for the other 3 pairs. We also discussed some remaining issues and suggested solutions for some of them.

14 Future Work

We plan to extend this tool for as many languages as possible. Although we haven't come across any language so far that doesn't use the pattern described in section-5 for number expressions, it is possible that some languages use a different pattern. However, alternative forms of number expressions are likely to be present in many languages. For example, in the ancient Sanskrit epic Ramayana, 14 (the number of years Rama was exiled to the forest) is expressed in many different ways (e.g., 9 and 5). We hope we will be able to take care of most of such cases through some (not very major) modifications. We also plan to implement all the modifications required for the issues discussed in section-11. In the next stage, we will make the tool compatible with the ICU libraries and the NUMEX standard and generalize it handle more general number expressions like quantities or measures.

References

1. Corbett Greville. *Number*. Cambridge University Press, 2000.
2. Bh. Krishnamurti and J.P.L. Gwynn. *A Grammar of Modern Telugu*. Oxford University Press, 1985.
3. Daniel Radzinski. Chinese number-names, tree adjoining languages, and mild context-sensitivity. *Computational Linguistics*, 17(3):277–299, 1991.
4. Richard Sproat, editor. *Multilingual Text-to-Speech Synthesis: The Bell Labs Approach*. Kluwer, Dordrecht, 1997.