Unified Verbalization for Speech Recognition & Synthesis Across Languages

Sandy Ritchie, Richard Sproat, Kyle Gorman, Daan van Esch, Christian Schallhart, Nikos Bampounis, Benoît Brard, Jonas Fromseier Mortensen, Millie Holt, Eoin Mahon

Google

{sandyritchie, rws, kbg, dvanesch, schallhart, nbampounis, benoitb, jfmortensen, millieh, emahon}@google.com

Abstract

We describe a new approach to converting written tokens to their spoken form, which can be shared by automatic speech recognition (ASR) and text-to-speech synthesis (TTS) systems. Both ASR and TTS need to map from the written to the spoken domain, and we present an approach that enables us to share verbalization grammars between the two systems while exploiting linguistic commonalities to provide simple default verbalizations. We also describe improvements to an induction system for number names grammars. Between these shared ASR/TTS verbalizers and the improved induction system for number names grammars, we achieve significant gains in development time and scalability across languages.

Index Terms: speech recognition, text-to-speech, verbalization, factorization, allomorphy, concord, markedness

1. Introduction

Most automatic speech recognition (ASR) and text-to-speech (TTS) technologies employ systems for converting between "written" and "spoken" forms of currencies, dates, measures, numbers, and the like; e.g., '25' → *twenty five*, '£9.50' → *nine pounds fifty*, '11:05' → *five past eleven*. Such categories are sometimes known as *semiotic classes* [1]. This component of the text normalization process is known as *verbalization* and the systems which perform it as *verbalizers* [2, 3]. Verbalizers for most semiotic classes depend on underlying core number names grammars specifying the verbalization of numbers like English *one*, *two*, *three*. We first describe how we modify the induction algorithm in [4] to build these number names grammars across a wider range of languages. We then describe a system which builds on this algorithm to induce verbalization grammars for ASR and TTS systems alike.

Before we can build grammars for semiotic classes such as currencies, we require number names grammars. While semiotic classes typically use similar structures across many languages, the basic underlying number systems of languages typically show a lot of diversity in terms of their structures. Cardinal numbers like *one*, *two*, *three* and ordinals like *first*, *second*, *third* exhibit far more cross-linguistic heterogeneity than most other semiotic classes. For example, languages differ substantially in which numeric base they use, in how they factor large numbers, and in how they prefer to order those factors. This variation, and the recursive complexity of number naming in general, makes them less amenable to templates than other semiotic classes. [4] induce the basic factorization and other features of number names systems from a small set of labeled examples. In Section 2, we describe some extensions to this system which provide better support for factorizations commonly found in the languages of South Asia and sub-Saharan Africa.

Even highly dissimilar languages exhibit considerable overlap in how they verbalize other semiotic classes apart from number names (see also [5, 6]). For example, while languages vary in whether they write a currency symbol before or after the amount—'£5' versus '5€', the word denoting the currency is usually spoken after the amount, as in English *five pounds* (**pounds five*). This is a general property of how numeral words modify nouns across languages, though there are exceptions: in Sinhala and some Bantu languages, for example, the basic word order is the equivalent of *pounds five* (see [7] for an overview). Such exceptions are rare, however. In other words, these structures are typologically *marked*. We use this notion of markedness as a guiding principle in designing template verbalizer grammars. By default, we generate the unmarked verbalization using a simple template, in this case numeral-noun word order *five pounds*. We also include options to specify that a language has some marked feature like noun-numeral word order. Marked verbalizations are also generated by the template verbalizers, but require additional flags and possibly additional information such as morphosyntactic features. These principles and some of our design decisions are set out in more detail in Section 3. We describe a system of customizable, cross-lingual, finite-state template grammars which take advantage of shared tendencies across languages, while also supporting linguistic features like allomorphy, morphological concord, and marked word orders.

With an induced number names grammar and a customized template verbalizer, basic verbalizations of major semiotic classes can be produced without the need for complex custom grammars, paving the way to scaling verbalization to more languages in the future. This research forms part of a wider research effort at Google investigating how language technology can be scaled to more languages quickly [8, 9, 10, 11].

2. Number names induction

Our general process of minimally supervised number names induction is described in [4]. In essence, we use a set of training data consisting of digits mapped to their verbalization (like 123 → *one hundred twenty three*) to induce a finite state transducer (FST) which can produce the factorization for any number.[1] The system is inspired by the insight that number verbalizations can be viewed as arithmetic expressions composed of addends, as in *twenty three*, and multiplicands, as in *four hundred* [12]. As part of an effort to scale this system to a wider variety of languages, we encountered major issues with two marked features of number names: *addend flops* and *multiplicand flops*. In both cases, the system in [4] did not account for the full host of structural phenomena observed across hu-

[1] An open-source data set containing verbalizations for numbers 1 to 100 along with powers of ten (1000, 10000, etc.) is available at https://github.com/google/UniNum.

man languages. We describe these phenomena in Sections 2.1 and 2.2, and then present solutions to them in Section 2.3.

2.1. Addend flops

Addends refer to number expressions which denote a sum. For instance, in the English verbalization *twenty four*, the expression denotes the sum of 20 and 4. In modern English, as in the vast majority of languages we have surveyed, the larger addend always precedes the smaller addend, and the alternative order is ungrammatical under the intended reading (**four twenty*, **four and twenty*).[2] However, in some other languages, including most other West Germanic languages, some addends take the opposite order. In German, for example, 24 is *vierundzwanzig*, lit. 'four-and-twenty'. Such flop systems present an interesting challenge: we need to map from one kind of factorization or arithmetic in the written domain to a different kind in the spoken domain. Abstractly, this process can be represented as a three-stage process consisting of factorization, flop, and verbalization (1):

(1) '24' → '20 4' → '4 20' → *vierundzwanzig*

We are aware of two cases where addends are flopped in all cases: Malagasy and Sanskrit. In Sanskrit, for example, small addends always precede large addends, as can be seen by comparing the German and Sanskrit examples in (2) and (3).

(2) *zwei- hundert- zwei- und- zwanzig*
two hundred two and twenty
'two hundred twenty two' (200 + 2 + 20)

(3) *dvā viṃś ottara dvi śatam*
two twenty and two hundred
'two hundred twenty two' (2 + 20 + 200)

In German, 2 precedes 20, but their sum 22 follows 200, because addend flops occur only with decades (20, 30, 40, ...) and digits (1–9). However, in Sanskrit, 2 precedes 20, and both precede 200; this is a general feature of Malagasy and Sanskrit. Even in very long numbers, the addends occur from smallest to largest, as in (4a), whereas in the Arabic numeral system (and in English and many others) they occur from largest to smallest, as in (4b):

(4) a. 2 + 20 + 200 + 2000 + 2000000 ...
b. ...2000000 + 2000 + 200 + 20 + 2

Unbounded addend flops are challenging to encode with finite-state grammars because the mapping between the digit sequence and the verbalization requires the order of factors to be reversed, and unbounded reversal is not a rational relation [13, p. 191]. Our current system does not handle such unbounded cases, but in future work, this problem can be handled with pre- or post-processing.

2.2. Multiplicand flops

Multiplicands are number expressions which denote a product. For instance, in the English verbalization *two hundred*, *two* and *hundred* denote the product of 2 and 100. In modern English, as in most languages we have surveyed, smaller multiplicands precede larger multiplicands and reversing their order leads them to be interpreted as addends (as in *one hundred two*). However, in Igbo, for example, larger multiplicands always precede smaller multiplicands:

(5) a. *nnari abụọ*
hundred two
'two hundred'
b. *nnari na abụọ*
hundred and two
'one hundred two'

The conjunction *na* 'and' in (5b) is used to signal that *nnari* 'hundred' and *abụọ* 'two' are addends rather than multiplicands. Multiplicand flops also pose a problem for verbalization, both because it requires a reordering of factors, and additional bookkeeping is required to prevent ambiguity:

(6) a. '200' → '2 100' → '100 2' → (5a) or (5b)?
b. '102' → '100 2' → ... → (5a) or (5b)?

We therefore must include some representation of the status of small numbers in multiplicand flop systems to produce the correct spoken form. In our induction algorithm (Section 2.3), we keep these cases separate in two ways. For numbers less than 1000, the system learns that a conjunction like *na* 'and' should be mapped to a reserved symbol '&', so 'hundred and two' maps to '100 & 2' whereas 'hundred two' maps to '100 2', as in (7):

(7) a. '200' → '2 100' → '100 2' → (5a) 'nnari abụọ'
b. '102' → '100 & 2' → (5b) 'nnari na abụọ'

The system can then generalize the resulting arithmetic structures (* 100 2) versus (+ 100 2) to larger numbers less than 1000. For numbers involving larger factors, a boundary marker '|' is inserted between the addends to distinguish, say '1000 | 2' (1002) from '1000 2' (2000).

2.3. Algorithmic improvements

One of the limitations of the algorithm reported in [4] was the fact that all of the 'arithmetic' was handled using grammatical rules. So, for example, the sequence of factors '3 100 20 4' (324) corresponding to English *three hundred twenty four* needs to be combined as '(+ (* 3 100) (+ 20 4))'. In French the number 97, *quatre-vingt-dix-sept* is factored as '4 20 10 7', combined as '(+ (* 4 20) (+ 10 7))'. For German, a reordering rule is needed in order to cover the addend flops described above. All of these require that we provide grammatical rules to cover all cases, including bases of 10 or 20, 'flop' rules with certain powers of ten, and so on. This proved hard to maintain and as we extended to new languages we found ourselves continually adding to the rule set. When we began work on verbalizers for Wolof, a language that makes productive use of base 5 ('8' is '5 3', '900' is '5 4 100'), we decided to try a different approach.

Rather than the finite-state rule-based 'arithmetic' described above, the new induction method uses real arithmetic to try to determine the optimal binary tree to build over a string of number factors, given the numerical denotation. For the French case above, the input would be the pair '4 20 10 7' and its denotation 97, and the algorithm computes all possible trees involving addition and multiplication over the factor string that evaluate to the value 97. This requires a brute-force computation over all possible trees, but since the strings are very short, this is not a problem in practice. In many cases, as in this example, there is more than one possible tree: here both '(+ (* 4 20) (+ 10 7))' and '(+ (+ (* 4 20) 10) 7)' are possible.

[2] Notwithstanding the well-known historical use of *four and twenty* in the English nursery rhyme *Sing a Song of Sixpence.*

In this case we choose the best parse based on analogy with previous parses from *smaller* numbers. In the case of '96' the only available parse of the factors '4 20 16' is '(+ (* 4 20) 16)', and so we choose '(+ (* 4 20) (+ 10 7))' on analogy with that structure.

Since the data which we collected to train the earlier system [4] always includes all numbers from 1 to 200, we can induce grammars for all numbers in that range, and also determine if the expression for 100 includes the word 'one' (as in English *one hundred*) or not (as in Spanish *cien* 'hundred'). Since the training data also includes the hundreds (300, 400, . . . , 900), the remaining numbers from 201 to 999 can then be generated (and checked against examples in the training data) by substituting the terms for the higher hundreds into the template '1xx' covering numbers from 101 to 199. After we have a collection of trees for numbers from 1 to 999, we compile an FST that maps between the strings representing those numbers into the proper factorization of each number in the language.

The above method requires no special rules to cover different orderings of factors, nor do we need any rules to cover different bases: base 5 works as straightforwardly as base 10 or base 20. For numbers involving larger factors (1000 and above), the combination with smaller numbers as multiplicands is controlled by a compile-time flag. In the English verbalization of '246,000', *two hundred forty six* is a pre-modifier of *thousand*, whereas in Igbo the factors can be glossed as *thousand two hundred forty six*, with *two hundred forty six* as a post-modifier. While one could attempt to induce this marked word order from data, it is simple enough to have the grammar developer set the flag appropriately for the target language, and in any case much simpler than the various choices of bases and flop rules that had to be selected by the developer in the earlier system.

3. Verbalization templates

With induced number names grammars, much of the work of verbalizing dates, times, currencies and the like is already done, because these semiotic classes consist mostly of verbalizations for number names. However, typically, each language requires some additional work to handle language-internal variation in verbalization. For instance, a currency expression like '$2.50' can be read as *two fifty* in English but also as *two (US) dollars and fifty cents*. To produce the second verbalization, we need some support for reordering of the numbers and currency symbol in currency expressions.

In order to gather data for new languages, we use a questionnaire asking language consultants to describe all the ways written tokens in various domains can be verbalized (see also [10, 11]). We then need to convert this information to a machine-readable format so that it can be used in verbalizer grammars. Initially [10, 11], this was performed by populating a Thrax [14] grammar template. However, we have moved this system to Pynini [15], a Python library which inherits the functionality of Thrax and can use Python's extensive libraries and testing frameworks.

The new grammar template framework uses a system of classes to instantiate shared variables in a base class. These generalized classes are based on a taxonomy of semiotic classes described in [6]. The base class is then inherited by separate ASR- and TTS-specific subclasses; we need separate subclasses because ASR systems should be able to accept multiple variants (where applicable), while TTS verbalizers should emit exactly one form to speak. In the subclasses, different rules are applied to the variables to produce FSTs which convert written tokens to their spoken form. We then populate a configuration file with information from the questionnaire, and specify the templates and subclasses needed for a particular language or locale. This simple architecture allows us to quickly generate basic verbalizations for languages. Linguists then develop unit tests and make language-specific adjustments to bring the outputs of the template grammars as close as possible to naturalistic verbalizations.

3.1. Description of a unified grammar template: money

The money template verbalizes currency tokens for ASR and TTS. As an illustration, this process is briefly described here, followed by a description of some of the linguistic features of money verbalizations supported by the template.

Written-domain money tokens like '$1.01' are modified in the ASR template so that they contain the same elements as more structured inputs from our TTS text normalization system, such as `integer_part:1`, `currency:usd`, `fractional_part:1`, `currency:usd`.

First, the currency symbol is removed from the beginning of the string:

(8) `$1.01` → `1.01`

Next, the currency code is inserted in the middle and end of the string:

(9) `1.01` → `1 usd .01 usd`

This achieved using a concatenated series of transducers and acceptors, here given as Pynini code:

```
transducer("\$", "") + \
    closure(DIGIT, 1) + \
    transducer("", "usd") + \
    acceptor(".") + \
    closure(DIGIT, 2) + \
    transducer("", "usd")
```

With these changes, the elements of the ASR and TTS inputs now occur in the same order. By concatenating a further series of transducers which convert each of these components to their spoken form, defined differently for ASR and TTS in their respective subclasses, we can produce verbalizations with English word order directly from these inputs, as in Table 1.

Table 1: *Aligning ASR and TTS verbalization*

ASR	TTS		output
`1`	`integer_part:1`	→	`one`
`usd`	`currency:usd`	→	`dollar`
`.01`	`fractional_part:1`	→	`one`
`usd`	`currency:usd`	→	`cent`

As well as producing basic verbalizations like this one, the money template supports various marked linguistic features of money expressions. For example, as stated in the introduction, the currency amount typically precedes the unit, as in *five pounds*, and the template outputs this word order by default. However, languages like Swahili and Sinhala exhibit the opposite word order (the equivalent of *pounds five*). The word order of the output can be adjusted using a boolean parameter that can be flipped to support this type of language.

Another assumption made by the template is that there is a singular/plural split in the verbalization of currency units, e.g. *one* ***dollar*** versus *two* ***dollars***. Languages like Maori and Sundanese don't exhibit this split, and the template supports this alternative style using another simple parameter. Further, languages like Polish and Ilocano include a word for *and* between the main and subunit, e.g. *five dollars* ***and*** *fifty cents* while others do not. The template supports both styles.

Another common feature of money expressions is agreement in gender and other features between numbers and currency units, as in Spanish *una peseta* (feminine) versus *un euro* (masculine). If a language has this property, users can define the features of the currency units, and in conjunction with a number names grammar which can produce inflected forms of numbers, the template will produce the correct inflected form of the number, as in (10):

(10) *doscient-as peset-as y un*
two.hundred-F.PL peseta-F.PL and one.M.SG
céntim-o
cent-M.SG
'two hundred pesetas and one cent'

Here *doscientas* 'two hundred' agrees with the feminine gender of the currency *peseta*, while *un* 'one' agrees with the masculine gender of *céntimo*. The only requirement of the user to produce this verbalization is to specify the gender or other features of the different units of currency.

Tokens denoting large amounts of money can feature decimals, exponents, and big powers of ten, as in '3.5×10^2 bn'. The money template uses another decimal template plus a few more rules to support verbalization of this kind of token, as in *three point five times ten to the two billion dollars*.

The templates can also be customized to support other small differences from the basic verbalizations produced by default. For example, in Hebrew, tokens like '$0.50' can be verbalized as the rough equivalent of *zero dollars fifty*, and as *half a dollar*. Rather than developing a custom verbalizer just to support this single alternative, users can specify 'fix-up' FSTs which are composed with the FST produced by the template.

3.2. Advantages of unified approach

There are some significant advantages to sharing verbalizer templates between ASR and TTS: (1) The system reduces code duplication, as both ASR and TTS verbalizations are produced with a single set of files. Comparing a typical set of Thrax verbalizers for the money class with the new system, we see a reduction from around 200 lines of custom handwritten code to just 25 lines specifying the constants (words for *dollar* and the like) and the template parameters to be used. (2) Having a template-based system that is robust to verbalization structures across many of the world's languages paves the way for us to scale faster to more languages, as development cost is significantly reduced by using these templates. The templates enable developers to flip parameters to match the grammatical structures of the target languages, and they make it easy to share verbalization knowledge across subsystems. They also make it easy to inject knowledge elicited from speakers of the language.

One specific simplification is that the system offers a principled approach to variation in the spoken domain. Verbalizers for ASR are normally designed to output many variants, since they need to match whatever is said. On the other hand, TTS verbalizers need to produce exactly one output for any given input. In our system, TTS clients can select alternative verbalizations, normally available only for ASR, by selecting a verbalization 'style'. For example, the money template produces the following variants for '$1.50':

(11) a. one United States dollar and fifty cents
b. one dollar and fifty cents
c. one dollar fifty
d. one fifty

In the unified system, the rules which produce these variants can be shared across the two platforms. For TTS, we specify a style number for each of these outputs, and clients of the TTS system can select among them for different purposes. For example, if an end user asks *What is ten dollars in pounds?*, the TTS client who produces the response message might choose a style like (11a), as in *ten United States dollars is seven British pounds and sixty pence*, in order to state unambiguously which currencies they are referring to. With a single set of rules, we can specify both TTS styles and ASR variants, a significant saving in code complexity and development time.

Another simplification is out-of-the-box support for allomorphy and morphological concord. Previously, each language required complex custom architecture to produce inflected variants of numbers and other tokens. With templates and number names grammars which can produce inflected forms of numbers, it is possible to support phenomena such as noun-modifier agreement without the need for custom grammars, as demonstrated by example (10) from Spanish. This substantially reduces the amount of custom code required.

4. Future work

There are several areas of the system which will benefit from further development. First, developing tools to parse the questionnaires obtained from language consultants would allow grammars to be generated more rapidly and with less human effort. We also could leverage other data sources such as the Unicode Common Locale Data Repository [16] to obtain some of the template parameters for languages. Second, we could extend our coverage by creating templates for more semiotic classes. Finally, we could further improve our templates for increased robustness to typological diversity (such as the Malagasy and Sanskrit addend flops described in Section 2.1).

5. Conclusion

We have described a system of induced number names grammars and template verbalizers which enable us to generate basic verbalizer systems for many languages. In this system, we support marked features of number names systems such as flop arithmetic and morphosyntactic phenomena within semiotic classes such as noun-modifier agreement, while maximizing consistency across languages using templates. This enables us to develop these verbalization systems for new languages faster, and lowers the maintenance burden per language. Our system handles both simple unmarked verbalization formats as well as more complex linguistic features without the need for substantial language-specific architecture.

6. References

[1] P. Taylor, *Text to Speech Synthesis*. Cambridge: Cambridge University Press, 2009.

[2] H. Sak, F. Beaufays, K. Nakajima, and C. Allauzen, "Written-domain language modeling for automatic speech recognition," in *INTERSPEECH*, 2013, pp. 675–679.

[3] ——, "Language model verbalization for automatic speech recognition," in *ICASSP*, 2013, pp. 8262–8266.

[4] K. Gorman and R. Sproat, "Minimally supervised number normalization," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 507–519, 2016.

[5] R. Sproat, A. W. Black, S. Chen, S. Kumar, M. Ostendorf, and C. Richards, "Normalization of non-standard words," in *Computer Speech and Language*, 2001, vol. 15, no. 3, p. 287–333.

[6] D. van Esch and R. Sproat, "An expanded taxonomy of semiotic classes for text normalization," in *INTERSPEECH*, 2017, pp. 4016–4020.

[7] M. S. Dryer, "Order of numeral and noun," in *The World Atlas of Language Structures Online*, M. S. Dryer and M. Haspelmath, Eds. Leipzig: Max Planck Institute for Evolutionary Anthropology, 2013. [Online]. Available: https://wals.info/chapter/89

[8] M. Chua, D. van Esch, N. Coccaro, E. Cho, S. Bhandari, and L. Jia, "Text normalization infrastructure that scales to hundreds of language varieties," in *LREC*, 2018, pp. 1353–1356.

[9] M. Prasad, T. Breiner, and D. van Esch, "Mining training data for language modeling across the world's languages," in *SLTU*, 2018, pp. 61–65.

[10] K. Sodimana, P. D. Silva, R. Sproat, T. Wattanavekin, A. Gutkin, and K. Pipatsrisawat, "Text normalization for Bangla, Khmer, Nepali, Javanese, Sinhala and Sundanese text-to-speech systems," in *SLTU*, 2018, pp. 147–151.

[11] K. Sodimana, P. De Silva, S. Sarin, O. Kjartansson, M. Jansche, K. Pipatsrisawat, and L. Ha, "A step-by-step process for building TTS voices using open source data and frameworks for Bangla, Javanese, Khmer, Nepali, Sinhala, and Sundanese," in *SLTU*, 2018, pp. 66–70.

[12] J. R. Hurford, *The Linguistic Theory of Numerals*. Cambridge: Cambridge University Press, 1975.

[13] R. Sproat, *A Computational Theory of Writing Systems*. Cambridge: Cambridge University Press, 2000.

[14] B. Roark, R. Sproat, C. Allauzen, M. Riley, J. Sorensen, and T. Tai, "The OpenGrm open-source finite-state grammar software library," in *ACL*, 2012, pp. 61–66.

[15] K. Gorman, "Pynini: A Python library for weighted finite-state grammar compilation," in *ACL Workshop on Statistical NLP and Weighted Automata*, 2016, pp. 75–80.

[16] Unicode, Inc, "Unicode Common Locale Data Repository," http://cldr.unicode.org/, 2019.