Text Normalization Infrastructure that Scales to Hundreds of Language Varieties

Mason Chua, Daan van Esch, Noah Coccaro, Eunjoon Cho, Sujeet Bhandari, Libin Jia
Google LLC, 1600 Amphitheatre Parkway, Mountain View, CA 94043
{aftran, dvanesch, noahc}@google.com

Abstract

We describe the automated multi-language text normalization infrastructure that prepares textual data to train language models used in Google's keyboards and speech recognition systems, across hundreds of language varieties. Training corpora are sourced from various types of data sets, and the text is then normalized using a sequence of hand-written grammars and learned models. These systems need to scale to hundreds or thousands of language varieties in order to meet product needs. Frequent data refreshes, privacy considerations and simultaneous updates across such a high number of languages make manual inspection of the normalized training data infeasible, while there is ample opportunity for data normalization issues. By tracking metrics about the data and how it was processed, we are able to catch internal data preparation issues and external data corruption issues that can be hard to notice using standard extrinsic evaluation methods. Showing the importance of paying attention to data normalization behavior in large-scale pipelines, these metrics have highlighted issues in Google's real-world speech recognition system that have caused significant, but latent, quality degradation.

Keywords: language model, text normalization, internationalization, industrial systems, data mining, data processing, scale, less-resourced languages

1. Introduction

As technology adoption increases, products need to support ever more language varieties (ITU/UNESCO Broadband Commission, 2017). For example, there is a clear need for spell-checking and keyboards in hundreds or even thousands of language varieties. These applications typically require a language model, as do more advanced technologies like speech recognition systems. It would not be feasible to maintain separate training pipelines for all these use cases multiplied by all these language varieties, so we use one unified pipeline that is flexible enough to train all the models. The training includes automatic, configurable preprocessing for each data set, allowing text from a variety of sources to be normalized into the domains appropriate for keyboard prediction and speech recognition (Schwarm and Ostendorf, 2002; Ju et al., 2008; Scannell, 2014).

2. Training Pipeline

Each language model is produced by interpolation of component models, which are trained on individual data sources. The training set for the typical component model begins as publicly crawled or logs data. It is processed by language identification classifiers and privacy-enhancing systems (including for anonymization). The data is then filtered and edited by a sequence of normalizers (section 2.1) specific to the use case and language variety.

The creation of the interpolated language model is controlled by a configuration file specifying how to find and normalize each data set, as well as global model options like interpolation method. The configuration system separates language-specific data and its preprocessing from the training algorithms used. A single training binary can therefore train a model for any supported language, and new languages can be added simply by adding a configuration file, without changes to the trainer logic. Due to this automation and abstraction, the amount of manual work required to add new languages or train new models does not depend on the size of the data sets.

Having a single training pipeline controlled by language-specific configuration files is critical to operating at large scale, across many language varieties and use cases. For any given language, there may be half a dozen different language models, each with their own configuration file, reflecting varying use cases, including keyboard input (Hellsten et al., 2017), on-device or cloud-based speech recognition, handwriting recognition, and more. In total, we currently maintain about 2,700 distinct configuration files (not all of which are used for production applications). Each of these language models may include more than a dozen different types of data sources.

2.1. Text Normalization

One critical part of the configuration files is the specification of the normalizations to apply to the textual data. Each training example is potentially modified or discarded by several functions, usually implemented as Thrax grammars (Roark et al., 2012), including the data cleaning in section 2.1.1, annotators required for class-based language models (Vasserman et al., 2015; van Esch and Sproat, 2017), and enforcers of spelling consistency. Other types of normalizers deal with language identification, privacy enhancement and anonymization, and offensive content filtering. The most complex type of normalizer uses language models trained by this same pipeline, e.g. to ensure consistent string variation (section 2.1.2) or capitalization.

2.1.1. Thrax-based Normalization

Most of the data preprocessing is done by grammars written in Thrax, which was chosen for its composability with other finite state transducers used in our systems (Mohri et al., 2008). In order to support several languages without a proportional increase in engineering cost, we use a

Work done while at Google LLC.

shared language-independent normalizer template of about 200 lines of code. It can be instantiated for each language by adding a few dozen more lines of code that define language-specific character sets and rewrite rules, and call upon the language's pronunciation models and word lists. The language-independent template is the composition of the following rewrite rules, with only minor differences between speech recognition (presented here) and keyboard.

1. Ensuring that tokens are always separated by a single space. Turns "hi there" into "hi there".
2. Applying any language-specific formatting fixes defined when the template is instantiated for a specific language. There are usually no fixes, but in Azerbaijani and Turkish, for example, we use this step to make sure dotted and dotless I are preserved by the language-independent lowercaser in the next step.
3. Lowercasing the sentence in languages that have case. For example, "Hello, Dr. Nduom, how are you?" becomes "hello, dr. nduom, how are you?"
4. Discarding any examples that are not associated with a pronunciation by the language-specific verbalizer. For example, "bbbbbbbbbbbx a cat stepped on the keyboard".
5. Detaching punctuation from words. This step simplifies upcoming steps and makes sure the language model recognizes all occurrences of a vocabulary item as the same regardless of what sentence-level punctuation it appears next to. For example, "hello, dr. nduom, how are you?" becomes "hello , dr . nduom , how are you ?"
6. Reattaching punctuation that is part of the word, like in the English abbreviations "no." and "esq." For example, "hello , dr . nduom , we shipped a no . 2 pencil to peppler st . yesterday ." becomes "hello , dr. nduom , we shipped a no. 2 pencil to peppler st. yesterday ." This step and the previous one are separate for the sake of code readability.
7. Deleting each freestanding punctuation symbol unless the speech recognizer for that language supports inputting the symbol by saying its name ("hello comma Doctor Nduom comma how are you question mark").
8. Making the spelling consistent for some common or product-relevant words. For example, correcting "youtobe" to "youtube" or "color" to "colour" in many varieties of English. This step uses hand-curated lists.
9. Restoring proper capitalization for internal symbols that will be expanded during recognition. For example, "meet at $time" becomes "meet at $TIME".
10. Applying any language-specific rewrites that are more easily expressed on normalized data. In Italian, for example, we ensure that numbers are written as Roman numerals in certain street names.
11. Deleting any extra spaces left over due to step 7.

The rewrite rules are typically much less than twenty lines of code, possibly reusing shared character classes, definitions or word lists. Step 4, for example, is implemented as the PASS_ONLY_VALID_SENTENCES acceptor below.

```
core_token = Optimize[
  config.GRAPHEME+ |
  config.LOANWORD_GRAPHEME+ |
  config.VERBALIZABLE];

token = Optimize[
  (config.INITIAL_PUNCTUATION?
  core_token config.FINAL_PUNCTUATION*) |
  non_terminal];

export PASS_ONLY_VALID_SENTENCES =
Optimize[
  (token u.space)*
  token
  config.FINAL_PUNCTUATION*];
```

It defines an acceptor of zero or more word-and-space pairs followed by an additional word and zero or more sentence-final punctuations. `config` is the language-specific configuration, which must provide grammars defining valid punctuations and lists of allowed words.

2.1.2. Learned Normalization

Keyboards and speech recognition systems should consistently produce text in a particular style in order to be useful to the user. Since the training data includes Web text and search queries, there will be typos and intentional variation in spelling, capitalization and punctuation. We use a learned normalizer, called the *string variation normalizer*, to remove some of these inconsistencies and make the training data match the style of a reference corpus.

While the individual actions performed on the string are expressed as Thrax grammars, the string variation normalizer allows us to specify them without context. Instead, it learns when to apply them from the reference corpus. For example, for some output like "During the Renaissance period, people visited B&Bs on Valentine's Day", the normalizer had to decide whether to capitalize "Renaissance", whether to write "period" or ".", whether to use an ampersand or "and" in "B&Bs", whether to capitalize "Valentine's" and "Day", and whether to use an apostrophe in "Valentine's". The string variation normalizer evaluates all possible sequences of edits of this sort and chooses the result with the best score according to a language model trained on the reference corpus.

The language model used by the string variation normalizer is trained using the same pipeline as the end-use language model itself, except with a corpus that is cleaner and usually human-curated. It might seem overcomplicated to introduce a language model to normalize the text for training the final language model, but this complexity is outweighed by the added value in terms of accuracy and stylistic consistency. The use of the string variation normalizer typically results in at least a 1% relative decrease in word error rate and a 1% relative increase in sentence accuracy for speech recognition when evaluated on a voice search test set. In

one case, the recognizer used by voice search had relative changes of -5.7% word error rate and +2.1% sentence accuracy.

3. Monitoring to Allow for Scaling

The complexity and diversity of normalization processes described in section 2.1 are justified by user-facing quality gains, confirmed by experiments showing higher accuracy on extrinsic evaluations. However, this complexity also creates opportunities for data normalization issues that are not easily caught by extrinsic evaluation. Incorrect removal of punctuation, for example, is more easily caught by inspecting the data before and after normalization. There are also opportunities for data to become corrupted at its source before it reaches the normalization pipeline, for example due to incorrect character encoding or unexpected control symbols.

These issues could be caught by carefully inspecting the intermediate data between every normalization step for every new model trained. However, we avoid this approach for three reasons: privacy considerations; the sheer number of language models across languages (each one with many data sets that undergo many steps of normalization); and wanting staff to be able to develop improvements to models even when they do not speak the language involved.

We instead automate the inspection process by logging metrics during training, exemplified in the next section. The monitoring of these metrics has allowed us to identify normalization issues in several production language models. In US English, for example, the fixing of identified issues has resulted in a 6.6% relative word error rate decrease on a test set consisting of Google Home queries. Without automatic monitoring, problems in languages that are less familiar to research and engineering staff would be even more likely to occur.

Due to the size of the data sets, which can contain tens of billions of sentences, even simple metrics, like the size of the corpus, require parallel computation enabled by Flume (Chambers et al., 2010). Operations are automatically combined at runtime to avoid repeated iterations over the data.

3.1. Possible Data Issues and their Associated Monitoring

This section describes some potential data corruption issues that can arise and how they are caught using logged metrics.

Low quality for one dialect or region. When a language is spoken in more than one region or country, we often include multiple regions in a single model. Selecting training data from the right regions warrants extra care because changes can severely impact quality for populations of users in ways that can only be detected with expensive region-specific evaluations or test sets that are not always available. We therefore tag each piece of data with the region it came from, usually at the country level, and audit the histogram of tags for unexpected changes.

Language change. The language models are automatically re-trained with fresh data every few weeks in order to keep up with some types of language change, broadly construed. This process works well for neologisms, trending search terms and seasonal variations in the distribution of topics being written about. Changes that involve new characters, however, require manual intervention, because the normalization process (section 2.1.1) filters out unrecognized characters. If a new currency symbol is added, like the Indian rupee sign was in 2010 (Everson, 2010), we want to notice and expand the currency character lists used by our Thrax grammars. There are an infinite variety of possible external issues that could lead to malformed data. For example, if a popular keyboard were to have a bug where it used the wrong (but visually identical) Unicode codepoint, we would want to be alerted to fix the data, rather than discarding it in step 4.

To help identify character normalization issues, we log the number of times each character occurs before and after normalization. A sudden or gradual increase of this count for a particular character is reason to investigate whether there has been a change to the writing system. We also log how many vocabulary items each character appears in. Unexpected changes can indicate issues in the normalizer pipeline. A bug that causes sentence-level punctuation to be attached to tokens, for example, might be caught due to an increase in the number of vocabulary items that contain punctuation characters.

Deletion during normalization. Normalizers, explained in section 2.1, have the power to discard or corrupt all the data before it reaches the trainer. If Web markup is accidentally included in the data, for example, normalizers that assume markup has been removed will discard the data due to unrecognized tokens. To detect this type of issue, we log how many times each normalizer discards, accepts or edits the input. Excessive discarding or sudden changes in the distribution of actions suggest a bug in the normalizer logic or an incorrect assumption about its inputs. The counts are automatically presented to experimenters in a stacked bar chart (figure 1).

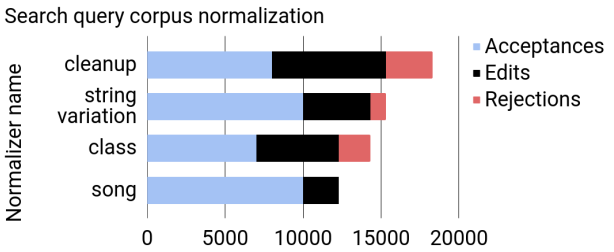

Figure 1: A made-up example of the horizontal stacked bar chart generated for each data set used by each model. The three colors of each bar show how many training examples were passed as-is, edited and deleted by each normalizer, respectively. The four normalizers were applied to the data in top-to-bottom order. Since the output of a normalizer is the input of the next normalizer, each bar is as long as the non-rejection part of the bar above it.

Data from the wrong language, script or corpus. Data might be annotated with the wrong language code due to quality issues with the internal language identification system or end-user devices misreporting their locale. Since

each corpus usually represents a different data source (such as voice search queries or crawled text), this issue is more likely to happen in a single corpus than all corpora at once. A model trained with significant amounts of data from the wrong language will be noticed before deployment due to worse sentence accuracy and word error rate. But those metrics do not directly show what happened, only that there is a problem. So we log the number of vocabulary items that are unique to each corpus. A high number may suggest incorrect language data in that corpus.

Similarly, normalizer grammar bugs can result in training data with non-canonical characters. For example, Greek text might contain Unicode Greek math symbols instead of standard Unicode Greek letters, which could decrease quality by creating both incorrect output text and more sparsity of contexts in the language model. So we also report character-based metrics, displaying each character's Unicode name (for example, "GREEK SMALL LETTER IOTA WITH DIALYTIKA AND TONOS") with its count in the data and the model's vocabulary. This also allows for faster detection of tokenisation issues that result in sentence-level punctuation attached to words.

Although we cannot foresee all data misclassification mistakes, many will involve sudden or gradual changes in corpus size. If, for example, all assistant-type queries were to start being classified as maps-type queries, every data refresh would shrink the assistant-type corpus and grow the maps-type corpus. Language model interpolation would also become less effective due to the apparent disappearing distinction between user behavior in the two products. So we monitor corpus sizes and interpolation weights for sudden changes and unexpected trends.

Inadequate pronunciation information. A speech recognizer is only as good as its implicit or explicit pronunciation model that associates phoneme sequences with text. To better diagnose quality issues, we log how many vocabulary items get a pronunciation in the recognizer purely by consulting a human-curated pronunciation dictionary, coupled with Thrax verbalization rules (Sak et al., 2013a; Sak et al., 2013b). The remaining items rely on some learned model (Wu et al., 2014; van Esch et al., 2016). If the number of words receiving a manual pronunciation is high, quality issues are unlikely to be due the learned model.

4. Conclusion

We described our text normalization infrastructure that trains language models for speech recognition and keyboard prediction in hundreds of language varieties. Most of its internals are language-independent, with language-specific behavior specified in configuration options, training data and a limited portion of normalizer grammars. The sharing of most infrastructure enables a common set of auditing techniques that have allowed us to catch text normalization errors across languages.

5. Bibliographical References

Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. (2010). Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM.

Everson, M. (2010). Proposal to encode the INDIAN RUPEE SIGN in the UCS. In *Universal Multiple-Octet Coded Character Set*. International Organization for Standardization.

Hellsten, L., Roark, B., Goyal, P., Allauzen, C., Beaufays, F., Ouyang, T., Riley, M., and Rybach, D. (2017). Transliterated mobile keyboard input via weighted finite-state transducers. In *Proceedings of the 13th International Conference on Finite State Methods and Natural Language Processing (FSMNLP 2017)*, pages 10–19.

ITU/UNESCO Broadband Commission. (2017). The State of Broadband 2017: Broadband catalyzing sustainable development. *Broadband Commission for Sustainable Development*.

Ju, Y.-C., Odell, J., and Ju, Y. C. (2008). A language-modeling approach to inverse text normalization and data cleanup. International Speech Communication Association, September.

Mohri, M., Pereira, F., and Riley, M. (2008). Speech recognition with weighted finite-state transducers. In *Springer Handbook of Speech Processing*, pages 559–584. Springer.

Roark, B., Sproat, R., Allauzen, C., Riley, M., Sorensen, J., and Tai, T. (2012). The opengrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pages 61–66. Association for Computational Linguistics.

Sak, H., Beaufays, F., Nakajima, K., and Allauzen, C. (2013a). Language model verbalization for automatic speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8262–8266. IEEE.

Sak, H., Sung, Y.-h., Beaufays, F., and Allauzen, C. (2013b). Written-domain language modeling for automatic speech recognition. In *INTERSPEECH*, pages 675–679.

Scannell, K. (2014). Statistical models for text normalization and machine translation. In *Proceedings of the First Celtic Language Technology Workshop*, pages 33–40.

Schwarm, S. and Ostendorf, M. (2002). Text normalization with varied data sources for conversational speech language modeling. In *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, volume 1, pages I–789. IEEE.

van Esch, D. and Sproat, R. (2017). An expanded taxonomy of semiotic classes for text normalization. *Proc. Interspeech 2017*, pages 4016–4020.

van Esch, D., Chua, M., and Rao, K. (2016). Predicting pronunciations with syllabification and stress with recurrent neural networks. In *INTERSPEECH*, pages 2841–2845.

Vasserman, L., Schogol, V., and Hall, K. (2015). Sequence-based class tagging for robust transcription in asr. In *Sixteenth Annual Conference of the International Speech Communication Association*.

Wu, K., Allauzen, C., Hall, K., Riley, M., and Roark, B. (2014). Encoding linear models as weighted finite-state transducers. In *Fifteenth Annual Conference of the International Speech Communication Association*.