Lexical Normalisation of Twitter Data: Experiment and Reflection

Donghan Yang

1. Introduction

Twitter is the world famous social platform where people "tweet" (a 140-chacater maximum length text message), and these messages are usually full of spelling errors, acronyms, emoticons, abbreviations and messy characters, special characters, numbers, etc. Trying to make sense of the twitter message is both a challenging and rewarding task as it enables us to utilize the twitter data more efficiently and conduct large scale data analysis include sentiment analysis, etc.

2. Task description

Natural language processing usually begins with tokenization (extracting distinct words from a corpus). Also, tokens are usually down-cased so that they are easier to deal with and we need to filter out the messy characters that are not valid words as well. This project however will focus on lexical normalisation of the tweets rather than pre-processing of text data. Lexical normalisation is to canonicalize the tokens from its twitter form to the intended canonical form, a few examples to illustrate:

Token samples and its classification	**Canonical form**
new ("IV=in vocabulary", common adjective)	new
Lebron ("OOV=out of vocabulary", name)	Lebron
ily (OOV, abbreviation, one-to-many normalisation)	I love you
2morrow (OOV, phonetic substitution)	tomorrow
lol (OOV, acronym)	laughing out loud
acheive (OOV, spelling error)	achieve
#LA; http://; @NYC; >_< ("NO = not a candidate")	Not considered in normalisation process

The given data sets and descriptions are extracted from *Lexical normalisation of short text messages: Makn sens a #twitter* (Bo Han, Timothy 2011) and it contains several files include raw tweets files (one tweet per line), tokens files (extracted from the raw tweets files, labelled/unlabelled, labelled file contains canonical forms for evaluation and the classification of the token), dictionary file (for identifying IV words).

The project is aimed at evaluating several approximate string matching techniques using several evaluation metrics on the labelled dataset and inferring some knowledge from the whole evaluation process. Then the report will conclude by discussing the design and implementation of a complete lexical normalisation system.

3. Related work

Despite its difficulty and complexity, there have been some work done in this area. Bo Han and Timothy Baldwin (2011) proposed that the key to lexical normalisation is to distinguish between correct OOV words that are not yet in the reference library and other "ill-formed" tokens such as typos, abbreviations, phonetic substitutions, etc. Bilal Ahmed (2015) presented a set of normalisation procedures for "ill-formed" OOV tokens include approximate string matching, phonetic matching, context matching, etc. Dmitry Supranovich and Viachaslau Patsepnia (2015) submitted a system that achieved very good performance during the ACL 2015 workshop on noisy user-generated text. They first built a lexicon based on the lexicon dataset (Han 2011, Liu, 2011) and a social media abbreviation list (Beal, 2015) for dictionary look up and trained a conditional random field model to determine whether the token needs to be normalised or not based on features such as word length, number of vowels, context, alphanumeric presence, etc.

4. Approach

4.1 Data set

The given labelled token dataset consists of 8841 tokens and 6418 of them are "IV", 542 are "NO", which leaves us 1881 tokens considered as "OOV". Using python NLTK[1] library, we can easily obtain a frequency distribution of the OOV tokens and can be sorted as well:

```
import nltk
import operator
frequency = nltk.FreqDist(OOV)
sorted_x = sorted(fdist.items(),
key=operator.itemgetter(1),reverse=True)
```

Sample OOV tokens with frequency in descend order:

[('u', 184), ('lol', 80), ('n', 35), ('im', 32), ('w', 22), ('dont', 22), ('jus', 21), ('ppl', 20), ('da', 19), ('lmao', 17), ('2', 17), ('smh', 16), ('d', 16), ('haha', 14), ('aint', 13), ('y', 12), ('cuz', 11), ('o', 10), ('r', 9), ('niggas', 9), ('cont', 9), ('dat', 9), ('lil', 8), ('lmfao', 8), ('yu', 8), ('wat', 7), ('tht', 7), ('p', 7), ('c', 6), ('goin', 6), ('nigga', 6), ('yall', 6), ('kno', 6), ('thats', 6), ('fuckin', 6)…]

A quick glance suggests that single-letter abbreviations and some acronyms are appearing at a very high frequency, some of them involve one-to-many normalisation which is impossible to achieve by using

[1] http://www.nltk.org/

regular spell-checking methods alone and require lexicon. If we use accuracy as the evaluation metric and just use original OOV tokens as the correct answers then we get 35.14%, which suggests that 35.14% of the OOV tokens do not need to be normalised.

4.2 Levenshtein Distance[2]

Spelling errors are very common among social media and it is when the approximate string matching comes in handy. Levenshtein distance is a special case of the GED since the original GED is not technically a "distance" measure as it allows negative values. Levenshtein distance measures the similarity between two strings by assigning a score/distance for different character operations:

Match (0), Insert (+1), Delete (+1), Replace (+1)

The total distance is number of edits required to transform a string.

After removing the tokens which its canonical form is not in the reference collection, by simply applying the Levenshtein distance algorithm to the dataset and finding a set of matches based on its Levenshtein distance to the given token results in a precision-recall matrix as follows:

Levenshtein	Precision	Recall
1 edit distance	1.84%	39.07%
2 edit distances	0.29%	75.89%
3 edit distances	0.04%	89.81%
4 edit distances	0.01%	94.02%

Precision-recall trade-off is obviously shown in the table as if we increase the Levenshtein distance we generate more potential candidates, the precision will worsen and the recall will improve since there is a larger probability that the potential candidates will contain the true canonical form.

Token	Canonical form	GED
u	you	✓ (contain)
r	are	✓ (contain)
2	to	✓ (contain)
soooo	so	×
yesssss	yes	×
diiiiiirty	dirty	×

The immediate observation would be that there are many tokens that contain many duplicates of characters and adding unwanted edit distances so that Levenshtein distance algorithm performs poorly on those tokens.

Overall, the precision is poor and too many candidates are generated. The next phase of experiment would be to narrow down the candidates list and improve the precision.

[2] https://pypi.python.org/pypi/editdistance

We can pre-process the tokens by removing the consecutive duplicating characters within a token (only for tokens with 3 or more consecutive duplicating characters) and re-evaluate the performance of the Levenshtein distance algorithm.

Levenshtein	Precision	Recall
1 edit distance	1.95%	44.86%
2 edit distances	0.27%	79.44%
3 edit distances	0.04%	91.87%

The output is in line with the expectation as now the algorithm is producing better recall for any level of edit distance, however, the precision is slightly worse because now we truncate some long strings into normal-length and no doubt will generate more candidates.

4.3 N - gram distance

The next approximate string matching technique we will evaluate is the n-gram distance[3]. It is an algorithm to decompose a string into several partitions and get similarity based on common partitions and distinct partitions.

It is less accurate for both very long and short string patterns as the n-grams set could get enormous hence more likely result in a larger distance. Or on the other hand lacking characters makes it impossible to match, for example, "u" to "you" as the information is limited.

Following accuracy scores are generated by tuning the n parameter and select the most similar match (highest n-gram similarity score) from the dictionary file provided:

N-gram	Accuracy
N = 1	15.51%
N = 2	31.12%
N = 3	30.09%

The accuracy of n-gram algorithm is not bad given that it is the sole selection criterion of the match. It works well on pairs of strings have lots of overlapping of substring sequences.

If we allow for multiple candidates based on n-gram similarity score and keep the top k candidates for each token we get the following:

N-gram (N=2)	Precision	Recall
Top 10 for each token	4.73%	46.07%
Top 5 for each token	9.96%	38.79%

By manually forcing the maximum number of candidates for each token, we can increase the precision quickly. The recall dropped from 46.07% to 38.79%.

[3] http://pythonhosted.org/ngram/ngram.html

4.4 Further analysis

After evaluating several algorithms, we can see that using GED alone we can achieve recall as high as over 90%, however, the precision is extremely poor as we are short of ways to rank the candidates so they are equally likely to be the final canonical form. Apparently, this is not true, because some of the words are rarely appearing in common English usage and should be removed.

Director of research at Google Peter Norvig[4] provides a simple but elegant solution to this narrowing down process. He wrote a demo spelling checker by first extracting distinct words from a file that includes book excerpts from Project Gutenberg[5] and lists of common English words from other sources to build a dictionary. Then using GED to generate candidates that are 1 and 2 edits away, furthermore, counting the frequencies of appearances of candidates in the previous file and rank candidates based on edit distances (higher priority) and the frequency.

We use our dataset to evaluate the algorithm to see whether the precision is improved or not:

	Precision	**Recall**	**Accuracy**
- GED (1 and 2 edits) - Language model: frequency - Error model: priority based on edits	10.74%	40.65%	34.02%

It indeed outperforms both N-grams and GED by improving the precision to roughly 10.74% and the recall is around 40%.

We haven't considered phonetic matching yet since it is more suitable for name matching rather than general words matching. Soundex is central to phonetic matching. It tries to group consonants and encode groups of consonants into numbers. Each word is represented by a 4-character length encoding and then we can derive the similarity score using edit distances based on the encoding rather than the original words.

Modern phonetic matching algorithms are developed primarily based on this principle of encoding and matching.

We will evaluate a famous spell checker "Aspell"[6] which is based on phonetic matching algorithm Metaphone and GED. It first selects candidates based on GED between the encodings of the pairs and then using a score calculated by a weighted average of a weighted GED over both the encodings and the words to rank the candidates.

We evaluated the performance of the spelling checker over our dataset and present the evaluation matrix as the following (precision-recall calculated by keeping top 5 word suggestions, accuracy by keeping top suggestion):

	Precision	**Recall**	**Accuracy**
MySpell[7]	7.11%	27.57%	16.26%
Aspell	9.24%	36.92%	15.42%

4.5 Reflections and Improvements

By examining various approximate string matching algorithms and the performance of various ready-to-use spelling checkers, we have derived the following knowledge and raised some points for further improvements:

- It is challenging to build a spell-checking algorithm with high precision solely based on edit distance without a mechanism of distinguishing between "common" words and "rare" words
- Levenshtein Distance Algorithm parameters can be tuned to suit for different situations, for example if there are many tokens with many consecutive duplicating characters, the penalization on deletion operation can be lowered
- Pre-processing of the data can be helpful when the data is clearly exhibiting some features, for this data set, we can easily improve the performance by manually tagging numbers as its phonetic substitutions, adding a twitter lexicon for searching, etc.
- The reference library is critical for the performance of spell-checking algorithm as we have seen in Norvig's algorithm, it uses a library with context which incorporates some valuable information about human behaviour
- Adding more algorithms to select candidates, potentially increase the robustness of the candidates by adding computation time
- Optimized language model and error model under twitter context, for example, high frequency words appearing on twitter, common twitter spelling errors and misuse of words (on purpose?), n-grams of corpus of tweets, etc.

4.6 Conclusions

In this project, we started by formulating a problem of twitter lexical normalization and then isolated the OOV tokens, examined several string matching algorithms and came up with some knowledge about the task. Now we go back to the original question: what makes a good normalization system?

[4] http://norvig.com/spell-correct.html
[5] http://www.gutenberg.org/wiki/Main_Page
[6] http://aspell.net/0.50-doc/man-html/8_How.html
[7] https://en.wikipedia.org/wiki/MySpell

- Identifying IV tokens and NO tokens:
  Dictionary, regular expression, or more complex classification algorithms with features like string length, character types, twitter appearing frequency, POS tagging etc.
- Handling phonetic substitutions:
  e.g. 4 -> four, 2 -> to
- Recovering acronyms:
  e.g. lol -> laugh out loud
  write a parser to query online slang dictionary
- A good spelling corrector:
  Context-aware (google n-gram dataset, Project Gutenberg), phonetic-aware, language model, error model, candidate model, selection mechanism, etc.

References

Bo Han and Timothy Baldwin. 2011. Lexical normalization of short text messages: Makn sens a #twitter. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011)*, pages 368-378, Portland, USA.

Bilal Ahmed. 2015. *Lexical Normalisation of Twitter Data.* Science and Information Conference, July 28-30, 2015, London, UK

Dmitry Supranovich and Viachaslau Patsepnia. 2015. IHS_RD: Lexical Normalization for English Tweets. In *Proceedings of the ACL 2015 Workshop on Noisy User-generated Text*, pages 78–81, Beijing, China, July 31, 2015.

Vangie Beal. Text messaging and online chat abbreviations. Web. 01 Apr. 2015. http://www.webopedia.com/quick_ref/textmessage abbreviations.asp

Fei Liu, Fuliang Weng, Bingqing Wang, Yang Liu. Insertion, Deletion, or Substitution? Normalizing Text Messages without Pre-categorization nor Supervision. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011)*, short paper, pages 71-76.

Fei Liu, Fuliang Weng, Xiao Jiang. A Broad-Coverage Normalization System for Social Media Language. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL 2012)*, pages 1035-1044.

Bird, Steven, Edward Loper and Ewan Klein (2009). Natural Language Processing with Python. O'Reilly Media Inc.

Hiroyuki Tanaka. 2013. editdistance. https://github.com/aflc/editdistance.

Graham Poulter. 2012. abydos. http://github.com/gpoulter/python-ngram.

Norvig, P. 2007. How to Write a Spelling Corrector. http://norvig.com/spell-correct.html.

K. Gimpel, N. Schneider, B. O'Connor, D. Das, D. Mills, J. Eisenstein, M. Heilman, D. Yogatama, J. Flanigan, and N. A. Smith. 2011. Part-of-speech tagging for Twitter: Annotation, features, and experiments. In *Proc. of ACL*.