Lexical Normalisation of Twitter Data

Tiancong Li

1 Introduction

This report provides a method to normalize the Twitter data. More specifically, by giving an unlabelled token, we expect the system to produce a normalized output, such like from `tomoroe` to `tomorrow`.

There are many researches about normalization of media data, such as the one described in (Han et al., 2013). They focused on the OOV data and firstly used morphophonemic similarity to produce the candidates, then selected the best match according to word similarity and context.

The data set used here contains labelled-tokens.txt, labelled-tokens.txt, and dict.txt, which respectively represents the labelled, unlabelled data, and dictionary. Our system will use the labelled data to evaluate itself, and the final target is to give a canonical form of unlabelled data.

Since different similarity methods focus on the different aspects, we would like to see what kind of effects we could have. For Instance, edit distance focuses on how many changes needed to be made, and ngram cares about the subests' intersections between words, while soudex calculates the phonemic similarity of words. In this report, the algorithms mentioned above will be discussed and used.

2 Method

To nomalize a token, firstly we will check whether it is already in the dictionary. If it is, we assume the token is already in the right form. Secondly, for the OOV(out of vocabulary) tokens, the normalization will be implemented.

2.1 Preprocessing

As mentioned above, before performing the normalization functions, some tokens are already correctly written and do not need to be normalized. Several forms are shown belown.

1. The token already in the dictionary, marked as IV(in vocabulary).
2. The token starts with `"@"`.
3. The token starts with `"#"`.
4. The token starts with `"http"`.
5. The token contains `"__"`.

If a token satisfies any form shown above, the ouput will be the token itself.

2.2 Normalization

The tokens without specific forms will be normalized. We use the data in labelled-tokens.txt to test the performance of different normalization methods.

Since the dictionary is very large, the time of the normalization could be extremely long. So, we only use the words in the dictionary with the same first letter as the token. By doing that, we assume that the miss spell does not occur at the first letter of the word.(Christopher D. Manning, 2009)

2.2.1 Single method approach

To apply a lexical normalization, the most straightforward idea is to use one of the *similarity methods*, and get the output with the highest similarity.

Before normalization, we will firstly preprocess it to filter the tokens with specific forms mentioned in `2.1`. The output is shown below.

Method	precision	recall
Levenshtein Distance	3.76%	27.51%
Local Edit Distance	0.13%	22.00%
Soundex	0.30%	31.59%
2gram	15.98%	23.19%

As we can see, the performance of the single method is poor.

2.2.2 Combinational methods approach

So, to improve the performance of our system, we try to use a combination of normalization methods rather than a single one. More specifically, we use a combination of *Levenshtein Distance*(Tanaka, 2016) and 2*gram Distance*(Poulter, 2017), and the token will firstly be processed by *Levenshtein Distance*, then followed by 2*gram Distance*.

Instead of selecting the output with the highest similarity, we set thresholds to both methods. Firstly, the threshold of *Levenshtein Distance* is set to 4, and we will test the best threshold for 2*gram Distance* (the *Python* package of 2*gram* uses $1/distance$ rather than *distance* as the threshold). The output is shown below(the first two columns represent thresholds).

Leven	2gram	precision	recall
4	0.1	0.11%	39.86%
4	0.2	0.42%	37.88%
4	0.5	37.20%	47.44%
4	0.7	41.72%	41.72%

At this point, we will focus more on the recall rather than precision, since we could increase the precision later to the almost same level of recall.

So, we can find that 2*gram Distance* with threshold 0.5 gives the best recall. At the next Step, we set the threshold of 2*gram Distance* as 0.5, and test the various thresholds of *Levenshtein Distance*. The output is shown below.

Leven	2gram	precision	recall	time(s)
1	0.5	44.49%	47.09%	25
2	0.5	39.90%	46.50%	28
5	0.5	36.99%	47.55%	69
7	0.5	36.88%	47.67%	143
10	0.5	36.88%	47.67%	247

As we can see, as the threshold of *Levenshtein Distance* grows, the performance does not change a lot, but the time cost is growing fast. At this moment, the combination of *Levenshtein Distance* with threshold 7 and 2*gram Distance* with threshold 0.5 seems to have the best performance (We use 7|0.5 to represent the combination).

We find some bad cases when looking into details. For example, the token is "lyke" and the correct form is "like", where our system fails to predict a result. Obviously, this could be solved by *Soundex*(YouGov, 2015). So, we further add *Soundex* to the existed combination, by which I mean the token will go through *Levenshtein Distance* and then the same token will go through the *Soundex*. After that, the union set of the two outputs will go through 2*gram Distance* with threshold 0.5.

The output is shown below after *Soundex* is added.

Leven	2gram	precision	recall	time(s)
1	0.5	45.27%	50.70%	30
2	0.5	40.92%	48.60%	32
5	0.5	36.99%	47.55%	74
7	0.5	36.79%	47.55%	155

As we can see, after *Soundex* added, the performance of combination 7|0.5 almost remains the same(even a little worse), but the figure of 1|0.5 increases to 50.70%, and this combination is also less time consuming. So, we prefer to use the 1|0.5 with *Soundex* added, which gives us a good performance and less runtime.

2.3 Improvements

At this moment, we get our best solution which is the combination of 1|0.5 with *Soundex* added. However, some improvements can be made.

2.3.1 Specific cases

Looking into the details of the output, some bad cases are interesting.

token	correct word
freakin	freaking
sonqs	songs
s0ngg	song
alllllllll	all
niggaz	niggas
wal-mart	walmart
1	one
2day	today
4get	forget
dese	these

By observing these bad cases, some changes need to be made to the token before the normalization.

1. Change 0 to `o`.
2. Change 2 to `to`.
3. Change 4 to `for`.

4. Delete - in the token.

5. If token has repeated letters, then make them no more than two.

And after the normalization, if the system does not give us a result, we could change the token and normalize it again. Possible changes are shown below.

1. If token ends with `in`, then add `g` to the end.

2. Subtitude `q` with `g`.

3. If token ends with `z`, then substitude it with `s`.

4. If token starts with `de`, then substitude it with `the`.

After these specific operations, the output of the combination of 1|0.5 with *Soundex* is shown below.

Leven	2gram	precision	recall	time(s)
1	0.5	47.22%	55.48%	56

As we can see, the improvement is obvious while the time consuming is acceptable.

2.3.2 Increase the Precision

At this moment, we need to improve the precision of the system, by which I mean reducing the candidates of current output.

To achieve the goal, we choose *ngram* method, and a result is shown below.

threshold	precision	recall
0.6	54.73%	55.24%
0.7	54.90%	54.90%
0.8	54.31%	54.31%

As we can see, the threshold 0.7 gives the best performance.

2.3.3 Extra Dictionary

The point of this section is straightforward. Because the labelled data is given, we could use it to construct a key-value dictionary, and use this dictionary when preprocessing the unlabelled data. So one more condition will be add to preprocessing, which is:

if token is one of the key in the key-value dictionary, return the corresponding value as the normalized word.

3 Conclusion

At the end, the system is contructed.

Firstly, the unlabelled data will be preprocessed if it satisfies the conditions shown in `2.1` or `2.3.3`. Secondly, the data will be operated to handle specific cases described in `2.3.1`, and go through the normalization, which is the combination of 1|0.5 with *Soundex*. Finally, the data will go through *ngram* with threshod 0.7 to get better precision.

At the end, we get the output with the accuracy of 54.90%. After trying this system on `Kaggle`, the accuracy of 0.88343 is given.

It is clear to see that instead of using single normalization method, we use a combination of strategies, and by this way, we could consider more aspects and make a better performance. As for the threshold adjusting, the simple idea is that we want to leave some weights to all the normalization methods by setting the threshold of each one appropriately.

References

Hinrich Schtze Christopher D. Manning, Prabhakar Raghavan. 2009. *Introduction to Information Retrieval*. Cambridge University Press.

Bo Han, Paul Cook, and Timothy Baldwin. 2013. Lexical normalization for social media text. *ACM Trans. Intell. Syst. Technol.*, 4(1):5:1–5:27, February.

Graham Poulter. 2017. ngram 3.3.2. `http://github.com/gpoulter/python-ngram`.

Hiroyuki Tanaka. 2016. editdistance 0.3.1. `https://www.github.com/aflc/editdistance`.

Plc. YouGov. 2015. Fuzzy 1.1. `https://bitbucket.org/yougov/fuzzy`.