Application Project 1
Preprocessing toolkit for English

Dinno Koluh
Boğaziçi University
Department of Computer Engineering
Natural Language Processing
`dinno.koluh@studio.unibo.it`

Abstract

This paper describes a system for the preprocessing of the English language. Namely, the described toolkit consists of a rule-based and machine-learning-based tokenizer and sentence splitter respectively, normalizer, stemmer, a system for stopword elimination, vocabulary generation and word frequency generation. These systems take into account ambiguous punctuations, URLs, hashtags, e-mails, MWEs, clitics expansion and tokens with numbers. Satisfactory results are reported for a variety of test sets and are available with the source code here.

1 Introduction

Tokenization is an essential part of the NLP pipeline because most of the other systems and actually the whole pipeline builds upon the fact that tokenization is done correctly, efficiently and robustly. We are going to use also this fact to build the sentence splitter i.e. rebuilding sentences from tokenized entities. Tokenization would be trivial if all independent words or expression would be split by a whitespace but it is not easy as that. We have MWEs that even though they are words separated by whitespaces they act as a single entity (e.g. New York). Then we have also abbreviations which have periods inside of them but are not the ending of a sentence (e.g. e.g.). Then we have other special entities like URLs, hashtags, e-mails etc. We are going to show how these problems were tackled in the next parts of the paper.

Assuming we have a good tokenized set of words we can now perform other tasks in the toolkit. We should point out that we will have different types of tokens based on the area of use (e.g. *rough tokens* will be used in sentence splitting, but *clean tokens* will be used for stemming). As mentioned, tokens will be used for sentence splitting by rebuilding sentences and splitting them depending on a fix set of rules. We will also perform tokenization and sentence splitting with the use of machine-learning techniques such as Logistic regression (for sentence splitting) and Naive Bayes classification (for tokenization).

The stemming will be performed using the Porter stemmer [1]. The normalization procedure will be the aggregation of stemming, case-lowering, clitics expansion, lexicon usage etc. All of these procedures give rise to a normalized set of tokens which can be used to build a vocabulary and word-frequency dictionary. It should be pointed out that all the lexicons used in the project are minimalistic, and only serve a testing purpose. For a real life application the lexicons should be expanded.

Let us now look at the most crucial part of the system, the tokenization.

2 Tokenizer

2.1 Rule-based

Let us start of by giving a concrete example. We want to tokenize the following text:

"You can't find Mr. Nobody N. in New York City. Since 22/11/1996, the only way is at mr_nobody@gmail.com (or at his website: www.mr_nobody.com)."

We will have different types of tokens depending on the current position in the development phase of getting the final tokenization of an input text. The first type are ***rough tokens***. We get rough tokens by splitting the input text just at whitespaces and newlines. Such tokens will be essential for sentence splitting. For the given example the rough tokenization gives:

['You', ”can't”, 'find', 'Mr.', 'Nobody', 'N.', 'in', 'New', 'York', 'City.', 'Since', '22/11/1996,', 'the', 'only', 'way', 'is', 'at', 'mr_nobody@gmail.com', '(or', 'at', 'his', 'website:', 'www.mr_nobody.com).']

This type of token has not lost information about the input text which is very important, but the tokens contain undesired characters which need to be removed.

The next type of tokens are ***dirty tokens***. These tokens take as an input the rough tokens from the previous step, and strip them from punctuation signs but keeping the punctuations as valid tokens (so not fully removing them). The procedure is not as simple as just striping all punctuations from tokens. We have many cases where the punctuation sign should not be removed (e.g. apostrophe in ”can't”, period in ”mr_nobody@gmail.com” or ”Mr.”, forward slash in ”22/11/1996”) but also other cases where it should be removed (e.g. period in ”City.”, comma in ”22/11/1996,”, parenthesis in ”(or”). So with a fixed set of rules all of those cases can be addressed separately. The resulting dirty tokenization gives us:

['You', ”can't”, 'find', 'Mr.', 'Nobody', 'N.', 'in', 'New', 'York', 'City', '.', 'Since', '22/11/1996', ',', 'the', 'only', 'way', 'is', 'at', 'mr_nobody@gmail.com', '(', 'or', 'at', 'his', 'website', ':', 'www.mr_nobody.com', '.)']

Let us mention some important points in the rule-based approach. We need an abbreviation lexicon for words like ”Mr.” as they should keep the punctuation sign and this fact will be important in sentence splitting. We should also keep in mind that name abbreviations start with an upper-case letter like ”N.” so that should also be used as a rule.

Concerning emails, URLs or other entities that contain a special character or a special substring a recursive approach is used to keep/separate such tokens. Assume that we want to split the token ”,no_address@enron.com:”. In this case, the special character is ”@” and it is used as the starting point for further splitting. So, the initial token is split into subtokens, the one before the occurrence of the special character and the one after the occurrence of the special character ([”,no_address”, ”enron.com:”]). Then, those two tokens are fed back into the splitting algorithm and are split further ([”,”, ”no_address”], [”enron.com”, ”:”]). After getting the final list of subtokens we take the last element from the first list and the first element from the second list and concatenate them back with the special character inside ([”,”, ”no_address@enron.com”, ”:”]). This can be done for any desired character (e.g. for “'“ in the “'don't'“ will be split as ([“'“, “don't“, “'“]).

The same idea holds for strings contained inside tokens. It is useful for abbreviations contained in ambiguous context inside a token (e.g. ”Mr.,” will be split as [”Mr.”, ”,”] using ”Mr.” as the special substring).

The next type of tokens are ***clean tokens***. They take the dirty tokens as input and remove all tokens that don't begin with a letter or number. These tokens will be the ones that are used for vocabulary building. One may ask, what is then the purpose of having the dirty tokens, why keep the punctuations as separate tokens? The frequency of punctuations can give us some insight in the structure of the input text, and just having the sequence of such tokens we still retain information of the input text, but when we remove the punctuations, we will inevitably lose information. Dirty tokens will be also essential for the classification procedure in the Naive Bayes model.

As far as regarding MWEs, we keep a lexicon of such words and then introduce a rebuilding step for rebuilding MWEs depending on a combination of two neighbouring tokens, and then building upon them multi-word MWEs (e.g. New York City). Before this step all tokens are set to lower-case and all clitics are expanded using a clitics dictionary. Keeping in mind these steps, the clean tokens from the initial examples are given as:

['you', 'cannot', 'find', 'mr.', 'nobody', 'n.', 'in', 'new york city', 'since', '22/11/1996', 'the', 'only', 'way', 'is', 'at', 'mr_nobody@gmail.com', 'or', 'at', 'his', 'website', 'www.mr_nobody.com']

This is the end step, so the clean tokens are the ones that should be used in further NLP applications. We will also introduce two more types later which will be useful for stemming and stop-word removal, but the clean tokens are the bench mark ones.

2.2 With Naive Bayes

We will demonstrate how to build a binary Naive Bayes classifier. We classify each character in the input as positive if it will be used as a split between tokens, or negative if the character is just a part of the current token. To be able to classify a character we need to provide features which will discriminate between the two classes. The features that were used are:

1. Is the character a letter?
2. Is the character a number?
3. Is the character a whitespace?
4. Is the character between two alphanumeric values?
5. Is the character a punctuation sign?
6. Is the character next to a number?
7. Is the character an ’@’ sign?
8. Is the character a punctuation sign between two alphanumeric values?

So we will obtain a binary feature vector for each character and they will be used for training. The problem now is how to automatically classify each character so that we can obtain the actual target (class) of a specific character. The answer to this problem is to use the already constructed rule-based tokenizer. The idea is the following.

We can construct a sliding window of length $2d$ such that the center is at the position i of the text which is the current character to be classified. We take a sample from the text in the range $[i-d, i+d]$ and feed it in the rule-based tokenizer. Then we just check if the character was tokenized as a single dirty token. If so, then the character should be used as a splitter.

One downside to this approach is the challenge of constructing a suitable data-set for feature extraction, so that there is no overfitting as the most frequent occurring features will be the ones that a character is alphanumeric or a whitespace. After constructing a suitable dataset, it is fed into the Naive Bayes classifier, and the classifier is trained. For a test set we extract the features of each character in the input test text. These features are then fed into the classifier so that it predicts the classes with some probability. We will show the first 10 classified characters of the previous example and the tokenization of the whole sentence.

Y [1, 0, 0, 0, 0, 0, 0, 0] $\rightarrow$ [0] with $p =$ [0.98046019 0.01953981]
o [1, 0, 0, 1, 0, 0, 0, 0] $\rightarrow$ [0] with $p =$ [0.98212894 0.01787106]
u [1, 0, 0, 0, 0, 0, 0, 0] $\rightarrow$ [0] with $p =$ [0.98046019 0.01953981]
[0, 0, 1, 1, 0, 0, 0, 0] $\rightarrow$ [1] with $p =$ [0.02317899 0.97682101]
c [1, 0, 0, 0, 0, 0, 0, 0] $\rightarrow$ [0] with $p =$ [0.98046019 0.01953981]
a [1, 0, 0, 1, 0, 0, 0, 0] $\rightarrow$ [0] with $p =$ [0.98212894 0.01787106]
n [1, 0, 0, 0, 0, 0, 0, 0] $\rightarrow$ [0] with $p =$ [0.98046019 0.01953981]
’ [0, 0, 0, 1, 1, 0, 0, 1] $\rightarrow$ [0] with $p =$ [0.57554159 0.42445841]
t [1, 0, 0, 0, 0, 0, 0, 0] $\rightarrow$ [0] with $p =$ [0.98046019 0.01953981]
[0, 0, 1, 1, 0, 0, 0, 0] $\rightarrow$ [1] with $p =$ [0.02317899 0.97682101]

[’You’, ”can’t”, ’find’, ’Mr’, ’.’, ’Nobody’, ’N’, ’.’, ’in’, ’New’, ’York’, ’City’, ’.’, ’Since’, ’22/11/1996’, ’,’, ’the’, ’only’, ’way’, ’is’, ’at’, ’mr_nobody@gmail.com’, ’(’, ’or’, ’at’, ’his’, ’website’, ’:’, ’www.mr_nobody.com’, ’)’, ’.’]

Other features (e.g. for MWEs) could be added to get a more accurate classifier, but the feature creation function was made such that it is easily scalable up to a desired number of features.

3 Sentence splitter

3.1 Rule-based

Let us again start by giving a concrete example. We want to sentence-split the following text:

"You can't find Mr. Nobody N. in New York City. Since 22/11/1996, the only way is at mr_nobody@gmail.com (or at his website: www.mr_nobody.com). Invalid of the form user@enron.com., whenever possible (i.e., recipient is specified in some parse-able format like "Doe, John" or "Mary K. Smith" at http://www.stanford.edu). But 10.8$ is enough for us! I drove 50 m.p.h. and got a speeding ticket? Not possible!!"

As already mentioned we will use the rough tokens to rebuild the sentences from them, as they still keep all the information of the input text so we just need to build a careful set of rules if an element from the set S = {".", "!", "?"} should be a sentence splitter or not. So let us look at the rough tokens of the input text:

['You', "can't", 'find', 'Mr.', 'Nobody', 'N.', 'in', 'New', 'York', 'City.', 'Since', '22/11/1996,', 'the', 'only', 'way', 'is', 'at', 'mr_nobody@gmail.com', '(or', 'at', 'his', 'website:', 'www.mr_nobody.com).', 'Invalid', 'of', 'the', 'form', 'user@enron.com.,', 'whenever', 'possible', '(i.e.,', 'recipient', 'is', 'specified', 'in', 'some', 'parse-able', 'format', 'like', '"Doe,', 'John"', 'or', '"Mary', 'K.', 'Smith"', 'at', 'http://www.stanford.edu).', 'But', '10.8$', 'is', 'enough', 'for', 'us!', 'I', 'drove', '50', 'm.p.h.', 'and', 'got', 'a', 'speeding', 'ticket?', 'Not', 'possible!!']

It is trivial to build the rule that if the ending character of a token belongs to the set S that it should be a sentence splitter. But there are many exceptions for which rules are constructed. Some of them are

- If a token is inside the abbreviations lexicon it should be not a sentence splitter (e.g. Mr.).
- If the last character of a token is a period and the character before the period is an upper case letter it should not be a sentence splitter (e.g. K.) as it is most likely an abbreviation of a name.
- If a token has a number and some characters that may be a period it should not be a sentence splitter (e.g. 10.8$)
- If a token contains an email it should not be a sentence splitter (e.g. user@enron.com)

Most certainly are other rules which could make the sentence splitter more accurate but these ones are the basic ones and give satisfactory results. The output of the sentence splitter for these rules is:

["You can't find Mr. Nobody N. in New York City."]
['Since 22/11/1996, the only way is at mr_nobody@gmail.com (or at his website: www.mr_nobody.com).']
['Invalid of the form user@enron.com., whenever possible (i.e., recipient is specified in some parse-able format like "Doe, John" or "Mary K. Smith" at http://www.stanford.edu).']
['But 10.8$ is enough for us!']
['I drove 50 m.p.h. and got a speeding ticket?']
['Not possible!!']

3.2 With logistic regression

For a sentence splitter using logistical regression, a binary classifier was built which discriminates each occurrence of an element from the set S as a sentence splitter (true) or not (false). The only preprocessing used is to remove newlines and apart from that the classifier works on raw text. To develop such a mechanism, handmade features were made that depend on the environment of an element from the set S. Such features are similar to the rules we mentioned before but we also need the cases for which the splitter should actually split the sentence. We constructed a simple classifier which uses eight features:

1. Is punctuation a period?

2. Is previous character lower-case?
3. Is previous character upper-case?
4. Is previous character a number?
5. Is next character a letter?
6. Is next character a number?
7. Is next character a whitespace?
8. Is previous token an abbreviation?

In this way we construct a binary vector. Let us look at an example for the first occurrence of a punctuation in the input text (Mr.). The binary vector would be:

$$v_0 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

. The next task is to assign to each vector a class. This could be done manually but it would not be scalable. So, the previously constructed rule-based approach was used as being the ground truth so in this way we could get the classes of each punctuation in an automatic manner.

Using sample texts a dataset was made that was used to train the classifier. Then new test text could be fed into the classifier where each punctuation is extracted, we get the features and feed them into the classifier and we get the probability of that punctuation actually being a sentence splitter. For the input text we get the same result as with the rule-based sentence splitter. Below are shown the probabilities of each punctuation being a splitter and the decision of the classifier (0 or 1).

$$p = \begin{bmatrix} 0.7063189 & 0.2936811 \\ 0.6448781 & 0.3551219 \\ 0.29224788 & 0.70775212 \\ 0.93637777 & 0.06362223 \\ 0.93637777 & 0.06362223 \\ 0.93637777 & 0.06362223 \\ 0.46025011 & 0.53974989 \\ 0.93637777 & 0.06362223 \\ 0.82581031 & 0.17418969 \\ 0.93637777 & 0.06362223 \\ 0.82581031 & 0.17418969 \\ 0.6448781 & 0.3551219 \\ 0.93637777 & 0.06362223 \\ 0.93637777 & 0.06362223 \\ 0.46025011 & 0.53974989 \\ 0.92868398 & 0.07131602 \\ 0.20121053 & 0.79878947 \\ 0.93637777 & 0.06362223 \\ 0.93637777 & 0.06362223 \\ 0.7063189 & 0.2936811 \\ 0.20121053 & 0.79878947 \\ 0.7430661 & 0.2569339 \\ 0.85657405 & 0.14342595 \end{bmatrix} \text{decision} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

4 Stemming

The implemented stemming algorithm was the widely used Porter stemmer. It has a fixed number of rules which need to be implemented in successive steps so that the stem can be obtained. This type of tokens are called ***stemmed tokens***. Clean tokens are the input for the stemmer. From the example used on other token types the stemmed tokens are:

['you', 'cannot', 'find', 'mr.', 'nobodi', 'n.', 'in', 'new york c', 'sinc', '22/11/1996', 'the', 'onli', 'wai', 'is', 'at', 'mr_nobody@gmail.com', 'or', 'at', 'hi', 'websit', 'www.mr_nobody.com']

5 Stopword Elimination

Stopword elimination can be carried out trivially by having a predefined lexicon of common stopwords and then the tokens can be pruned using such a lexicon, in fact the tokens obtained after this step will called ***pruned tokens***. And using our example, these are the obtained tokens:

['cannot', 'find', 'mr.', 'nobody', 'n.', 'new york city', 'since', '22/11/1996', 'way', 'mr_nobody@gmail.com', 'website', 'www.mr_nobody.com']

The other way is dynamically removing the most commonly occurring tokens across several documents in a corpus. The frequency of the most commonly occurring tokens can be obtained from the word frequency dictionary which was also trivially built inside the toolkit. The idea would be to put a threshold as just a number, or rather a percentage of the occurring words (e.g. first 2% of words after sorting by frequency should be removed). To have a representative example, a modestly sized corpus should be used with different documents.

6 Normalization

The normalization step is defined as putting words with multiple forms into a single, normalized one. We have already done some normalization in the steps above. These include:

- Case-lowering for all tokens (e.g. "You", "you", "YOU" $\rightarrow$ "you")
- Clitics expansion (e.g. "You'll", "you will" $\rightarrow$ "you will")
- Stemming (e.g. "Network", "networking" $\rightarrow$ "network")

For other forms (e.g. "USA", "U.S.", "U.S.A", "United States" $\rightarrow$ "usa") a normalization lexicon needs to be used. That can be trivially implemented. The output of the normalization step can be used to construct the vocabulary and word frequency dictionary of the input text.

7 Discussion and Conclusion

In this paper a step-by-step NLP preprocessing toolkit was proposed for the main NLP tasks. The basis of the toolkit was tokenization. It was carried out in steps by constructing different types of tokens which have a different usage in the specific NLP system. Alongside the traditional rule-based system a machine-learning system was also developed which gave satisfactory results. It was build in such a manner that it can be easily scaled up by just adding new suitable features. Still the biggest challenge for this type of system is to find a suitable training set.

The task of building this toolkit was successfully completed with promising results for future development.

References

[1] M.F. Porter, "An algorithm for suffix stripping, Program", Vol.14(3) p.130-137, 1980.
URL: `https://tartarus.org/martin/PorterStemmer/`