Language Independent Tokenizer using NeuraBASE

Christopher Seidl, Neuramatix

March 6, 2020

Abstract

Tokenization is the process of splitting a running text into semantically meaningful sections (tokens). It is a prerequisite for effective text processing as it helps to expose meaningful patterns in large textual data-sets.

In most systems, tokenization is achieved through recursively applying a set of cascading language dependent rules to the text based corpus. These rules are written by linguists using a deep knowledge of the idiosyncrasies of the language with which they are working. Due to their specificity, these rules are extremely brittle: for instance rules which can tokenize English text will need to be rewritten if they are to work with French text.

This paper proposes a different method of tokenization. Rather than injecting human knowledge of language into a system, we use a self-learning neural network to observe the patterns of a particular language, gradually learning how to deconstruct it into tokens. The system is extremely flexible because it starts with no knowledge of the language so it could in theory be applied to any language or code.

1 Introduction

Tokenization at its best mirrors the phenomenon of chunking in cognitive psychology [1]. Chunking is a process where information is broken down and reconstructed as higher order abstractions which are more easily remembered and manipulated. Through repeated exposure to chess positions, for instance, a grand-master stops seeing individual pieces (a pawn, a rook, etc.) and instead sees deeper structural patterns such as a castled kingside or fianchettoed bishop.

In the context of language, letters (or uni-code characters) can be "chunked" into syllables, words and even multi-word phrases. As the human mind becomes more familiar with language it starts to read word-by-word rather than letter-by-letter.

Furthermore, we are never taught to build these higher order representations, it happens automatically after repeated exposure to linguistic patterns. Through using a self learning network for tokenization we aspire to do the same [2].

1.1 Example of good tokenization:

```
[THERE IS ][SO MUCH ][TO UNDERSTAND][.]
```

Above is an example of the kind of tokenization we want to achieve. Despite our network being a zero knowledge system with no understanding of the way white-space delimits words in English, it has not split any of the words across tokens. It has also correctly tokenized across white-space boundaries.

Furthermore, each token has meaning as a standalone linguistic element. For example the token corresponding to `[THERE IS ]` has meaning independent of the other words in the sentence as do all the other tokens. This tokenization is desirable because it has tokenized the sentence exactly the same way our minds would have "chunked" it.

1.2 Example of poor tokenization:

```
[SENTENCE][S COME ][IN MANY ][SHAPE]
[S AND SIZ][ES][.]
```

Whereas the previous example split the sentence into standalone tokens with their own distinct and independent meanings, this example has created many tokens which depend on each other for meaning. Consider `[S COME ]` and `[S AND SIZ]`. Whereas in the previous example meaning could be ascribed to each token, these two tokens mean nothing unless their wider context in the sentence is considered.

This is undesirable: it means rather than building a neural network from self-contained linguistic blocks, you are combining elements which mean nothing on their own and just happen to appear next to each other. This is likely to lead to overtraining.

1.3 Quantifying tokenization

As a general principle, each token should have its own distinct meaning and should not depend on context to be understood. If a token does not comply with this principle, it will be labelled as invalid. Otherwise, if it has an independent meaning it will be labelled as valid.

Results will be quantified by considering the fraction of valid tokens over total tokens in an input. The example of good tokenization has a score of 100% whereas the poor tokenization example has a score of 57% (only `[SENTENCE]`, `[IN MANY ]`, `[SHAPE]` and `[.]` are valid).

2 Different Approaches

2.1 Overview of System

The network in this paper was a zero knowledge system which had no understanding of language and no embedded linguistic rules[1]. It took successive text inputs (taken from Wikipedia) and processed them to develop an understanding of English. The network had following three components:

2.1.1 Tokenization

Firstly, a line of text was taken from the corpus and was decomposed into a series of tokens. Each token was found somewhere in the network. The network has a library of tokens indexed in an inverted tree structure, to find each token a recursive search of the data structure was implemented.

On the first iteration, the network was empty so the string would be decomposed into a token for each character. This series of tokens was then sent to the joining stage where longer tokens would be formed and added to the library.

```
IN:  "THE BLACK CAT"
OUT: [TH][E][ BLA][CK ][CAT]
```

2.1.2 Joining

In order to actually learn, new tokens had to be formed using the knowledge gained from the previous tokenization stage (similar to the way neurogenesis consolidates new memories in the brain). To achieve this, a new token was formed from each pair of successive tokens.

```
IN:  [TH][E][ BLA][CK ][CAT]
OUT: [THE],[E BLA],[ BLACK ],[CK CAT]
```

2.1.3 Feedback

Finally, to reinforce the learning which took place, all the tokens along the paths which were activated in the inverted tree structure had their activation count incremented (to simulate the reinforcing effects of frequently firing neurons in the

[1] This is in strong contrast to proposals such as *Cutter – a Universal Multilingual Tokenizer*, Graën, which rely on hard coded linguistic rules

brain). These activation counts were used later to calculate entropy.

2.2 Greedy Algorithm

The first tokenization algorithm sequentially processed the input string character by character, searching through the network to find the longest tokens possible.

```
IN:  "I WENT TO THE SHOPS TO BUY"

1 => [I_WENT_]
2 => [I WENT ][TO_THE]
3 => [I WENT ][TO THE][_SHOP]
4 => [I WENT ][TO THE][ SHOP][S_TO_BU]
5 => [I WENT ][TO THE][ SHOP][S TO BU][Y_]

OUT: [I WENT ][TO THE][ SHOP][S TO BU][Y ]
```

The algorithm began well by generating valid tokens (such as `[I WENT ]` and `[TO THE]`) however issues arose when it generated `[S TO BU]`. This is sub-optimal: instead of generating a single invalid token, a better algorithm would have generated three valid tokens `[S ][TO ][BUY ]`. Importantly these options were present within the network but were not activated because the algorithm prioritised tokens with the longest length.

It was clear from this example that tokenization does not have optimal substructure and therefore a more intelligent approach was required.

2.3 Decrease and Conquer

The next approach was to consider how the human brain breaks blocks of text into meaningful chunks of information. Given a sentence, our attention is caught by what has the highest entropy.

This inspired me to redesign my tokenization algorithm to begin tokenization by splitting the input into character sized tokens and then to combine the token with the highest entropy with it highest entropy neighbour. This process was continued recursively until there were no new tokens which could be formed.

2.3.1 Calculating Entropy of Tokens

Let all the tokens in a particular input be given by:

$$T = \{t_0, t_1, t_2, t_3 ... t_m\}$$

Where $A(t_n)$ returns the activation count of t_n, the corresponding activation counts for each of the tokens are given by:

$$C = \{c_n \mid c_n = A(t_n), 0 \leq n \leq m\}$$

The entropy [3] of a particular token t_n is given by:

$$H(t_n) = -\left(\frac{c_n}{C_{max}}\right) \cdot \log\left(\frac{c_n}{C_{max}}\right)$$

2.3.2 Results

```
IN: "THE CAT SAT."

0 => [T][H][E][ ][C][A][T][ ][S][A][T][.]
1 => [T][H][E][ ][CA][T][ ][S][A][T][.]
2 => [T][H][E][ ][CAT][ ][S][A][T][.]
3 => [T][H][E][ ][CAT_][S][A][T][.]
4 => [TH][E][ ][CAT ][S][A][T][.]
5 => [THE][ ][CAT ][S][A][T][.]
6 => [THE_][CAT ][S][A][T][.]
7 => [THE ][CAT ][SA][T][.]

OUT: [THE ][CAT ][SA][T][.]
```

This algorithm started from the tokens with the highest entropy and proceeded outwards until the longest tokens possible had been created. It appeared to deliver much better results than the greedy algorithm although it necessarily had a less optimal time complexity.

What I found particularly promising about this algorithm was the way in which it frequently

broke up unknown words into syllables. For instance it processes "ELEGANT" as [E][LE][GANT] and "ACCIDENT" as [A][CCI][DENT] in a similar way to how a child learning English might break an unknown word into its constituent syllables.

Another good feature of this algorithm is its ability to correct itself as it absorbs more data. It might occasionally make a sub-optimal tokenization, but it will correct itself later on.

The greedy algorithm produced poorer quality results as the training set was increased. However, the divide and conquer approach continued to give improved results as it was trained on more data.

2.3.3 Selective Join

The divide and conquer approach had many strengths, it was particularly good at finding the most appropriate tokens in a network because of the way it used entropy. Its greatest weakness however was its tendency to create tokens bridging multiple words in a way that obscured the meaning of the text. For instance consider the following output:

```
[WEATHER IS ][NOT ][TR][IVIAL][ ][- ]
[IT'S E][SPECIAL][LY I][MPORTANT]
[ WHEN ][YOU]['RE ][STANDI][NG IN ][IT.]
```

The tokenizer has formed [LY I] when it should never have split up the word "IMPORTANT". Despite the fact that [LY] and [I] occur frequently next to each other in the corpus, the system should never have formed a token for [LY I]. To ensure the network only joined tokens which belonged together and not tokens which happened to appear next to each other, a selective join of neurons only with sufficiently high entropy was implemented.

Where H_{ave} was the average entropy for a section of text. It was only pairs of tokens t_i and t_j which satisfied the following requirement that were allowed to join:

$$[H(t_i) > H_{ave}] \wedge [H(t_j) > H_{ave}]$$

This yielded results like the following:

```
[WE][ATH][ER][ ][IS][ ][NOT][ ]
[T][RIV][IA][L][ ][-][ ][IT]['][S][ ]
[ESP][EC][IA][LL][Y][ ][IM][P][OR][TANT][ ]
[WH][EN][ ][Y][OU]['][RE][ ][ST][ANDING][ ]
[IN][ ][IT.]
```

It worked extremely well as a sub-word tokenizer because it was able to easily break words into interchangeable components which typically had different frequencies within the corpus.

A key finding was the way in which an entropy based selective join excluded white-space. Due to the way entropy was calculated in the system, white-space typically had an entropy of zero. This was also problematic because some desirable tokens such as [IN THE] contain white-space and consequently would never be formed.

2.4 Greedy Algorithm with Rollback

The divide and conquer algorithm provided higher quality results at the cost of performance. This limited the body of data which the system could be trained on. As a result an alternative approach was developed which combined the strengths of the greedy and divide and conquer approaches.

The greedy approach was computationally efficient because it started reading the text from left to right rather than trying to simultaneously read and form new tokens from all places in the input at once. Its downfall however was its tendency to overreach and form sub-optimal tokens.

The strength of the divide and conquer approach was its ability to use entropy to find the most appropriate tokens in a text.

Combining the best of each approach led to a greedy algorithm which would start at the beginning of the text and successively try to tokenize longer and longer strings of text. At each iteration it would calculate the entropy of the potential token and then at the end, select the token with the highest entropy.

It produced results of a similar quality as the divide and conquer approach in far less computationally expensive way. As a result I was able to train on a data-set several orders of magnitude larger than what my divide and conquer approach allowed.

The final output yielded results like the following:

```
[THE][ ][MY][STER][IO][US][ ]
[DIA][RY][ ][RECO][RD][S][ ]
[THE][ ][VO][ICE]
```

3 Conclusion

The most challenging thing about extracting meaning from text in a purely unsupervised fashion is that there are many instances of semantically unrelated tokens which occur next to each other at high frequencies. There are also instances of tokens which should be joined together despite only occurring next to each other at relatively low frequencies.

The fundamental challenge is to filter through the noise of a massive corpus to determine which patterns are significant and should be remembered and which are irrelevant. Many different approaches were tested but using Shannon Entropy to rate the statistical significance of different tokens was by far the most effective.

These challenges meant that while it was possible to devise a good tokenization system (one with about 80% accuracy), it was very difficult to do better than that unless we relied on a strong selective join feature. The issue with having an especially stringent selective join however was that it restricted the size of tokens formed and made it very difficult to generate tokens spanning multiple words.

References

[1] *Chunking mechanisms in human learning*. Gobert et al, 2001.

[2] *Neuronal Network Modelling From A Bayesian Perspective*. Hercus et al, 2001.

[3] *Mathematical Theory of Computation*. Shannon, 1948.