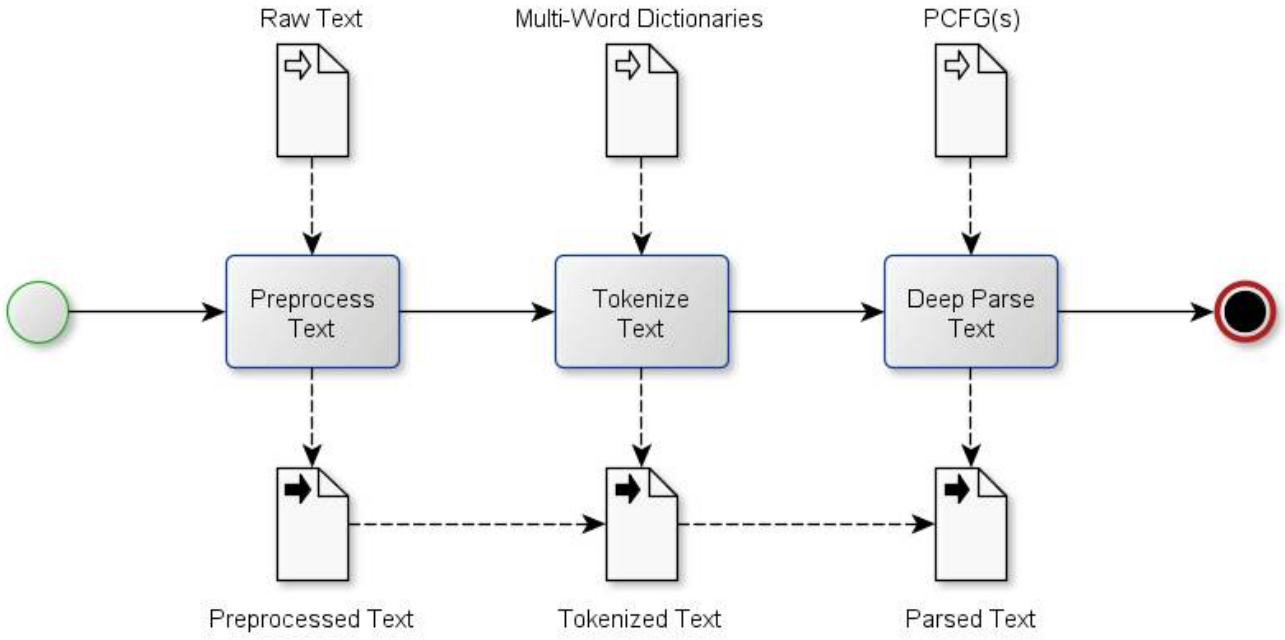
Tokenization

The process of segmenting running text into words and sentences.

Tokenization (or word segmentation) is the task of separating out (tokenizng) words from running text. In English, words are often separated from each other by blanks (white space), but white space is not always sufficient. Both “Los Angeles” and “rock 'n' roll” are individual words despite the fact that they contain spaces. We may also need to separate single words like “I'm” into separate words “I” and “am”.



White Space Tokenization

Word tokenization may seem simple in a language like English that separates words by a special 'space' character. Not every language does this (e.g. Chinese, Japanese, Thai). But a closer examination will make it clear that white space is not sufficient by itself even for English.

“I said, 'what're you? Crazy?'” said Sandowsky. “I can't afford to do that.”

	Naïve Whitespace Parser	Apache Open NLP 1.5.2 (using en-token.bin)	Stanford 2.0.3	custom	Hypothetical Tokenizer (Ideal Tokenization)
1		'	'	'	“
2	“I	I	I	I	I
3	Said,	said	said	said	said
4		,	,	,	,
5	'what're	'what	`	'	'
6			what	what're	what
7		're	're		are
8	you?	you	you	you	you
9		?	?	?	?
10	Crazy?'”	Crazy	Crazy	Crazy	Crazy
11		?	?	?	?
12		'	'	'	'
13	said	said	said	said	said
14	Sandowsky.	Sandowsky	Sandowsky	Sandowsky	Sandowsky
15		.	.	.	.
16		'	'	'	“
17	'I	I	I	I	I
18	can't	ca	ca	can't	can
19		n't	n't		not
20	afford	afford	afford	afford	afford
21	to	to	to	to	to
22	do	do	do	do	do
23	that.'	that	that	that	that
24		.	.	.	.
25		'	'	'	'

The naïve white space parser is shown to perform poorly here.

The Stanford tokenizer does somewhat better than the OpenNLP tokenizer, which is to be expected. The custom parser (included in the appendix) in the 4th column, does a nearly perfect job, though

without the enclitic expansion shown in the first hypothetical pass.

The more accurate (and complex) segmentation process in the fourth and fifth columns require a morphological parsing process.

We can address some of these issues in the first three examples by treating punctuation, in addition to white space, as a word boundary. But punctuation often occurs internally, in examples like u.s.a., Ph.D., AT&T, ma'am, cap'n, 01/02/06 and stanford.edu. Similarity, assuming we want 7.1 or 82.4 as a word, we can't segment on every period, since that would segment these into “7” and “1” and “82” and “4”. Should “data-base” be considered two separate tokens or a single token? The number “$2,023.74” should be considered a single token, but in this case, the comma and period do not represent delimiters, where in other cases they might. And should the “$” sign be considered part of that token, or a separate token in its own right?

The java.util.SimpleTokenizer class in Java is an example of a white space tokenizer, where you can define the set of characters that mark the boundaries of tokens. Another Java class, java.text.BreakIterator, can identify word or sentence boundaries, but still does not handle ambiguities.

Named Entity Extraction

It's almost impossible to separate tokenization from named entity extraction. It really isn't possible to come up with a generic set of rules that will handle all ambiguous cases within English; the easiest approach is usually just to have multi-word expression dictionaries.

Install Rational Software Architect on AIX 5.3

	Naïve Whitespace Parser	Hypothetical Tokenizer (Ideal Tokenization)
1	Install	Install
2	Rational	Rational Software Architect for WebSphere
3	Software	
4	Architect	
5	for	
6	WebSphere	
7	on	on
8	AIX	AIX 5.3
9	5.3	

Dictionaries will have to exist that express to the tokenization process that “Rational Software Architect for WebSphere” is a single token (a product), and “AIX 5.3” is likewise a single product.

The impact that tokenization has upon the rest of the process can not be understated. A typical next step, following tokenization, is to send the segmented text to a deep parser. In the first column, the rational product would end up being deep parsed into a structure like this:

OpenNLP 1.5.2 (en-parser-chunking.bin)

```
<node prob="0.99"
      span="Rational Software Architect for WebSphere" type="NP">
      <node prob="1.0"
            span="Rational Software Architect" type="NP">
            <node prob="0.86" span="Rational" type="NNP"/>
            <node prob="0.94" span="Software" type="NNP"/>
            <node prob="0.93" span="Architect" type="NNP"/>
      </node>
      <node prob="0.99" span="for WebSphere" type="PP">
            <node prob="0.93" span="for" type="IN"/>
           <node prob="1.0" span="WebSphere" type="NP">
            <node prob="0.24 span="WebSphere" type="NNP"/>
            </node>
      </node>
</node>
```

(Output from Stanford 2.0.3 is identical).

Note the formation of a prepositional phrase (PP) around "for WebSphere" and the noun phrase trigram "Rational Software Architect". If the sentence was semantically segmented with the aid of a multi-word dictionary, the output from the deep parser would have looked like this:

```
<node span="Rational Software Architect for WebSphere"
      type="NP">
     <node span="Rational Software Architect for WebSphere"
           type="NNP"/>
</node>
```

There is a single noun phrase containing one noun (NNP = singular noun[1]).

English Enclitics

A clitic is a unit whose status lies between that of an affix and a word. The phonological behavior of a clitic is like affixes; they tend to be short and unaccented. Their syntactic behavior is more like words, often acting as pronouns, articles, conjunctions, or verbs. Clitics preceding a word are called proclitics, and those following are enclictics.

English enclitics include:

The abbreviated forms of be:

- 'm in I'm
- 're in you're
- 's in she's

1 An overview of the Penn Treebank II Tags can be found here: http://bulba.sdsu.edu/jeanette/thesis/PennTags.html

The abbreviated forms of auxiliary verbs:

- ’ll in they’ll
- ’ve in they’ve
- ’d in you’d

Note that clitics in English are ambiguous. The word “she's” can mean “she has” or “she is”.

A tokenizer can also be used to expand clitic contractions that are marked by apostrophes, for example:

what're => what are
we're => we are

This requires ambiguity resolution, since apostrophes are also used as genitive markers as in “the book's over in the containers' above” or as quotative markers. While these contractions tend to be clictics, not all clictics are marked this way with contractions. In general, then, segmenting and expanding clitics and be done as part of a morphological parsing process.

Enclitic Analysis (New York Times):

Clitic	Example	Full Form	Total Frequency	Totals	% Occurrence
's	He's	He have, He Has	15,909,933	17,054,319	93.29%
're	You're	You are	381,176	17,054,319	2.24%
'm	I'm	I am	257,465	17,054,319	1.51%
've	They've	They have	242,249	17,054,319	1.42%
'll	We'll	We will	142,452	17,054,319	0.84%
'd	She'd	She would, She had	121,044	17,054,319	0.71%

Illustration 1: Analysis of NYT Corpus from 1987 - 2007

References:

1. Jurafsky, Dan. Speech and Language Processing, 2nd Edition. NJ: Pearson, 2009. Book.
2. Jackson, Peter. Natural Language Processing for Online Applications. PHL: John Benjamins Publishing Company, 2007. Book.

Appendix A – Clitic Anlaysis on the NYT Corpus

1. *Clitic Analysis – All.csv*
   Contains all clitics and term frequencies, sorted by alpha.

2. *Clitic Analysis - Min 5 Freq.csv*
   Contains all clitics and term frequencies (threshold >=5), sorted by alpha.

3. *Clitic Analysis.xls*
   Analysis of "*Clitic Analysis - Min 5 Freq.csv*" formatted in an Excel Spreadsheet.

4. *Clitic Analysis – Totals.xls*
   Summarized counts of clitics (table shown in this article)

Appendix B - Code

OpenNLP Tokenizer Code:

```
private String path;

private Tokenizer tokenizer;

private TokenizerModel tokenizerModel;

@Override
protected TokenizerModel initializeModel() {
	try {
		InputStream modelIn = new FileInputStream(getPath());
		try {
			return new TokenizerModel(modelIn);
		} catch (IOException e) {
			e.printStackTrace();
		} finally {
			if (modelIn != null) {
				try {
					modelIn.close();
				} catch (IOException e) {
				}
			}
		}
	} catch (FileNotFoundException e) {
		e.printStackTrace();
	}

	return null;
}

@Override
public String[] tokenize(String input) {
	return getTokenizer().tokenize(input);
}

/**
 * @return the path
 */
public String getPath() {
	return path;
}

/**
 * @return the tokenizer
 */
public Tokenizer getTokenizer() {
	if (null == tokenizer) {
```

```
			setTokenizer(new TokenizerME(getTokenizerModel()));
		}

		return tokenizer;
	}

	/**
	 * @return the tokenizerModel
	 */
	public TokenizerModel getTokenizerModel() {
		if (null == tokenizerModel) {
			setTokenizerModel(initializeModel());
		}
		return tokenizerModel;
	}

	/**
	 * @return {@link TokenizerModel}
	 */
	protected abstract TokenizerModel initializeModel();

	/**
	 * @param path the path to set
	 */
	public void setPath(String path) {
		this.path = path;
	}

	/**
	 * @param tokenizer the tokenizer to set
	 */
	public void setTokenizer(Tokenizer tokenizer) {
		this.tokenizer = tokenizer;
	}

	/**
	 * @param tokenizerModel the tokenizerModel to set
	 */
	public void setTokenizerModel(TokenizerModel tokenizerModel) {
		this.tokenizerModel = tokenizerModel;
	}

	@Override
	public String[] tokenize(File file) throws BusinessException {
		try {
			return tokenize(StreamUtilities.getInstance().toString(file));
		} catch (IOException e) {
			logger.error(e);
			throw new BusinessException("Invalid Input");
		}
	}

	@Override
	public String[] tokenize(InputStream input) throws BusinessException {
		try {
			return tokenize(StreamUtilities.getInstance().toString(input));
		} catch (IOException e) {
			logger.error(e);
```

```
                throw new BusinessException("Invalid Input");
        }
}
```

Stanford Tokenizer Code:

```
@Override
public List<String> tokenize(String input) throws BusinessException {
        List<String> list = new ArrayList<String>();

        TokenizerFactory<CoreLabel> tokenizerFactory =
                PTBTokenizer.factory(new CoreLabelTokenFactory(), "");

        List<CoreLabel> tokens =
                tokenizerFactory.getTokenizer(
                        new StringReader(input)).tokenize();

        for (CoreLabel token : tokens) {
                String value = token.toString();

                value = StringUtilities.substringBefore(
                        StringUtilities.substringAfter(
                                value, "TextAnnotation="), " ");

                list.add(value);
        }

        return list;
}
```

Custom Tokenizer:

```
@Override
public List<String> process(String input) throws BusinessException {
        List<String> tokens = new ArrayList<String>();
        StringBuilder sb = new StringBuilder();

        char[] arr = input.toCharArray();
        for (int i = 0; i < arr.length; i++) {

                char prior = (i - 1 > 0) ? arr[i - 1] : ' ';
                char current = arr[i];
                char next = (i + 1 < arr.length) ? arr[i + 1] : ' ';

                // extract acronyms
                // this will actually extract acronyms of any length
                // once it detects this pattern a.b.c
                // it's a greedy lexer that breaks at ' '
                if (CodeUtilities.isAlpha(current) && '.' == next) {

                        // Pattern-1 = U.S.A      (5 chars)
                        // Pattern-2 = U.S.A.     (6 chars)
                        if (i + 5 < input.length()) {

                                // Pattern-1
                                if (CodeUtilities.isAlpha(arr[i])
```

```
                        && '.' == arr[i + 1]
                        && CodeUtilities.isAlpha(arr[i + 2])
                        && '.' == arr[i + 3]
                        && CodeUtilities.isAlpha(arr[i + 4])) {

                        for (; i < arr.length && arr[i] != ' '; i++) {
                                sb.append(arr[i]);
                        }

                        // check for Pattern-2 (trailing '.')
                        if (i + 1 < input.length()
                                && '.' == arr[i + 1]) {
                                sb.append(arr[i++]);
                        }

                        addToken(tokens, sb);
                        sb = new StringBuilder();
                        continue;
                }
        }
}

if ('w' == current && '/' == next) {
        sb.append(current);
        sb.append(next);
        addToken(tokens, sb);
        sb = new StringBuilder();
        i += 1;
        continue;
}

// extract URLs
if ('h' == current && 't' == next) {
        if (i + 7 < input.length() &&
                "http://".equals(input.substring(i, i + 7))) {

                for (; i < arr.length && arr[i] != ' '; i++) {
                        sb.append(arr[i]);
                }

                addToken(tokens, sb);
                sb = new StringBuilder();
                continue;
        }
}

// extract windows drive letter paths
// c:/ or c:\
if (CodeUtilities.isAlpha(current) && ':' == next) {
        if (i + 2 < input.length()
                && (arr[i + 2] == '\\'
                || arr[i + 2] == '/')) {
                sb.append(current);
                sb.append(next);
                sb.append(arr[i + 2]);
                i += 2;
                continue;
        }
```

```
}

// keep numbers together when separated by a period
// "4.0" should not be tokenized as { "4", ".", "0" }
if (CodeUtilities.isNumber(current) && '.' == next) {
      if (i + 2 < input.length() &&
            CodeUtilities.isNumber(arr[i + 2])) {
            sb.append(current);
            sb.append(next);
            sb.append(arr[i + 2]);
            i += 2;
            continue;
      }
}

// keep alpha characters separated by hyphens together
// "b-node" should not be tokenized as { "b", "-", "node" }
if (CodeUtilities.isAlpha(current) && '-' == next) {
      if (i + 2 < input.length() &&
            CodeUtilities.isAlpha(arr[i + 2])) {
            sb.append(current);
            sb.append(next);
            sb.append(arr[i + 2]);
            i += 2;
            continue;
      }
}

// TODO: need a greedy look-ahead to
// avoid splitting this into multiple tokens
// "redbook@vnet.ibm.com" currently is
// tokenized as { "redbook@vnet", ".", "ibm", ".", "com" }
// need to greedily lex all tokens up to the space
// once the space is found, see if the last 4 chars are '.com'
// if so, then take the entire segment as a single token
// don't separate tokens concatenated with an underscore
// eg. "ws_srv01" is a single token, not { "ws", "_", "srv01" }
if (CodeUtilities.isAlpha(current) && '_' == next) {
      if (i + 2 < input.length() &&
            CodeUtilities.isAlpha(arr[i + 2])) {
            sb.append(current);
            sb.append(next);
            i++;
            continue;
      }
}

// extract twitter channels
if (('#' == current ||
      '@' == current) &&
      ' ' != next &&
      !CodeUtilities.isSpecial(next)) {
      sb.append(current);
      continue;
}

// keep tokens like tcp/ip and os/2 and system/z together
if (' ' != current && '/' == next) {
```

```
                        sb.append(current);
                        sb.append(next);
                        i++;
                        continue;
                }

                if (' ' == current) {
                        addToken(tokens, sb);
                        sb = new StringBuilder();
                        continue;
                }

                // don't tokenize on <word>'s or <words>'
                // but do tokenize on '<words>
                if ('\'' == current) {
                        if (' ' == prior) {
                                addToken(tokens, "'");
                        } else {
                                sb.append(current);
                        }

                        continue;
                }

                if (CodeUtilities.isSpecial(current)) {
                        addToken(tokens, sb);
                        addToken(tokens, String.valueOf(current));
                        sb = new StringBuilder();
                        continue;
                }

                sb.append(current);
        }

        if (0 != sb.length()) {
                addToken(tokens, sb);
        }

        return tokens;
}
```