ITCS 4111/5111: Introduction to NLP

Tokenization: From text to sentences and tokens

Razvan C. Bunescu
Department of Computer Science @ CCI
rbunescu@uncc.edu
Tokenization: From Text to Tokens

- **Tokenization** = segmenting text into tokens:
  - **token** = a sequence of characters, in a particular document at a particular position.
  - **type** = the class of all tokens that contain the same character sequence.
    - “... to be or not to be ...”  
    - “... so be it, he said ...”  
    - 3 tokens, 1 type
  - **term** = a (normalized) type that is included in the dictionary.
    - **text** = “to sleep perchance to dream”, “US ambassador dreams”
    - **tokens** = to, sleep, perchance, to, dream, US, ambassador, dreams
    - **types** = to, sleep, perchance, dream, US, ambassador, dreams
    - **terms** = sleep, perchance, dream, USA, ambassador (lemmas, norm)
Tokenization: From Text to Tokens

• Split on whitespace and non-alphanumeric?
  – Good as a starting point, but complicated by many tricky cases:
    • Appostrophes are ambiguous:
      – possessive constructions:
        » the books’s cover => the book s cover
      – contractions:
        » he’s happy => he is happy
        » aren’t => are not
    – quotations:
      » ‘let it be’ => let it be
Tokenization: From Text to Tokens

• Split on whitespace and non-alphanumeric?
  – Good as a starting point, but complicated by many tricky cases:
    • Whitespaces in proper names or collocations:
      – San Francisco => San_Francisco
        » how do we determine it should be a single token?
    • Hyphenations:
      – co-education => co-education
      – state-of-the-art => state of the art? state_of_the_art?
      – lowercase, lower-case, lower case => lower_case
      – Hewlett-Packard => Hewlett_Packard? Hewlett Packard?
    • Whitespaces and Hyphenations:
      – San Francisco-Los Angeles => San_Francisco Los_Angeles
Tokenization: From Text to Tokens

- Split on whitespace and non-alphanumerics?
  - Good as a starting point, but complicated by many tricky cases:
    - **Whitespaces and Hyphenations:**
      - split on hyphens and whitespaces, but use a phrase index.
    - **Unusual strings** that should be recognized as tokens:
      - C++, C#, B-52, C4.5, M*A*S*H.
    - **URLs, IP addresses, email addresses, tracking numbers.**
      - exclude numbers, monetary amounts, URLs from indexing?

- **Use same tokenization rules for all documents:**
  - e.g. training vs. testing.
Tokenization is Language Dependent

- Need to know the language of document/query:
  - **Language Identification**, based on classifiers trained on short character subsequences as features, is highly effective.
  - **French** (reduced definite article, postposed clitic pronouns):
    - l’ensemble, un ensemble, donne-moi.
  - **German** (compound nouns), need *compound splitter*:
    - Computerlinguistik
    - Lebensversicherungsgesellschaftsangestellter
  - **East Asian languages**, need *word segmenter*:
    - 莎拉波娃现在居住在美国东南部的佛罗里达。
      - Not always guaranteed a unique tokenization
    - Complicated in Japanese, with multiple alphabets intermingled.
Tokenization is Language Dependent

• Need to know the language of document/query:
  – Arabic and Hebrew:
    • Written right to left, but with certain items like numbers written left to right.
    • Words are separated, but letter forms within a word form complex ligatures

إستقلت الجزائر في سنة 1962 بع 132 عام من الاحتلال الفرنسي.

Algeria achieved its independence in 1962 after 132 years of French occupation.
Language Dependent Processing

- **Compound Splitting for German:**
  - usually implemented by finding segments that match against dictionary entries.

- **Word Segmentation for Chinese:**
  - ML sequence tagging models trained on manually segmented text:
    - Logistic Regression, HMMs, Conditional Random Fields.
  - Multiple segmentations are possible:

> Figure 2.4 Ambiguities in Chinese word segmentation. The two characters can be treated as one word meaning ‘monk’ or as a sequence of two words meaning ‘and’ and ‘still’.
From Tokens to Terms: Normalization

- **Token Normalization** = reducing multiple tokens to the same canonical term, such that matches occur despite superficial differences.

1. Create equivalence classes, named after one member of the class:
   - \{anti-discriminatory, antidiscriminatory\}
   - \{U.S.A., USA\}
     - but what about C.A.T vs. CAT?

2. Can complicate later processing tasks is annotation already done on original, unnormalized version of text:
   - Need to maintain positional correspondence between normalized token and its original, unnormalized version
From Tokens to Terms: Normalization

- **Accents and diacritics** in French:
  - résumé vs. resume.

- **Umlauts** in German:
  - Tuebingen vs. Tübingen

- **British vs. American spellings**:
  - colour vs. color.

- **Multiple formats for dates, times**:
  - 09/30/2013 vs. Sep 30, 2013.
From Tokens to Terms: Normalization

- **Case-Folding** = reduce all letters to lower case:
  - change Automobile at beginning of sentences to automobile.
  - how about Ferrari?
  - but may lead to unintended matches:
    - the Fed vs. fed.
    - Bush, Black, General Motors, Associated Press, ...

- **Heuristic** = lowercase only some tokens:
  - words at beginning of sentences.
  - all words in a title where most words are capitalized.

- **Truecasing** = use a classifier to decide when to fold:
  - trained on many heuristic features.
Lemmatization and Stemming

- **Lemmatization** = reduce a word to its base/dictionary form, i.e. its lemma:
  - is, am, are => be
  - car, cars => car

- Lemmatization commonly only collapses the different *inflectional* forms of a lemma:
  - saw => see (if verb), or saw (if noun).
From Tokens to Terms: Stemming

• **Stemming** = reduce *inflectional* and sometimes *derivationally* related forms of a word to a common base form i.e. the *stem*.
  - automate, automates, automatic, automation => automat
  - see, saw => s

• Crude affix chopping that is language dependent:

  *for example compressed and compression are both accepted as equivalent to compress.*

  *for example compress and compress ar both accept as equival to compress.*
Porter’s Algorithm

http://www.tartarus.org/~martin/PorterStemmer/

- The most common stemmer for English:
  - at least as good as other stemming options.
  - 5 phases of word reductions, applied sequentially.
  - conventions for rule selection and application:
    - select the reduction rule that applies to the longest suffix:
      | Rule | Example |
      |------|--------|
      | SSES → SS | caresses → caress |
      | IES → I | ponies → poni |
      | SS → SS | caress → caress |
      | S → SS | cats → cat |

- check the number of syllables, for suffix determination:

\[(m > 1) \quad \text{EMENT} \rightarrow\]

would map replacement to replac, but not cement to c.
Other Stemming Algorithms

• Lovins stemmer, Paice/Husk stemmer, Snowball:
  – http://www.comp.lancs.ac.uk/computing/research/stemming/

• Stemming is language- and often application-specific:
  – open source and commercial plug-ins.

• Does it improve IR performance?
  – mixed results for English: improves recall, but hurts precision.
    • operative (dentistry) ⇒ oper
  – definitely useful for languages with richer morphology:
    • Spanish, German, Finish (30% gains).
Sentence Segmentation

• Generally based on punctuation marks: ??!
  
  – Periods are ambiguous, as sentence boundary markers and abbreviation/acronym markers:
    
    • Mr., Inc., m.p.h.
  
  – Sometimes they mark both:
    
    • SAN FRANCISCO (MarketWatch) – Technology stocks were mostly in positive territory on Monday, powered by gains in shares of Microsoft Corp. and IBM Corp.

• Tokenization approaches:
  
  – Regular Expressions.
  
  – Machine Learning (state of the art).
Extracting Linguistic Features with spaCy

```python
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $11.1 million.")

for token in doc:
    print(token.text, token.lemma_, token.pos_, token.tag_, token.dep_,
          token.shape_, token.is_alpha, token.is_stop)
```

<table>
<thead>
<tr>
<th>Apple</th>
<th>Apple</th>
<th>PROPN</th>
<th>NNP</th>
<th>nsubj</th>
<th>Xxxxx</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>is</td>
<td>be</td>
<td>AUX</td>
<td>VBZ</td>
<td>aux</td>
<td>xx</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>looking</td>
<td>look</td>
<td>VERB</td>
<td>VBG</td>
<td>ROOT</td>
<td>xxxx</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>at</td>
<td>at</td>
<td>ADP</td>
<td>IN</td>
<td>prep</td>
<td>xx</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>buying</td>
<td>buy</td>
<td>VERB</td>
<td>VBG</td>
<td>pcomp</td>
<td>xxxx</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>U.K.</td>
<td>U.K.</td>
<td>PROPN</td>
<td>NNP</td>
<td>compound</td>
<td>X.X.</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>startup</td>
<td>startup</td>
<td>NOUN</td>
<td>NN</td>
<td>dobj</td>
<td>xxxx</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>for</td>
<td>for</td>
<td>ADP</td>
<td>IN</td>
<td>prep</td>
<td>xxx</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>SYM</td>
<td>$</td>
<td>quantmod</td>
<td>$</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>11.1</td>
<td>11.1</td>
<td>NUM</td>
<td>CD</td>
<td>compound</td>
<td>dd.d</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>million</td>
<td>million</td>
<td>NUM</td>
<td>CD</td>
<td>pobj</td>
<td>xxxx</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>PUNCT</td>
<td>.</td>
<td>punct</td>
<td>.</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
Apple is looking at buying U.K. startup for $1 billion. The deal is unlikely to go through.
By default, spaCy’s `nlp()` function runs an entire linguistic pipeline:

- **Tokenizer** segment text into tokens.
- **Tagger** assigns part-of-speech tags.
- **Parser** assigns dependency labels.
- **NER** detects and labels named entities.
- ** Lemmatizer** assigns base forms.
- **Textcat** assigns document labels.
- **Custom** assigns custom attributes, methods or properties.

But this is inefficient if we only need to tokenize …
Tokenization in spaCy

- Run only the pipeline component(s) that are needed. Two options:
  1. **Call the component directly.**
  2. Use the default pipeline but disable components that are not needed.

```python
from spacy.lang.en import English
nlp = English()
tokenizer = nlp.tokenizer
tokens = tokenizer("U.S. economy is healing, but there’s a long way to go. "
  "The spread of Covid-19 led to surge in orders for factory robots")
for token in tokens:
    print(token, end = ' ')
print()
```

U.S. economy is healing, but there’s a long way to go. The spread of Covid-19 led to surge in orders for factory robots.
Tokenization in spaCy

https://spacy.io/usage/processing-pipelines

- Run only the pipeline component(s) that are needed. Two options:
  1. Call the component directly.
  2. Use the default pipeline but disable components that are not needed.

```python
import spacy

# Load the tagger, ner, and parser but don't enable them.
nlp = spacy.load("en_core_web_sm", disable=['tagger, ner, parser'])
nlp.remove_pipe("parser")
nlp.remove_pipe("tagger")
nlp.remove_pipe("ner")

doc = nlp("U.S. economy is healing, but there’s a long way to go. "
          "The spread of Covid-19 led to surge in orders for factory robots.")

for token in doc:
    print(token, end = ' ')
print()
```

U.S. economy is healing, but there’s a long way to go. The spread of Covid-19 led to surge in orders for factory robots.
Sentence Segmentation in spaCy

• Run only the pipeline component(s) that are needed:
  – But spaCy by default uses the parser for sentence segmentation!
  • Use a rule-based (but not as accurate) Sentencizer.

```python
from spacy.lang.en import English
nlp = English()
sentencizer = nlp.create_pipe("sentencizer")
nlp.add_pipe(sentencizer)

doc = nlp("U.S. economy is healing, but there's a long way to go. "
          "The spread of Covid-19 led to surge in orders for factory robots."")
for sent in doc.sents:
    for token in sent:
        print(token, end = ' ')
print()
```

U.S. economy is healing, but there’s a long way to go.
The spread of Covid-19 led to surge in orders for factory robots.
Statistical Properties of Text
Statistical Properties of Text

- **Zipf’s Law** models the distribution of terms in a corpus:
  - How many times does the $k^{th}$ most frequent word appears in a corpus of size N words?
  - Important for determining index terms and properties of compression algorithms.

- **Heap’s Law** models the number of words in the vocabulary as a function of the corpus size:
  - What is the number of unique words appearing in a corpus of size N words?
  - This determines how the size of the inverted index will scale with the size of the corpus.
Word Distribution

• **A few words are very common:**
  – The 2 most frequent words (e.g. “the”, “of”) can account for about 10% of word occurrences.

• **Most words are very rare:**
  – Half the words in a corpus appear only once, called *hapax legomena* (Greek for “read only once”)

• **A “heavy tailed” or “long tailed” distribution:**
  – Since most of the probability mass is in the “tail” compared to an exponential distribution.
Zipf’s Law

12.5% OF THE PLANETS HAVE
71% OF THE MASS

#OCCUPYJUPITER
Word Distribution

Frequency vs. rank for all words in Moby Dick
Moby Dick:
• 44% hapax legomena
• 17% dis legomena

“Honorificabilitudinitatibus”:
• Shakespeare’s hapax legomenon
• longest word with alternating vowels and consonants
Zipf’s Law

• Rank all the words in the vocabulary by their frequency, in decreasing order.
  – Let $r(w)$ be the rank of word $w$.
  – Let $f(w)$ be the frequency of word $w$.

• Zipf (1949) postulated that frequency and rank are related by a power law:
  $$f(w) = \frac{c}{r(w)}$$
  – $c$ is a normalization constant that depends on the corpus.
Zipf’s Law

• If the most frequent term (the) occurs $f_1$ times:
  – Then the second most frequent term (of) occurs $f_1/2$ times.
  – The third most frequent term (and) occurs $f_1/3$ times, ...

• **Power Laws:** $y = cx^k$
  – Zipf’s Law is a power law with $k = -1$.
  – Linear relationship between log($y$) and log($x$):
    • $\log(y) = \log c + k \log(x)$
    • on a log scale, power laws give a straight line with slope $k$.

• Zipf is quite accurate, except for very high and low rank.
Zipf’s Law Fit to Brown Corpus

\[ f(w) = \frac{100000}{r(w)} \]
Mandelbrot’s Distribution

- The following more general form gives a bit better fit:
  \[ f = c / (r + \rho)^K \]

- When fit to Brown corpus:
  - \( c = 105.4 \)
  - \( K = -1.15 \)
  - \( \rho = 100 \)
Mandelbrot’s Law Fit to Brown Corpus

Mandelbrot’s function on Brown corpus
Vocabulary vs. Collection Size

• How big is the term vocabulary?
  – That is, how many distinct words are there?

• Can we assume an upper bound?
  – Not really upper-bounded due to proper names, typos, etc.

• In practice, the vocabulary will keep growing with the collection size.
Heap’s Law

• **Given:**
  - $M$ is the size of the vocabulary.
  - $T$ is the number of tokens in the collection.

• **Then:**
  - $M = kT^b$
  - $k$, $b$ depend on the collection type:
    • typical values: $30 \leq k \leq 100$ and $b \approx 0.5$ (square root).
    • in a log-log plot of $M$ vs. $T$, Heaps’ law predicts a line with slope of about $1/2$. 
Heap’s Law Fit to Reuters RCV1

• For RCV1, the dashed line $\log_{10} M = 0.49 \log_{10} T + 1.64$ is the best least squares fit.

• Thus, $M = 10^{1.64} T^{0.49}$ so $k = 10^{1.64} \approx 44$ and $b = 0.49$.

• For first 1,000,020 tokens:
  – Law predicts 38,323 terms;
  – Actually, 38,365 terms.
  $\Rightarrow$ Good empirical fit for RCV1!
Explanations

• **Zipf’s Law:**
  – Zipf’s explanation was his “principle of least effort”:
    • Balance between speaker’s desire for a small vocabulary and hearer’s desire for a large one.
  – Herbert Simon’s explanation is “rich get richer.”
  – Li (1992) shows that just random typing of letters including a space will generate “words” with a Zipfian distribution.

• **Heaps’ Law:**
  – Can be derived from Zipf’s law by assuming documents are generated by randomly sampling words from a Zipfian distribution.
Homework Assignment 1

• **Distributions of Words & Sentences:**
  1. The first task is to compute the frequency vs. rank distribution of the words in Moby Dick.
     • For this, you will need to tokenize the document and create a vocabulary mapping word types to their document frequency.
  2. The second task is to segment the document into sentences and compute the sentence length distribution.
     • Here you will experiment with spaCy's default sentence segmenter as well as the simple rule-based Sentencizer.