# Basic Text Processing

- **Regular Expressions**
- **Text Normalization**

# Basic Text Processing: Regular Expressions

# Regular Expressions

- **Regular expressions** are the most important tool to describe text patterns and we can use them to specify the strings to be extracted from the text.
  - Regular expressions are widely used in many text preprocessing tasks.
  - A set of text preprocessing tasks is called as **text normalization**.
    - Normalizing text means converting it to a more convenient, standard form.

**Tokenization**: Separating out or **tokenizing** words from text.

- English words are often separated from each other by whitespace (not enough).
  - For processing tweets we'll need to tokenize emoticons like **:)** or hashtags like **#nlproc**.

**Lemmatization**: Task of determining that words have the same root.

- Words *sings*, *singing*, *sang*, *sung* have the same root word (**lemma**) *sing*.
- Words *kitabım*, *kitaplar,…* have the same root word *kitap*.
- **Stemming:** a simpler version of lemmatization in which we mainly just strip suffixes from the end of the word.

**Sentence Segmentation**: breaking up a text into individual sentences.

# Regular Expressions

- Each **Regular Expression (RE)** represents a set of strings having certain pattern.
  - In NLP, we can use REs to find strings having certain patterns in a given text.
- Regular Expressions are an algebraic way to describe formal languages.
  - Regular Expressions describe exactly the regular languages.
- A regular expression is built up of simpler regular expressions (using defining rules).
- Simple Definition for Regular Expressions over alphabet $\Sigma$
  - $\boldsymbol{\varepsilon}$ is a regular expression
  - If $\mathbf{a} \in \boldsymbol{\Sigma}$, $\mathbf{a}$ is a regular expression
  - **or :** If $E_1$ and $E_2$ are regular expressions, then $\mathbf{E_1 \mid E_2}$ is a regular expression
  - **concatenation** : If $E_1$ and $E_2$ are regular expressions, then $\mathbf{E_1E_2}$ is a regular expression
  - **Kleene Closure:** If E is a regular expression, then $\mathbf{E^*}$ is a regular expression
  - **Positive Closure:** If E is a regular expression, then $\mathbf{E^+}$ is a regular expression

# Searching Strings with Regular Expressions
## (*using Python style REs*)

- How can we search for any of following strings?
  - woodchuck
  - woodchucks
  - Woodchuck
  - Woodchucks
- The **simplest kind of regular expression** is a sequence of simple characters.
  - The regular expression **b** will match with the string “b”.
  - The regular expression **bc** will match with the string “bc”.
  - The regular expression **woodchuck** will match with the string “woodchuck”.
  - The regular expression **woodchucks** will match with the string “woodchucks”.
  - The regular expression **woodchuck** will NOT match with the string “Woodchuck”.

# Regular Expressions: Disjunctions disjunction of characters []

- **Disjunction of Characters:** The **string of characters inside the braces [ ]** specifies a **disjunction** of characters to match.
- The regular expression **[wW]** matches patterns containing either w or W.

| Regular Expression | Matches |
|---|---|
| `[wW]oodchuck` | Woodchuck, woodchuck |
| `[1234567890]` | Any digit |

- **Ranges in []:** If there is a well-defined sequence associated with a set of characters, **dash (-)** in brackets can specify any one character in a **range**.

| Regular Expression | Matches |
|---|---|
| `[A-Z]` | An upper case letter |
| `[a-z]` | A lower case letter |
| `[0-9]` | A single digit |

# Regular Expressions: Disjunctions Negations in []

- **Negations in []:**
  - The square braces can also be used to specify what a single character cannot be, by use of the caret ^.
  - If the caret ^ is the first symbol after the open square brace [, the resulting pattern is negated.

| Regular Expression | Matches |
| --- | --- |
| `[^A-Z]` | Not an upper case letter |
| `[^a-z]` | Not a lower case letter |
| `[^Ss]` | Neither 'S' nor 's' |
| `[^e^]` | Neither e nor ^ |
| `a^b` | The pattern **a^b** |

# Regular Expressions: Disjunctions
# or (disjunction) operator | (pipe symbol)

- If $E_1$ and $E_2$ are regular expressions, then $\mathbf{E_1 \mid E_2}$ is a regular expression

| Regular Expression | Matches |
|---|---|
| `woodchuck\|groundhog` | **woodchuck** or **groundhog** |
| `a\|b\|c` | **a** , **b** or **c** |
| `[gG]roundhog\|[Ww]oodchuck` | **woodchuck** , **Woodchuck** , **groundhog** or **Groundhog** |
| `fl(y\|ies)` | **fly** or **flies** |

# Regular Expressions: Closure Operators Kleene * and Kleene +

- **Kleene * (closure) operator:** The Kleene star means “zero or more occurrences of the immediately previous regular expression.
- **Kleene + (positive closure) operator:** The Kleene plus means “one or more occurrences of the immediately preceding regular expression.

| Regular Expression | Matches |
|---|---|
| `ba*` | **b, ba, baa, baaa, ...** |
| `ba+` | **ba, baa, baaa, ...** |
| `(ba)*` | **ε, ba, baba, bababa, …** |
| `(ba)+` | **ba, baba, bababa, …** |
| `(b\|a)+` | **b, a, bb, ba, aa, ab, …** |

# Regular Expressions: {} . ?

- **{m,n}** causes the resulting RE to match from m to n repetitions of the preceding RE.
- **{m}** specifies that exactly m copies of the previous RE should be matched
- The question mark **?** marks **optionality of the previous expression**.

| Regular Expression | Matches |
|---|---|
| `woodchucks?` | **woodchuck** or **woodchucks** |
| `colou?r` | **color** or **colour** |
| `(a|b)?c` | **ac, bc, c** |
| `(ba){2,3}` | **baba, bababa** |

- A wildcard expression **dot .** matches any single character (except a carriage return).

| Regular Expression | Matches |
|---|---|
| `beg.n` | **begin, begun, begxn, …** |
| `a.*b` | any string starts with a and ends with b |

# Regular Expressions: Anchors ^ $

- **Anchors** are special characters that anchor regular expressions to particular places in a string.
- The **caret ^** matches the start of a string.
  - The regular expression ^The matches the word **The** only at the start of a string.
- The **dollar sign $** matches the end of a line.

| Regular Expression | Matches |
|---|---|
| `.$` | any character at the end of a string |
| `\.$` | dot character at the end of a string |
| `^[A-Z]` | any uppercase character at the beginning of a string |
| `^The dog\.$` | a string that contains only the phrase **The dog.** |

# Regular Expressions: Precedence of Operators

- The order precedence of RE operator precedence, from highest precedence to lowest precedence is as follows

  - Parenthesis ()
  - Counters * + ? {}
  - Sequences and anchors ^ $
  - Disjunction |

- The regular expression **the*** matches **theeeee** but not **thethe**
- The regular expression **(the)*** matches **thethe** but not **theeeee**

# Regular Expressions: backslashed characters

- Aliases for common sets of characters

| RE | Expansion | Match |
| --- | --- | --- |
| `\d` | `[0-9]` | any digit |
| `\D` | `[^0-9]` | any non-digit |
| `\w` | `[a-zA-Z0-9_]` | any word character |
| `\W` | `[^a-zA-Z0-9_]` | any non-word character |
| `\s` | `[ \t\n\r\f\v]` | any whitespace character |
| `\S` | `[^ \t\n\r\f\v]` | any non-whitespace character |

# Regular Expressions: backslashed characters

- Special characters need to be **backslashed**.

<table>
<tr><th>RE</th><th>Match</th></tr>
<tr><td><code>\b</code></td><td>a word boundary: A word boundary is the position between a word character (\w) and a non-word character (\W).</td></tr>
<tr><td><code>\B</code></td><td>a non-word boundary: This is the opposite of \b, and it matches any position that is not a word boundary.</td></tr>
<tr><td><code>\n</code></td><td>a newline character</td></tr>
<tr><td><code>\t</code></td><td>a tab character</td></tr>
<tr><td colspan="2">escaping other special characters: We can escape almost any special character (like *, +, ?, (, ), {, }, [, ], |, ^, $, ., and \\) by preceding it with a backslash (\).</td></tr>
<tr><td colspan="2"><code>\. \+ * \\ \? \( \) \[ \] \| \^ \$</code></td></tr>
</table>

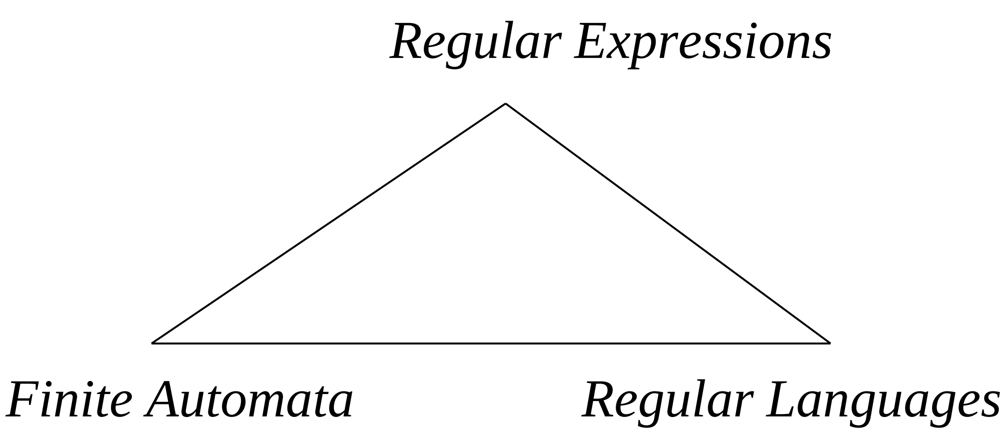
# Regular Expressions: Example

- We want to write a RE to find cases of the English article **the**
  - We can use **findall** method in **re** library for tokenization.

```
import re
sentence = "The book and the other book are not theology books. We breathe the air."

pre.findall(r"the",sentence)
  ➔ ['the', 'the', 'the', 'the', 'the']
    The book and the other book are not theology books. We breathe the air.

re.findall(r"[tT]he",sentence)
  ➔ ['The', 'the', 'the', 'the', 'the', 'the']
    The book and the other book are not theology books. We breathe the air.

re.findall(r"[tT]he\b",sentence)
  ➔ ['The', 'the', 'the', 'the']
    The book and the other book are not theology books. We breathe the air.

re.findall(r"\b[tT]he",sentence)
  ➔ ['The', 'the', 'the', 'the']
    The book and the other book are not theology books. We breathe the air.

re.findall(r"\b[tT]he\b",sentence)
  ➔ ['The', 'the', 'the']
    The book and the other book are not theology books. We breathe the air.
```

# Regular Expressions & FSAs

- Any regular expression can be realized as a **finite state automaton (FSA)**
- There are two kinds of FSAs
  - **Deterministic Finite State Automatons (DFAs)**
  - Non-deterministic Finite State Automatons (NFAs)
    - Any NFA can be converted into a corresponding DFA.
- A DFA (and a regular expression) represents a **regular language**.

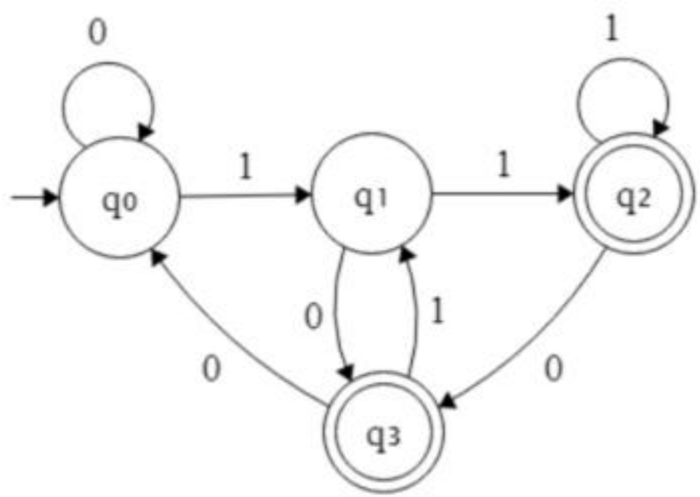


# Regular Expressions: A DFA and A NFA

- **A regular language:** The strings whose second characters from the right end are 1.

- Regular Expression for this regular language: **(0|1)*1(0|1)**

- A DFA for this language :



# Formal Definition of Finite-State Automaton

- FSA is $Q \times \Sigma \times q_0 \times F \times \delta$

- Q: a finite set of N states $q_0, q_1, \ldots q_N$
- $\Sigma$: a finite input alphabet of symbols
- $q_0$: the start state
- F: the set of final states
- $\delta(q,i)$: transition function
  - DFA : There is exactly one arc leaving a state q with a symbol a. There is no arc with the empty string.

# Basic Text Processing: Text Normalization

# Text Normalization

- Almost every natural language processing task needs to do text normalization.
- Three tasks are commonly applied as part of any normalization process:
  1. Segmenting/tokenizing words from the text
  2. Normalizing word formats
  3. Segmenting sentences in the text.

# Words

- Before processing words, we need to decide what counts as a **word**.
- How many words are in the following sentence?

  He stepped out into the hall, was delighted to encounter a water brother.

  – If we do NOT count punctuations as words → 13 words
  – If we count punctuations as words → 15 words
- Punctuations can be useful to identify boundaries of things and some aspects of meaning.

- Are capitalized tokens and uncapitalized tokens the same word?
  – The and the big possibly
  – US and us may be not (US: united states of America)

# Words

- Are the inflected forms like cat and cats the same word?
- They have the same **lemma** cat, but they have different **wordforms**.
- A **lemma** is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense.
- The **wordform** is the full inflected or derived form of the word.
  - For morphologically complex languages, we often need to deal with lemmatization.
  - For many tasks in English, however, wordforms are sufficient.

# Words: *How many words are there in English?*

- A **type** is a distinct Word in a corpus.
- **V: Vocabulary** is the set of types.
  - **|V|** is the size of the vocabulary.
- Each **word** in a corpus is a **token.**
  - N is the number of tokens in the corpus.

| **Corpus** | **# of Tokens = N** | **# of Types = \|V\|** |
|---|---|---|
| Shakespeare | 884,000 | 31 thousand |
| Switchboard phone conversations | 2.4 million | 20 thousand |
| Brown corpus | 1 million | 38 thousand |
| Google N-grams | 1 trillion | 13 million |

# Word Tokenization and Normalization

- **Tokenization** is the task of segmenting the text into words.
- **Normalization** is the task of putting words in a standard format.

- We can use **regular expressions** to segment the text into words for **tokenization** task.
  - Since tokenization needs to be run before any other language processing, it is important for it to be very fast.
  - The method for tokenization/normalization is to use **deterministic algorithms based on regular expressions** compiled into very efficient **finite state automata**.

# Tokenization

- Normally we want to break off **punctuations** as separate tokens, but sometimes we want to keep them in words internally.
- Punctuations as separate tokens: He ate apple**,** orange and banana**.**
- Punctuations kept internally:
  - m.p.h. Ph.D. AT&T Prices: $43.55 Dates: 27/09/2019
  - URLs: http://www.hacettepe.edu.tr Twitter hashtags: #nlproc
- A tokenizer can also expand **clitic** contractions that are marked by apostrophes.
  - what're to two tokens what are
  - we're to two tokens we are
- Tokenization algorithms may also tokenize multiword expressions like New York or rock 'n' roll as a single token.

# Tokenization in Python

- We can use methods in **re** or **nltk** library for tokenization.
  - We can use **findall** method in **re** library for tokenization.
  - We can use **word_tokenize** or **regexp_tokenize** methods in **nltk** library for tokenization.
- Token patterns are described with regular expressions.

```
re.findall(pattern,string)
```

- Returns all non-overlapping matches of ***pattern*** in ***string***, as a list of strings or tuples.
- The ***string*** is scanned left-to-right, and matches are returned in the order found.
- The result depends on the number of capturing groups in the pattern.
  - If there are no groups, return a list of strings matching the whole pattern.
  - If there is exactly one group, return a list of strings matching that group.
  - If multiple groups are present, return a list of tuples of strings matching the groups.

```
nltk.regexp_tokenize(string,pattern)
```

```
nltk.word_tokenize(string)
```

- `nltk.word_tokenize` may require to download some library.
  - `nltk.download('punkt_tab')`

# Tokenization in Python

- In regular expressions, **parentheses** are used for two main purposes:
  1. **Grouping:** Parentheses `(...)` are used to group parts of the pattern together.
  2. **Non-Grouping:** Parentheses `(?:...)` are used to create non-capturing groups.
     - This group is still used to group parts of the pattern together (like regular parentheses), but it does not create a capturing group.

```
str = "the evening shows start at 7:00pm and 10:15pm. the morning show at 9:00."

# Parentheses identify a group within the pattern
matches = re.findall(r"(\d\d?:\d\d)(am|pm)?", str)
➔ [('7:00', 'pm'), ('10:15', 'pm'), ('9:00', '')]

matches = re.findall(r"(?:\d\d?:\d\d)(am|pm)?", str)
➔ ['pm', 'pm', '']

matches = re.findall(r"(\d\d?:\d\d)(?:am|pm)?", str)
➔ ['7:00', '10:15', '9:00']

matches = re.findall(r"(?:\d\d?:\d\d)(?:am|pm)?", str)
➔ ['7:00pm', '10:15pm', '9:00']
```

# Tokenization in Python

```
import nltk
import re
sentence = 'That U.S.A. poster-print costs $12.40...'

nltk.word_tokenize(sentence)
➔ ['That', 'U.S.A.', 'poster-print', 'costs', '$', '12.40', '...']

pattern = r"\w+"  # tokens are any non-empty sequence of word characters
nltk.regexp_tokenize(sentence,pattern)      # or findall of re
re.findall(pattern,sentence)
➔ ['That', 'U', 'S', 'A', 'poster', 'print', 'costs', '12', '40']

pattern = r"\S+"  # tokens are white space separated
nltk.regexp_tokenize(sentence,pattern)
➔ ['That', 'U.S.A.', 'poster-print', 'costs', '$12.40...']

pattern = r"""(?x)       # multiline and set verbose for regular expressions
   (?:[A-Z]\.)+          # abbreviations
   | \w+(?:-\w+)*        # words with optional internal hyphens
   | \$?\d+(?:\.\d+)?%?  # currency, percentages, e.g. $12.40, 45%
   | \.\.\.              # ellipsis (three dots)
   | [][.,;\"\'?():_-]   # other single character tokens
   """

nltk.regexp_tokenize(sentence,pattern)
➔ ['That', 'U.S.A.', 'poster-print', 'costs', '$', '12.40', '...']
```

# Tokenization: Language Issues

- French:
  - ***L'ensemble*** to two words ***un ensemble***
- German noun compounds are not segmented:
  - ***Lebensversicherungsgesellschaftsangestellter***
  - German tokenizer needs **compound splitter**.
- Chinese and Japanese no spaces between words:
  - **莎拉波娃**现在居住在美国东南部的佛罗里达。
  - **莎拉波娃** 现在 **居住** **在** **美国** 东南部 **的** **佛**罗里达
  - Sharapova now lives in US southeastern Florida

# Word Tokenization in Chinese

- Word tokenization is also called **Word Segmentation**
- Chinese words are composed of characters
  - Characters are generally 1 syllable and 1 morpheme.
  - Average word is 2.4 characters long.

- Standard baseline segmentation algorithm: **Maximum Matching**

Given a wordlist of Chinese, and a string.

1. Start a pointer at the beginning of the string
2. Find the longest word in dictionary that matches the string starting at pointer
3. Move the pointer over the word in string
4. Go to 2

# Max-match segmentation

- Thecatinthehat — the cat in the hat
- Thetabledownthere — the table down there
  theta bled own there
  - Doesn't generally work in English!

- But works well in Chinese
  - 莎拉波娃现在居住在美国东南部的佛罗里达。
  - 莎拉波娃 现在 居住 在 美国 东南部 的 佛罗里达
- Modern probabilistic segmentation algorithms even better

# Subword Tokenization

- Another option for text tokenization → **Subword Tokenization**
- Instead of white-space word segmentation, use single-character segmentation
- Use the data (**training corpus**) to tell us how to tokenize.
- **Subword tokenization** (because tokens can be parts of words as well as whole words)
- To deal with this unknown word problem, modern tokenizers (used by Large Language Models) automatically induce sets of tokens that include tokens smaller than words, called **subwords**.
  - Subwords can be arbitrary substrings, or they can be meaning-bearing units like the morphemes -est or -er.
  - In modern tokenization schemes, most tokens are words, but some tokens are frequently occurring morphemes or other subwords like -er.
  - Every unseen word like lower can thus be represented by some sequence of known subword units, such as low and er, or even as a sequence of individual letters if necessary.

# Subword Tokenization

- Three common algorithms for subword tokenization:
  - **Byte-Pair Encoding (BPE)** (Sennrich et al., 2016)
  - **Unigram language modeling tokenization** (Kudo, 2018)
  - **WordPiece** (Schuster and Nakajima, 2012)

- All algorithms have 2 parts:
  - A **token learner** that takes a raw training corpus and induces a vocabulary (a set of tokens).
  - A **token segmenter** that takes a raw test sentence and tokenizes it according to that vocabulary

# Byte-Pair Encoding (BPE) - Token Learner

- Let vocabulary be the set of all individual characters

  = {A, B, C, D,…, a, b, c, d….}

- Repeat:
  - Choose the two symbols that are most frequently adjacent in the training corpus (say 'A', 'B')
  - Add a new merged symbol 'AB' to the vocabulary
  - Replace every adjacent 'A' 'B' in the corpus with 'AB’.

- Until k merges have been done.

# Byte-Pair Encoding (BPE) - Token Learner

**function** BYTE-PAIR ENCODING(strings $C$, number of merges $k$) **returns** vocab $V$

$V \leftarrow$ all unique characters in $C$ # initial set of tokens is characters
**for** $i = 1$ **to** $k$ **do** # merge tokens $k$ times
$t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in $C$
$t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
$V \leftarrow V + t_{NEW}$ # update the vocabulary
Replace each occurrence of $t_L, t_R$ in $C$ with $t_{NEW}$ # and update the corpus
**return** $V$

# Byte-Pair Encoding (BPE)
# - Token Learner: Example

- Most subword algorithms are run inside space-separated tokens.
- So we commonly first add a special end-of-word symbol '_' before space which comes after each word in training corpus
- Next, separate into letters (character).

- Tiny Corpus:

  low low low low low lowest lowest newer newer newer

  newer newer newer wider wider wider new new

- Add end-of-word tokens

➔ Initial Vocabulary: _, d, e, i, l, n, o, r, s, t, w

# Byte-Pair Encoding (BPE)
# - Token Learner: Example

| corpus | | vocabulary |
|---|---|---|
| 5 | l o w _ | _, d, e, i, l, n, o, r, s, t, w |
| 2 | l o w e s t _ | |
| 6 | n e w e r _ | |
| 3 | w i d e r _ | |
| 2 | n e w _ | |

- Merge **e r** to **er**

| corpus | | vocabulary |
|---|---|---|
| 5 | l o w _ | _, d, e, i, l, n, o, r, s, t, w, er |
| 2 | l o w e s t _ | |
| 6 | n e w er _ | |
| 3 | w i d er _ | |
| 2 | n e w _ | |

# Byte-Pair Encoding (BPE)
# - Token Learner: Example

| **corpus** | | **vocabulary** |
|---|---|---|
| 5 | l o w _ | _, d, e, i, l, n, o, r, s, t, w, er |
| 2 | l o w e s t _ | |
| 6 | n e w er _ | |
| 3 | w i d er _ | |
| 2 | n e w _ | |

- Merge **er _** to **er_**

| **corpus** | | **vocabulary** |
|---|---|---|
| 5 | l o w _ | _, d, e, i, l, n, o, r, s, t, w, er, er_ |
| 2 | l o w e s t _ | |
| 6 | n e w er_ | |
| 3 | w i d er_ | |
| 2 | n e w _ | |

# Byte-Pair Encoding (BPE) - Token Learner: Example

| **corpus** | | **vocabulary** |
|---|---|---|
| 5 | l o w _ | _, d, e, i, l, n, o, r, s, t, w, er, er_ |
| 2 | l o w e s t _ | |
| 6 | n e w er_ | |
| 3 | w i d er_ | |
| 2 | n e w _ | |

- Merge **n e** to **ne**

| **corpus** | | **vocabulary** |
|---|---|---|
| 5 | l o w _ | _, d, e, i, l, n, o, r, s, t, w, er, er_, ne |
| 2 | l o w e s t _ | |
| 6 | ne w er_ | |
| 3 | w i d er_ | |
| 2 | ne w _ | |

# Byte-Pair Encoding (BPE)
# - Token Learner: Example

- Next Merges:

| Merge | Current Vocabulary |
|---|---|
| (ne, w) | _, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new |
| (l, o) | _, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo |
| (lo, w) | _, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low |
| (new, er_) | _, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_ |
| (low, _) | _, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_ |

# Byte-Pair Encoding (BPE) - Token Segmenter

- On the test data, run each merge learned from the training data:
  - Greedily
  - In the order we learned them

- So: merge every e r to er, then merge er _ to er_, etc.
- Result:
  - Test set "n e w e r _" would be tokenized as a full word "newer_"
  - Test set "l o w e r _" would be two tokens: "low er_"

# Properties of Byte-Pair Encoding Tokens

- Usually include frequent words
- And frequent subwords
  - Which are often morphemes like -*est* or –*er*

- A **morpheme** is the smallest meaning-bearing unit of a language
  - *unlikeliest* has 3 morphemes *un-*, *likely*, and -*est*

# Text Normalization

- **Tokens** can also be **normalized**, in which a single normalized form is chosen for words with multiple forms like USA and US.
  - This standardization may be valuable, despite the spelling information that is lost in the normalization process.
  - For information retrieval, we want a query for US to match a document that has USA.

- **Case folding** is another kind of normalization: Reduce all letters to lower case.
  - For most applications (information retrieval), case folding is helpful.
  - For some NLP applications (MT, information extraction) cases can be helpful.
    - US versus us are important

# Lemmatization

- **Lemmatization** is the task of determining that two words have the **same root**, despite their surface differences.
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
- **Lemmatization**: have to find correct dictionary **headword form** of the Word.

- The most sophisticated methods for lemmatization involve complete **morphological parsing** of the word.
- **Morphology** is the study of the way words are built up from smaller meaning-bearing units called **morphemes**.
- Two broad classes of morphemes can be distinguished:
  - **Stems :** the central morpheme of the word, supplying the main meaning
  - **Affixes** : adding "additional" meanings of various kinds.

# Lemmatization

- **Lemmatization** algorithms can be complex.
- For this reason we sometimes make use of a simpler but cruder method, which mainly consists of chopping off word-final affixes.
- This naive version of morphological analysis is called **stemming**.
- One of the most widely used stemming algorithms is **Porter Stemmer.**
  - The algorithm is based on series of **rewrite rules** run in series, in which the output of each pass is fed as input to the next pass.
  - Some rules are:
    - ATIONAL → ATE (e.g., relational → relate)
    - ING → ε if stem contains vowel (e.g., motoring → motor)
    - SSES → SS (e.g., grasses → grass)

# Sentence Segmentation

- **Sentence segmentation** is another important step in text processing.
- The most useful cues for segmenting a text into sentences are **punctuation**, like **periods**, **question marks**, **exclamation points**.
- Question marks and exclamation points are relatively unambiguous markers of sentence boundaries.
- Periods, on the other hand, are more ambiguous.
  - Abbreviations like Inc. or Dr.
  - Numbers like .02% or 4.3
- Build a binary classifier
  - Looks at a "."
  - Decides EndOfSentence/NotEndOfSentence
  - Classifiers: hand-written rules, regular expressions, or machine-learning

# Summary

- The **regular expression** language is a powerful tool for pattern-matching.
- **Word tokenization** and **normalization** are generally done by cascades of simple ***regular expression substitutions*** or finite automata.