Byte Pair Encoding (BPE)

- The byte-pair encoding (BPE) algorithm was introduced by Sennrich et al., 2016.
- The BPE token learner begins with a vocabulary that is just the set of all individual characters.
- It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say 'A', 'B'), adds a new merged symbol 'AB' to the vocabulary, and replaces every adjacent 'A' 'B' in the corpus with the new 'AB'.
- It continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus is a parameter of the algorithm.
- The resulting vocabulary consists of the original set of characters plus k new symbols.

BPE

Suppose we have only the following 4 words in our corpus: "hug", "pug", "pun", "bun", "hugs".

Their frequencies are : ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5).

1. BPE starts by splitting the words to characters and saves the unique characters as the base vocabulary → V = ["b", "g", "h", "n", "p", "s", "u"].
2. After tokenizing using the base vocabulaty, the corpus becomes ("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
3. Then, it passes through the corpus word by word and token by token and count the pairs of characters:
    1. “hu”:10+5=15, “ug”:10+5+5=20, “pu”:5+12=17, “un”:4+12=16, “bu”:4, “gs”:5
    2. “ug” is the most frequent pattern → the first merge rule is (“u”, “g”) = “ug”
    3. V = ["b", "g", "h", "n", "p", "s", "u“, “ug”]
4. Next, we apply this merge rule to all the tokens → the corpus becomes: Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)
5. Similarly to step 3, BPE passes through the corpus word by word and token by token and count the new pairs of characters:
    1. “hug”:10+5=15, “pug”:5, “pu”:12, “un”:12+4=16, “bu”:4, “ugs”:5
    2. “un” is the most frequent pattern → the first merge rule is (“u”, “n”) = “un”
    3. V = ["b", "g", "h", "n", "p", "s", "u“, “ug”, “un”]
6. Next, we apply this merge rule to all the tokens → the corpus becomes: Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("h" "ug" "s", 5)
7. And we continue like this until we reach the desired vocabulary size.

Chap 1: Regular Expression, & Text Normalization

By Waad ALMASRI

Materials

- Regular Expressions (Regex)
- Text Normalization

Regex

https://regexr.com/

Definition & Application

Definition

- A language for specifying text search strings
- Useful to find a pattern we are searching for in a text or a corpus of texts

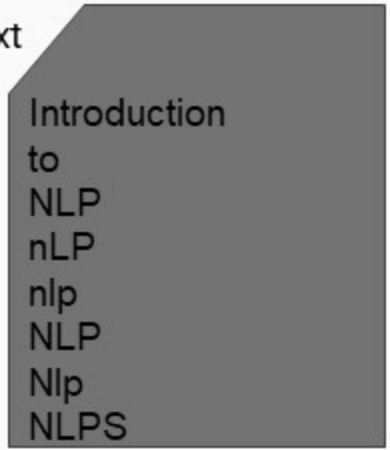
Application:

- Unix command-line tool Grep
- The search can be designed to return all matches or only the first match

Regex Rules

- Simple regex = sequence of simple characters
  - Example: grep NLP doc.txt

doc.txt




```
Apples-MacBook-Pro:courses apple$ grep NLP chap1-test-grep-regex.txt
NLP
NLP
NLPS
```

Regex Rules

- Simple regex = sequence of simple characters
- Regex are case sensitive
- To search a pattern while considering upper/lower cases, use a disjunction (the “or” operator)
  - Example

```
Apples-MacBook-Pro:courses apple$ grep [nN][lL][pP] chap1-test-grep-regex.txt
NLP
nLP
nlp
NLP
Nlp
NLPS
Apples-MacBook-Pro:courses apple$
```

Regex Rules

- What about numbers?
  - Example: we need to clean a text corpus from all numerical data. First step would be to find numbers in the corpus. To do so, there are 2 ways:

Regex Rules

- What about numbers?
  - Example: we need to clean a text corpus from all numerical data. First step would be to find numbers in the corpus. To do so, there are 2 ways:

Classical disjunction

```
Apples-MacBook-Pro:courses apple$ grep [0123456789] chap1-test-grep-regex.txt
123
Class2021
Inge3
User1
Id1234567
```

A range (more compact regex)

```
Apples-MacBook-Pro:courses apple$ grep [0-9] chap1-test-grep-regex.txt
123
Class2021
Inge3
User1
Id1234567
Apples-MacBook-Pro:courses apple$
```

Regex Rules

- Ranges can also be used with letters
  - Example:
    - [A-Z] = all upper case letters
    - [a-z] = all lower case letters
    - [a-zA-Z] = all letters
- Now, what if we would like to capture all digits except for zeros?
  - A caret ^ between square braces

RE	Match (single characters)	Example Patterns Matched
/[^A-Z]/	not an upper case letter	"Oyfn pripetchik"
/[^Ss]/	neither 'S' nor 's'	"I have no exquisite reason for't"
/[^.]/	not a period	"our resident Djinn"
/[e^]/	either 'e' or '^'	"look up ^ now"
/a^b/	the pattern 'a^b'	"look up a^b now"

Figure 2.4 The caret ^ for negation or just to mean ^. See below re: the backslash for escaping the period.

Regex Rules

- A caret has 3 uses:
  - To indicate negation “inside of square brackets” (as previously shown)
  - To match the start of a line
    - Example:
      - ^The = matches all the words “The” at the beginning of the lines
  - Simply to mean a caret, use the backslash “ \” before the caret
    - Example:
      - e\^[0-9] = matches e^0, e^1, etc.

Regex Rules

- What if we want to specify something like option1 or nothing?
  - We use **the question mark "?"**
  - Example:
    - Neighbou?r = neighbor or neighbour
    - AAA?h = AAh or AAAh
- However, in a real corpus, like social media texts, you will find things like AAAAAAAAh or Ah or others, To detect all these variations we use:
  - **Kleene *** = zero or more occurrences of the immediately previous character or regex
    - AA*h = Ah or AAh or AAAh or AAA.....Ah
  - Or **Kleene +** = one or more occurences of the immediately previous character or regex
    - A+h = Ah or AAh or AAA....Ah
- Exercise:
  - We need to search for prices in a corpus, what is the corresponding regex for prices?

Regex Rules

- The wildcard “.” or what is called the period = any single character
  - Example: search for all lines where the word “because” appears twice

Regex Rules

- The wildcard "." or what is called the period = any single character
  - Example: search for all lines where the word "because" appears twice
    - because.*because
      - .* = any string of characters
      - because.*because = any string of characters between 2 "because"
  - NB: if we want to search for the punctuation "period", we add a backslash in front of it
    - Example: [0-9]+[\.,][0-9]+ finds strings like 1.1 or 11.25 or 12,52 or 125874,12595 etc.
  - Another example: find all the variation of the verb "sing" in a corpus
    - s.ng = sing/sang/song/smng/ etc.

Regex Rules

- \$ matches the end of a line
  - Example: find lines that end with an exclamation mark
    - !\$
- \b matches a word boundary
  - Example: find lines where the word john99 appears
    - \bjohn99\b
- \B matches a non-word boundary
  - Example: find all lines with the symbol £
    - \B£\B

Regex Rules

Exercise: Let a regex be \b99\b and a corpus “line1 = There are 99 bottles of beer in the refrigerator. line 2 = There are 299 bottles of beer in the refrigerator. line3 = a beer bottle costs $99.” This regex matches ...

a) line1 only
b) line2 only
c) line3 only
d) Line1 and line2
e) Line1 and line3
f) Line2 and line3
g) all lines
h) none

Regex Rules

Exercise: Let a regex be \b99\b and a corpus “line1 = There are 99 bottles of beer in the refrigerator. line 2 = There are 299 bottles of beer in the refrigerator. line3 = a beer bottle costs $99.” This regex matches ...

a) line1 only
b) line2 only
c) line3 only
d) Line1 and line2
e) Line1 and line3
f) Line2 and line3
g) all lines
h) none

Reminder: \b matches a word boundary and a word in programming language is any sequence of digits, underscores or letters. Thus, \b99\b matches the word “99” in line1 because “99” follows a space and the “99” in line3 because 99 follows a “$” sign which is not a word but not in line2 because “99” follows a number “2”, in other terms, “299” is word that is different from “99”.

Disjunction, Grouping and Precedence

Disjunction

- The disjunction operator is the pipe symbol "|". It matches sequence1 or sequence2
  - Example: find in text lines mentioning molecule_A or molecule_B
    - molecule_A|molecule_B

Grouping

- Another more compact way to do so is by using the parenthesis operators
    - molecule_(A|B)
    - Because molecule_A|B matches the sequences molecule_A and B, which is not what we want
  - Enclosing a sequence in parenthesis makes it act as a single character for operators like the pipe symbol
  - Unlike "|", the kleene * is an operator that by default applies only to a single character and not a whole sequence

Disjunction, Grouping and Precedence

- Operators have precedence one on another i.e. a sort of a hierarchy (for example like mathematical operations: the product and division have a precedence over the sum and difference).
- from highest precedence to lowest precedence:

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

Regex Greedy and Non-Greedy

- Regex as defined previously are all greedy.
  - Example:
    - Corpus = <head> NLP Lesson </head>
    - A Regex of the form <.+> would match all the line <head> NLP Lesson </head> instead of only matching <head> and </head>. That is because Regex are very greedy.
    - Thus, to make it non-greedy (i.e. make it lazy) the question mark ? is added. By adding the ? after the +, we tell it to repeat as few times as possible, so the first > it comes across, is where we want to stop the matching.

Regex

RE	Expansion	Match	First Matches
\d	[0-9]	any digit	Party␣of␣5
\D	[^0-9]	any non-digit	Blue␣moon
\w	[a-zA-Z0-9_]	any alphanumeric/underscore	Daiyu
\W	[^\w]	a non-alphanumeric	!!!!
\s	[␣\r\t\n\f]	whitespace (space, tab)	
\S	[^\s]	Non-whitespace	in␣Concord

Figure 2.8 Aliases for common sets of characters.

RE	Match
*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression
{,m}	up to *m* occurrences of the previous char or expression

Figure 2.9 Regular expression operators for counting.

Regex - Exercice

Find the article “the” in a text

Regex - Exercice

Find the article “the” in a text - 3 mins

1. the
2. [tT]he
3. \b[tT]he\b
4. [^a-zA-Z][tT]he[^a-zA-Z]
5. (^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)

Regex - Exercice

Find the regex that matches a computer specs:

Example: “6 GHz 500 GB $1000”

Substitution & Capture Groups

- Substitution in Regex
  - It is replacing a string characterized by a regex by another string
  - Command line is : *s regex-expression pattern*
  - Example: put all numbers in a text into inequality symbols i.e. replace 12 by <12>
    - *s ([0-9]+) <\1>*
    - *s=replace the regex-expression [0-9]+ by <[0-9]+>*
- The parentheses are used to store a pattern in memory or what is called a **capture group**
  - Capture groups are stored in a numbered registered i.e. \1, \2, \3, etc. refers to the 1st, 2nd, 3rd, etc. capture group (a regex between parentheses)
  - **NB:** Since parentheses have a double function in Regex: grouping and capture groups, if we do not want to save a group in the register, we put the operator ?: after the open parenthesis i.e. (?:regex)

Words – to be continued

Let us define :

- **Types**: the number of distinct words in corpus = $|V|$
  - If V = set of words in the corpus, then $|V|$ is the size of the corpus
- **Tokens**: Total number of words = N
- **Disfluencies: fragments & fillers or filled pauses**
  - Example: a sentence from a phone conversation is of the form: *I do uh main- mainly business data processing*
    - Fragment = main-
    - Filler = uh
- **Wordform**: the full inflected or derived form of the word
  - Example: bag and bags are 2 distinct wordforms
- **Stem**: a word after stripping its suffixes
  - "Relational" → after stemming it becomes "relate"
- **lemma** : a set of lexical forms having the same stem
  - "Are", "am", "was", "were", etc. have the same lemma "be"

Corpora

Motivation: Why was the corpus collected, by whom, and who funded it?

Situation: When and in what situation was the text written/spoken? For example, was there a task? Was the language originally spoken conversation, edited text, social media communication, monologue vs. dialogue?

Language variety: What language (including dialect/region) was the corpus in?

Speaker demographics: What was, e.g., age or gender of the authors of the text?

Collection process: How big is the data? If it is a subsample how was it sampled? Was the data collected with consent? How was the data pre-processed, and what metadata is available?

Annotation process: What are the annotations, what are the demographics of the annotators, how were they trained, how was the data annotated? Distribution: Are there copyright or other intellectual property restrictions?

Text Normalization

1. Sentence Segmentation
2. Tokenizing words
3. Normalizing word formats
4. Segmenting sentences

Sentence Segmentation/Tokenization

It is the task to segment a text into sentences.

Rules for segmenting are punctuation:

- Periods: ambiguous
    - Confusion with periods occuring in words: e.g. Ph.D., Mrs., Mr., 2.99$, etc.
- question marks, and exclamation points: unambiguous

To alleviate this setback:

- Abbreviation dictionary
- Rule-based/Machine Learning algorithm to decide first whether a period is part of a word or is a sentence-boundary marker

Word Tokenization

Tokenization: the task of segmenting running text into words

Some general rules implemented in tokenizers to identify sentence boundaries:

- Periods
  - Confusion with periods occuring in words: e.g. Ph.D., Mrs., Mr., 2.99$, etc.
- Commas
  - In english commas are used inside numbers every 3 digits: e.g. 299,999.99$
  - In french commas are used to represent decimals: 2,99 €
- Clictic contractions
  - Example: we're → we are, j'ai → je ai, etc.
- Other customized rules:
  - Tokenizing some expressions as a single word: e.g. Rock 'n' roll, New York, Val de Seine, etc.

Common Tokenization standard: The Penn Treebank tokenization standard - released by the Linguistic Data Consortium (LDC).

Word Tokenization (2)

In python, NLTK (Natural Language ToolKit): (https://www.nltk.org/api/nltk.tokenize.html)

from nltk.tokenize import word_tokenize

from nltk import regexp_tokenize

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Figure 2.12 A Python trace of regular expression tokenization in the NLTK Python-based natural language processing toolkit (Bird et al., 2009), commented for readability; the (?x) verbose flag tells Python to strip comments and whitespace. Figure from Chapter 3 of Bird et al. (2009).

```
>>> from nltk.tokenize import word_tokenize
>>> s = '''Good muffins cost $3.88\nin New York.  Please buy me
... two of them.\n\nThanks.'''
>>> word_tokenize(s)
['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York', '.',
'Please', 'buy', 'me', 'two', 'of', 'them', '.', 'Thanks', '.']
```

```
>>> from nltk.tokenize import wordpunct_tokenize
>>> wordpunct_tokenize(s)
['Good', 'muffins', 'cost', '$', '3', '.', '88', 'in', 'New', 'York', '.',
'Please', 'buy', 'me', 'two', 'of', 'them', '.', 'Thanks', '.']
```

Word Normalization, Lemmatization and Stemming

Word normalization: is the task of putting words/tokens in a standard format.

- Case Folding: to map all words to lowercase
  - Useful for tasks such as information retrieval or speech recognition, etc.
  - Useless for sentiment analysis, text classification, Information extraction, etc.
    - Example 'us' after case folding is it US (the country) or us (the pronoun)?
- Lemmatization: converting words back to their root
  - Am, are, is, was, were → verb to be => their shared lemma is 'be'
  - Stories → story
  - ...
- Stemming: consists of chopping off word-final affixes
  - Most widely used stemming algorithm: Porter (1980)
  - Some rules & errors:

ATIONAL	→ ATE	(e.g., relational → relate)
ING	→ ϵ	if stem contains vowel (e.g., motoring → motor)
SSES	→ SS	(e.g., grasses → grass)

Errors of Commission		Errors of Omission	
organization	organ	European	Europe
doing	doe	analysis	analyzes
numerical	numerous	noise	noisy
policy	police	sparse	sparsity

References

- *Jurafsky, Daniel, and James H. Martin. "Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition."*

"There is no real ending. It's just the place where you stop the story."
Frank Herbert