Faculty of Computer Science, Dalhousie University *9-Oct-2025*

CSCI 4152/6509 — Natural Language Processing

Lecture 5: N-grams and Morphology

Location: Studley LSC-Psychology P5260 Instructor: Vlado Keselj
Time: 14:35 – 15:55

Previous Lecture

- Introduction to text processing with Perl
- Regular expressions in Perl
  - Use of special variables
  - Backreferences, shortest match

Problem example: (repeated) Consider the regular expressions `/(a+(b+))(c+(d+))\4/` and `/(a+(b+))(c+(d+))\3/`. Write some examples of strings matched by these regular expressions. Characterize all string matched by these regular expressions.

For example, the regular expression:

```
/(a+(b+))(c+(d+))\4/
 1  2    3  4
```

will match any string of the form $\mathtt{a}^n\mathtt{b}^m\mathtt{c}^k\mathtt{d}^l\mathtt{d}^l$. The numbers below the expression show how we count the opening parentheses to decide which one to match in the back reference. As a note, the superscripts n, m, k, and l denote repetition of the corresponding letter for that number of times. For a comparison, the following regular expression:

```
/(a+(b+))(c+(d+))\3/
 1  2    3  4
```

would match any string of the form $\mathtt{a}^n\mathtt{b}^m\mathtt{c}^k\mathtt{d}^l\mathtt{c}^k\mathtt{d}^l$.

Shortest Match

- default matching: left-most, longest match
- e.g., consider `/\d+/`
- Shortest match is sometimes preferred
  - e.g., consider: `/<div>.*<\/div>/` or `/<[^>]*>/` vs. `/<.*>/`
  - and: `/<div>.*?<\/div>/` and `/<.*?>/`
- Shortest match iterators:
  `*?` `+?` `??` `{n}?` `{n,m}?`

Regular Expression Substitutions

- syntax: `s/re/sub/options`
- Some substitution options
  `c` – do not reset search position after `/g` fail
  `e` – evaluate replacement as expression
  `g` – replace globally (all occurrences)
  `i` – case-insensitive pattern matching
  `m` – treat string as multiple lines

o – compile pattern only once
s – treat string as a single line
x – use extended regular expressions

5.3 Text Processing Example: Counting Letters

Text Processing Example

- Let us consider some simple text processing examples
- Use of Regular Expressions
- Convenient string manipulation
- Associative arrays
- Example: Counting Letters

Experiments on "Tom Sawyer"

- File: `TomSawyer.txt`:

```
                        The Adventures of Tom Sawyer

                                    by

                    Mark Twain (Samuel Langhorne Clemens)

    Preface
    MOST of the adventures recorded in this book really occurred; one or two
    were experiences of my own, the rest those of boys who were schoolmates
    of mine. Huck Finn is drawn from life; Tom Sawyer also, but not from an
    individual -- he is a combination of the characteristics of three boys
    whom I knew, and therefore belongs to the composite order of
    architecture.

    The odd superstitions touched upon were all prevalent among children and
    slaves in the West at the period of this story -- that is to say, thirty
    or forty years ago.

    Although my book is intended mainly for the entertainment of boys and
    girls, I hope it will not be shunned by men and women on that account,
    for part of my plan has been to try to pleasantly remind adults of what
    they once were themselves, and of how they felt and thought and talked,
    and what queer enterprises they sometimes engaged in.

    ...
```

Letter Count Total

```
#!/usr/bin/perl
# Letter count total

my $lc = 0;
```

```
while (<>) {
  while (/[a-zA-Z]/) { ++$lc; $_ = $'; }
}
print "$lc\n";

# ./letter-count-total.pl TomSawyer.txt
# 296605
```

Letter Frequencies

```
#!/usr/bin/perl
# Letter frequencies

while (<>) {
  while (/[a-zA-Z]/) {
    my $l = $&; $_ = $';
    $f{$l} += 1;
  }
}

for (keys %f) { print "$_ $f{$_}\n" }
```

Letter Frequencies Output

```
./letter-frequency.pl TomSawyer.txt
S 606
a 22969
T 1899
N 324
K 24
d 14670
Y 214
E 158
j 381
y 6531
u 8901
...
```

Letter Frequencies Modification

```
#!/usr/bin/perl
# Letter frequencies (2)

while (<>) {
  while (/[a-zA-Z]/) {
    my $l = $&; $_ = $';
    $f{lc $l} += 1;
  }
}

for (sort keys %f) { print "$_ $f{$_}\n" }
```

New Output

```
./letter-frequency2.pl TomSawyer.txt
a 23528
b 4969
c 6517
d 14879
e 35697
f 6027
g 6615
h 19608
i 18849
j 639
k 3030
...
```

We will look now at an implementation where letters and their frequencies are sorted by the frequency, from the highest-frequency letter to the lowest. We will also produce frequencies both as letter counts, and as normalized frequencies; i.e., as proportional frequencies of the letters out of 1.

Letter Frequencies Modification (3)

```
#!/usr/bin/perl
# Letter frequencies (3)

while (<>) {
  while (/[a-zA-Z]/) {
    my $l = $&; $_ = $';
    $f{lc $l} += 1; $tot ++;
  }
}

for (sort { $f{$b} <=> $f{$a} } keys %f) {
  print sprintf("%6d %.4lf %s\n",
              $f{$_}, $f{$_}/$tot, $_); }
```

Output 3

```
 35697 0.1204 e
 28897 0.0974 t
 23528 0.0793 a
 23264 0.0784 o
 20200 0.0681 n
 19608 0.0661 h
 18849 0.0635 i
 17760 0.0599 s
 15297 0.0516 r
 14879 0.0502 d
 12163 0.0410 l
  8959 0.0302 u
...
```

6 Elements of Morphology

- Reading: Section 3.1 in the textbook, "Survey of (Mostly) English Morphology"
- *morphemes* — smallest meaning-bearing units
- *stems* and *affixes*; stems provide the "main" meaning, while affixes act as modifiers
- affixes: prefix, suffix, infix, or circumfix
- cliticization — clitics appear as parts of a word, but syntactically they act as words (e.g., 'm, 're, 's)
- tokenization, stemming (Porter stemmer), lemmatization

The *morphemes* are the smallest meaning-bearing parts of a word. For example, the word *cats* contains two morphemes *cat* and *s*, the word *unbelievably* contains the four morphemes *un*, *believ*, *ab*, and *ly*, and the word *unmorpholinguistically* contains the six morphemes *un*, *morpho*, *ling*, *uist*, *ical*, and *ly*. It could be sometimes debatable what is the proper way of breaking a word into morphemes, but not having a clear correct answer is not uncommon in analysis of natural languages.

- suffix example: eat*s*; prefix example: *un*buckle; circumfix example from German: sagen (to say) and *ge*esag*t* (said, past participle); infix example from Tagalog (Philipine language): hingi (borrow) and h*um*ingi
- stacking multiple affixes is possible: unbelievably = un-believe-able-y
- English typically allows up to 4 affixes, but some languages allow up to 10 affixes, such as Turkish. Such languages are are called *agglutinative* languages.
- *cliticization* is considered to be a morphological process
- Clitics appear as orthographic or phonological parts of the words, but syntactically they act as words.
- Clitic examples: 'm in I'm, 're in we're, possessive 's

Tokenization

- Text processing in which plain text is broken into words or *tokens*
- Tokens include non-word units, such as numbers and punctuation
- Tokenization may normalize words by making them lower-case or similar
- Usually simple, but prone to ambiguities, as most of the other NLP tasks

Tokenization is text processing in which the plain text is broken into words. It may not be a simple process, depending on the type of text and kind of tokens that we want to recognize.

Stemming is the type of word processing in which a word is mapped into its *stem*, which is a part of the word that represents the main meaning of the word. For example, *foxes* is mapped to the stem *fox*, or the word *semantically* is mapped to the stem *semanti*.

It is used in Information Retrieval due to the property that if two words have the same stem, they are typically semantically very related. Hence, if words in documents and queries are replaced by their stems, the resulting indices are smaller, and words in a query can be easily matched with their morphological variations.

Lemmatization is a word processing method in which a *surface word form*, i.e., the word form as it appears in text, is mapped to its *lemma*, i.e., the canonical form as it appears in a dictionary. For example, the word *working* would be mapped into the verb *work*, or the word *semantically* would be mapped to the lemma *semantics.*

6.1 Morphological Processes

A *morphological process* is a word transformation that happens as a regular language transformation. There are tree main morphological processes in English:

1. inflection,
2. derivation, and
3. compounding.

1. Inflection: is a transformation that transforms a word from one lexical class into another related word in the same class. The transformation is performed by adding or changing a suffix or prefix. It is highly regular transformation. Some inflection examples are: dog → dogs, work → works, work → working, and work → worked.

We will discuss more the concept of *lexical class* or *part of speech* class later, but for now you are probably familiar with the following lexical classes (or types of words): nouns, verbs, adjectives, adverbs, and maybe some other.

Inflection is so regular transformation that usually we do not find inflected variations of a word in a dictionary. It is assumed that a reader of the dictionary will be able to derive these variations by herself. Similarly, we can frequently program inflection in a computer application rather than storing different variations of the word.

2. Derivation: is a transformation that transforms a word from one lexical class into a related word in a different class. Similarly to inflection, it is performed by adding or changing a suffix or prefix. There is also some regularity, but it is less regular than inflection. For example, a derivation is *wide (adjective)* → *widely (adverb)*, but a similar transformation *old* → *oldly* is not valid. Some other examples are: accept (verb) → acceptable (adjective), acceptable (adjective) → acceptably (adverb), and teach (verb) → teacher (noun).

There are exceptions where a derivation is used to transform a word in a lexical class to another word in the same class but it is a significantly a different word. For example, the transformation of the adjective *red* to *redish* is considered a derivation, rather than an inflection.

Since derivation is not as regular transformation as inflection, derived variations of a word are usually stored in a dictionary, and in a computer application we may want to store them in a lexicon, i.e., a word database, in many cases.

Below you can find a table with some more derivation examples:

Derivation type	Suffix	Example		
noun-to-verb	*-fy*	glory	→	glorify
noun-to-adjective	*-al*	tide	→	tidal
verb-to-noun (agent)	*-er*	teach	→	teacher
verb-to-noun (abstract)	*-ance*	delivery	→	deliverance
verb-to-adjective	*-able*	accept	→	acceptable
adjective-to-noun	*-ness*	slow	→	slowness
adjective-to-verb	*-ise*	modern	→	modernise (Brit.)
adjective-to-verb	*-ize*	modern	→	modernize (U.S.)
adjective-to-adjective	*-ish*	red	→	reddish
adjective-to-adverb	*-ly*	wide	→	widely

3. Compounding: is a transformation where two or more words are combined, usually by concatenation, to create a new word. Some examples are: news + group → newsgroup, down + market → downmarket, over + take → overtake, play + ground → playground, and lady + bug → ladybug.

7 Characters, Words, and N-grams

Slide notes:

Characters, Words, N-grams

- We saw some experiments with counting characters
- Let us look at Counting Words
- N-grams and Counting N-grams

7.1 Counting Words and Zipf's Law

- We looked at code for counting letters, words, and sentences
- We can look again at counting words; e.g., in "Tom Sawyer":
- We can observe: Zipf's law (1929): $r \times f \approx$ const.

Word	Freq (f)	Rank (r)
the	3331	1
and	2971	2
a	1776	3
to	1725	4
of	1440	5
was	1161	6
it	1030	7
I	1016	8
that	959	9
he	924	10
in	906	11
's	834	12
you	780	13
his	772	14
Tom	763	15
't	654	16
⋮	⋮	

One of the basic tasks that we can do using stream-oriented processing of language is to collect statistical values on letters, words, sentences, or similar tokens. We saw previously the code for finding frequency of different letters, and these data can be useful for example for computer identification of a natural language. We can do similar counting but this time of word frequencies. The table above shows the frequencies of words in the novel "Tom Sawyer" by Mark Twain.

Zipf's law is an observation that the product of rank and frequency of the words in a text is "quite constant," if we can use that term. For example, we can test this "law" on the words in the "Tom Sawyer" novel using the following code:

Counting Words

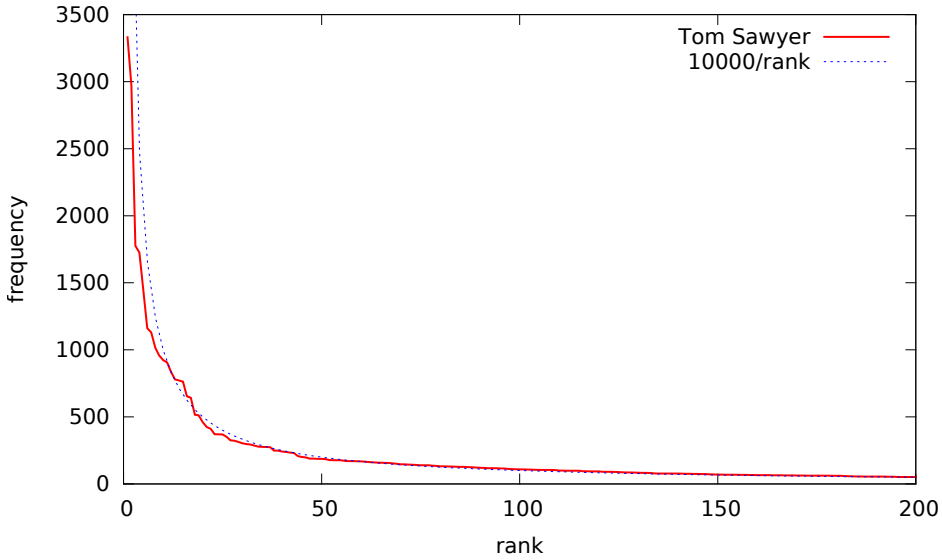
```
#!/usr/bin/perl
# word-frequency.pl

while (<>) {
  while (/'?[a-zA-Z]+/g) { $f{$&}++; $tot++; }
}

print "rank   f   f(norm) word       r*f\n".
      ('-'x35)."\n";
for (sort { $f{$b} <=> $f{$a} } keys %f) {
  print sprintf("%3d. %4d %lf %-8s %5d\n",
              ++$rank, $f{$_}, $f{$_}/$tot, $_,
              $rank*$f{$_});
}
```

Program Output (Zipf's Law)

```
rank   f    word       r*f       18.  516   for       9288
---------- -----------------     19.  511   had       9709
  1. 3331   the       3331       20.  460   they      9200
  2. 2971   and       5942       21.  425   him       8925
```

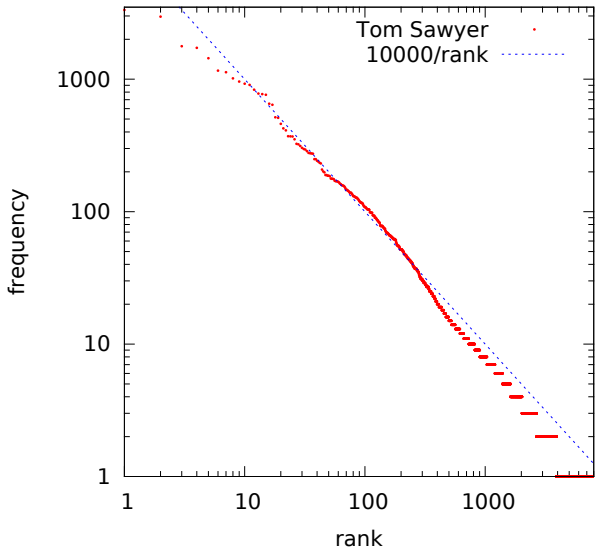
3.	1776	a	5328	22.	411	but	9042
4.	1725	to	6900	23.	371	on	8533
5.	1440	of	7200	24.	370	The	8880
6.	1161	was	6966	25.	369	as	9225
7.	1130	it	7910	26.	352	said	9152
8.	1016	I	8128	27.	325	He	8775
9.	959	that	8631	28.	322	at	9016
10.	924	he	9240	29.	313	she	9077
11.	906	in	9966	30.	303	up	9090
12.	834	's	10008	31.	297	so	9207
13.	780	you	10140	32.	294	be	9408
14.	772	his	10808	33.	286	all	9438
15.	763	Tom	11445	34.	278	her	9452
16.	654	't	10464	35.	276	out	9660
17.	642	with	10914	36.	275	not	9900

We can present this data in a graphical form and compare it with the function $f = 10000/r$ to demonstrate the



Zipf's law:

If we apply a logarithm on both sides of the Zipf's formula we get the formula $\log r + \log f \approx$ const., which means that the Zipf's law implies that the rank-frequency graph using log scales of x and y axis should be close to a straight line, descending under an angle of 45 degrees. The following graph illustrates this:



7.2 Counting N-grams

Given a sequence of tokens $T = (t_1, t_2, t_3, \ldots, t_k)$ an n-gram is an arbitrary subsequence of n such tokens, such as $(t_1, t_2, \ldots, t_n)$, or $(t_2, t_3, \ldots, t_{n+1})$, and so on. Given a textual document, it could be broken into a list of character or list of words, if we assume that a character is a token, or that a word is a token. In those cases, we look at character n-grams or word n-grams, respectively. We typically want to collect all n-grams from a text, and a way to visualize this is to imagine a sliding window over the text.

Character N-grams

- Consider the text:
  `The Adventures of Tom Sawyer`
- Character n-grams = substring of length n
- $n = 1 \Rightarrow$ *unigrams*: `T`, `h`, `e`, `_` (space), `A`, `d`, `v`, ...
- $n = 2 \Rightarrow$ *bigrams*: `Th`, `he`, `e_`, `_A`, `Ad`, `dv`, `ve`, ...
- $n = 3 \Rightarrow$ *trigrams*: `The`, `he_`, `e_A`, `_Ad`, `Adv`, `dve`, ...
- and so on; Similarly, we can have word n-grams, such as ($n = 3$): `The Adventures of`, `Adventures of Tom`, `of Tom Sawyer` ...
- or normalized into lowercase

For example, if we take another look at the Tom Sawyer novel:

```
The Adventures of Tom Sawyer

by

Mark Twain (Samuel Langhorne Clemens)

Preface
MOST of the adventures recorded in this book really occurred; one or two
were experiences of my own, the rest those of boys who were schoolmates
of mine. Huck Finn is drawn from life; Tom Sawyer also, but not from an
individual -- he is a combination of the characteristics of three boys
whom I knew, and therefore belongs to the composite order of
```

```
architecture.

The odd superstitions touched upon were all prevalent among children and
slaves in the West at the period of this story -- that is to say, thirty
or forty years ago.

Although my book is intended mainly for the entertainment of boys and
girls, I hope it will not be shunned by men and women on that account,
for part of my plan has been to try to pleasantly remind adults of what
they once were themselves, and of how they felt and thought and talked,
and what queer enterprises they sometimes engaged in.
...
```

Word and Character N-grams ($n = 3$)

Word tri-grams	Character tri-grams
the adventures of	T h e _ o f
adventures of tom	h e _ o f _
of tom sawyer	e _ A f _ T
tom sawyer by	_ A d _ T o
sawyer by mark	A d v T o m
by mark twain	d v e o m _
mark twain samuel	v e n m _ S
twain samuel langhorne	e n t _ S a
samuel langhorne clemens	n t u S a w
langhorne clemens preface	t u r a w y
clemens preface most	u r e w y e
preface most of	r e s y e r
most of the	e s _ e r _
...	s _ o ...

A Program to Extract Word N-grams

The following Perl program `word-ngrams.pl` lists all word ngrams extracted from the standard input. We set variable `$n` to 3 as we want word 3-grams to be extracted. The first while-loop reads input line by line, and in the second while-loop we match string that we assume would be words. We choose any sequence of letters to be a word, and possibly starting with an apostrophe (`'`). As we will see later, this will recognize usual words, but it will also break complex words like `I'm`, `you're`, or `man's` into words `I` and `'m`, `you` and `'re`, and `man` and `'s`.

```
#!/usr/bin/perl
# word-ngrams.pl

$n = 3;

while (<>) {
  while (/'?[a-zA-Z]+/g) {
     push @ng, lc($&); shift @ng if scalar(@ng) > $n;
     print "@ng\n" if scalar(@ng) == $n;
  }
}

# Output of: ./word-ngrams.pl TomSawyer.txt
```

```
# the adventures of
# adventures of tom
# ...
```

Some Perl List Operators

- `push @a, 1, 2, 3;` — adding elements at the end
- `pop @a;` — removing elements from the end
- `shift @a;` — removing elements from the start
- `unshift @a, 1, 2, 3;` — adding elements at the start
- `scalar(@a)` — number of elements in the array
- `$#a` — last index of an array, by default `$#a = scalar(@a) - 1`
- To be more precise, this is always true: `scalar(@a) == $#a - $[ + 1`
- `$[` (by default 0) is the index of first element of an array
- Arrays are dynamic: examples: `$a[5] = 1`, `$#a = 5`, `$#a = -1`

Since the first element of an array `@a` is `$a[0]` and the last element is `$a[$#a]` the number of elements is obviously `$#a+1$`. Another way to obtain the number of elements of an array is `scalar(@a)`. The Perl function `scalar` enforces a scalar context on an expression, and in a scalar context an array is interpreted just a number representing its length.

Perl arrays are dynamic, they expand and also can shrink easily. For example, after the command '`$a[5] = 1`' the array `@a` will be expanded if needed to at least six elements. The command '`$#a = 5`' sets array `@a` to exactly six elements. Similarly, '`$#a = -1`' erases an array by reducing it to zero elements, so it is equivalent to the command '`@a = ();`'.

Extracting Character N-grams (attempt 1)

```
#!/usr/bin/perl
# char-ngrams1.pl - first attempt

$n = 3;

while (<>) {
  while (/\S/g) {
     push @ng, $&; shift @ng if scalar(@ng) > $n;
     print "@ng\n" if scalar(@ng) == $n;
  }
}

# Output of: ./char-ngrams1.pl TomSawyer.txt
# T h e   A d v   e n t
# h e A   d v e   n t u
# e A d   v e n   ...
```

Extracting Character N-grams (attempt 2)

```
#!/usr/bin/perl
# char-ngrams2.pl - second attempt

$n = 3;

while (<>) {
  while (/\S|\s+/g) {
```

```
    my $token = $&;
    if ($token =~ /^\s+$/) { $token = '_' }
    push @ng, $token;
    shift @ng if scalar(@ng) > $n;
    print "@ng\n" if scalar(@ng) == $n;
  }
}

# Output of: ./char-ngrams2.pl TomSawyer.txt
# _ T h   f _ T   _ _ _
# T h e   _ T o   _ _ M
# h e _   T o m   _ M a
# e _ A   o m _   ...
# _ A d   m _ S      This may be what we want, but
# A d v   _ S a      probably not.
# d v e   S a w
# v e n   a w y
# e n t   w y e
# n t u   y e r
# t u r   e r _
# u r e   r _ _
# r e s   _ _ _
# e s _   _ _ b
# s _ o   _ b y
# _ o f   b y _
# o f _   y _ _
```

This output may be what we want, but probably not. Since we already reduced repeated whitespace characters to one underscore (‘_’), we probably want to treat the new line in the same way.