Strings, distances, text representations

Anton Alekseev, Steklov Mathematical Institute in St Petersburg
NRU ITMO, St Petersburg, 2018
anton.m.alexeyev+itmo@gmail.com

Motivation

The voice from the bloody enterprise:
If you can avoid ML, please do avoid it!

Standard algorithms on strings, automata, etc. are unsung NLP and Data Science heroes

What is also important: they are widely used for **data preparation and handcrafted features development**

String distances/metrics: why discuss this?

Tasks examples from real life:

1. Given a list of companies names extracted from texts automatically, put different spellings of the same organization into one cluster without any other external companies database available.

2. People often make orthographic errors and misprints on the web. Given gold standard dictionary and errors stats, we can easily program a simple but powerful approach to spelling check/correction using only string distances and basic statistics.

3. More ideas?

String metrics

We believe **there are no 'shifts'** between strings:
Hamming distance = counting 'replacements'

$$d_{ij} = \sum_{k=1}^{p} |x_{ik} - x_{jk}|$$

Invented for counting the number of positional mismatches in binary codes.

In our case -- characters.

R	i	c	h	a	r	d
r	i	c	h	e	r	d

H	a	m	m	i	n	g	
H	a	m	m	m	i	n	g

String metrics

Jaro similarity **(1989)**

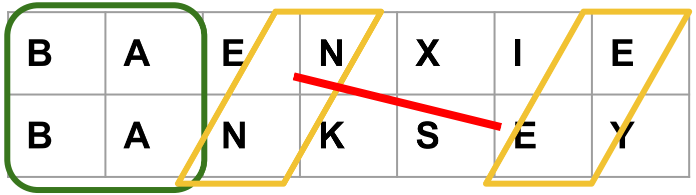
$$d_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{otherwise} \end{cases}$$

m - a number of *matching* characters. *matching* = positions differ by not more than

$$\left\lfloor \frac{\max(|s_1|, |s_2|)}{2} \right\rfloor - 1$$

Here equals to 2

t - half the number of all matching symbols, where the letters are in the wrong order



m = 4
t = 0
d = 0.71

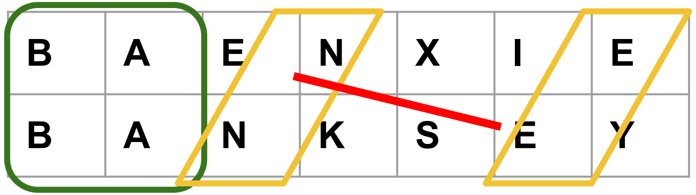
String metrics

Jaro-Winkler distance **(1990)**

$$d_w = d_j + (\ell p(1 - d_j)),$$

l - length of the prefices that match exactly
(a maximum of 4)
p - scaling coefficient
(from 0 to 0.25); rule of thumb -- approx. **0.1**

Was used for approximate last names matching for the purposes of the US population census



m **= 0.71**

d_w = d_j + 2 * 0.1 * (1 - d_j) = **0.768**

String metrics

Shifts are possible, though not numerous:
Levenshtein distance

The minimum number of operations required to transform one string into the other: **ins**ertions, **del**etions, **sub**stitutions.

To compute Levenshtein distance one has to solve a dynamic programming problem

p	**o**	**n**	**e**	**j**	**e**
o	**l**	**e**	**j**	**e**	**k**

poneje - DEL
oneje - INS
oneje**k** - SUB
o**l**ejek

d = 3

Levenshtein distance: how to compute

Wagner–Fischer algorithm

Solving the task for smaller prefices and then reusing the results for larger ones until we get the solution for the original strings.

Initially, all empty strings have distance 0
d(0,0) = 0

		B	A	R	T	O	L	D
	0							
B								
A								
R								
O								
N								

Levenshtein distance: how to compute

Zero for empty strings
d(0,0) = 0

Distance between empty one and a non-empty one
d(0,j) = j, d(i,0) = i

		B	A	R	T	O	L	D
	0	1	2	3	4	5	6	7
B	1							
A	2							
R	3							
O	4							
N	5							

Levenshtein distance: how to compute

Empty strings are equal
d(0,0) = 0

Between empty and non-empty strings
d(0,j) = j, d(i,0) = i

General case d(i, j)
if last letters match
= d(i-1, j-1)
If they don't - **one +** the minimum of
= d(i -1, j) - DEL (letter removal)
= d(i, j - 1) - INS (letter insertion)
= d(i-1, j-1) - SUB (letter substitution)

		B	**A**	**R**	**T**	**O**	**L**	**D**
	0	1	2	3	4	5	6	7
B	1	**0**	1	2	3	4	5	6
A	2	1	**0**	1	2	3	4	5
R	3	2	1	**0**	1	2	3	4
O	4	3	2	1	1	**1**	2	3
N	5	4	3	2	2	2	2	**3**

Modifications and applications

- **Damerau-Levenshtein** distance**:** adding the possibility to swap neighbouring characters
  (Based on Damerau's idea that most typos are of wrong-order-of-letters type)

- One could introduce different penalties for operations DEL, INS, SUP and sum them up instead of 1-s when computing Levenshtein distance

String metrics

If ‘modifications’ to the text are numerous but it still makes sense to try to match it, we should try **Longest Common Subsequence (LCS)**

O	O	O	_	**A**	**R**	**G**	**O**	_	_	_
A	_	**R**	_	**G**	_	**O**	_	L	L	C

LCS = 4

LCS: how to compute

Similar story

d(0, 0) = 0

however

d(0, j) = d(i, 0) = 0

General case:

if last letters match

d(i, j) = d(i -1, j - 1) + 1

If they don't, we take maximum of

d(i - 1, j) и d(i, j - 1)

		B	**_**	**A**	**T**	**M**	**E**	**N**
	0	0	0	0	0	0	0	0
R	0	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1	1
M	0	0	0	1	1	2	2	2
E	0	0	0	1	1	2	3	3
N	0	0	0	1	1	2	3	4

The Family

All string metrics discussed earlier are called **edit distances**, they employ**:** **insertion, substitution, transpositions and deletions.**

Each is best for certain problems, however sometimes they are unsuitable for computationally intensive tasks due to being too slow, e.g. for ad-hoc similar strings search.

String metrics

Bag-of-ngrams is a weak attempt to take word order into account.

Jaccard distance for character n-grams

(any other set distance may also be suitable)

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

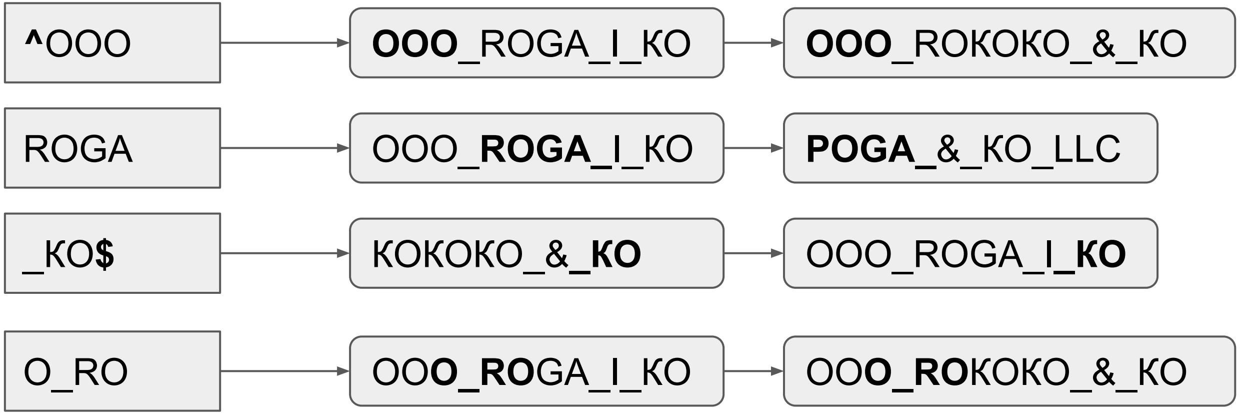
O	**O**	**O**	**_**	**R**	**O**	**G**	**A**	**_**	**I**	**_**	**K**	**O**
R	**O**	**G**	**A**	**_**	**&**	**_**	**K**	**O**	**_**	**L**	**L**	**C**

If you don’t count duplicates (though it may be useful)
For unigrams: 6 / (7 + 9 - 6)
For bigrams: ?
For trigrams: 4 / (11 + 11 - 4)

BTW: N-gram indices

We can construct the inverted index to be able to retrieve **strings with maximum number of n-grams** common with the query!

Then this search results set can be ranked by a more complex and computationally hard metric (e.g. Levenshtein distance).



Implementations

Python

- nltk.metrics.distance
- python-Levenshtein
- Jellyfish! (+ has **soundex**!)
- ...

https://cdn.dribbble.com/users/53712/screenshots/964040/untitled-1.gif

+ Lucene (Java) has NgramIndex

(I suggest you do not reinvent the wheel for production code!)

Do we have any time?

Notes on standard text representation approaches

Method #1, Bag-of-words: one hot
~ one-hot-encoding / dummy coding: many interpretable features
“Hush now, baby, baby, don't you cry”

Bag-of-words: word counts (sklearn: CountVectorizer)
counts or relative frequencies instead of one-hot values

hush	now	baby	wall	do	not	you	oh	cry
1	1	2	0	1	1	1	0	1

Bag-of-words: weird numbers (sklearn: TfIdfVectorizer)
TF-IDF or other estimates of terms importance

Notes on standard text representation approaches

By ‘forgetting’ about word order we lose information, however, there is a simple way to at least try to take word order into account!

Bag-of-ngrams (sklearn vectorizers support this out-of-the-box, btw)
ngram = n terms in a row as a single term

“New York”
“New Deli”
“**not** cool”
“catch up with”

+ other reasons why word order has to be dealt with

BOW: specifics and takeaways

https://twitter.com/stanfordnlp/status/399551909595344896

tens/hundreds of thousands of sparse features; curse of dimensionality may be a problem:

1. have to filter terms and introduce penalties for the most frequent and rare ones;
   implemented in almost any toolbox, e.g. in sklearn;
   (including **stopwords** filtering: “useless/common words”)

2. should choose models working with large number of sparse features
   ~~one can’t simply solve all problems with *Random Forest!*~~

3. should always experiment with choosing N in Ngrams and weights for terms (one-hot/tfidf etc.)

BOW: specifics and takeaways

tens/hundreds of thousands of sparse features; curse of dimensionality may be a problem:

1. have to filter terms and introduce penalties for the most frequent and rare ones;
   implemented in almost any toolbox, e.g. in sklearn;
   (including **stopwords** filtering: “useless/common words”)

2. should choose models working with large number of sparse features
   ~~one can’t simply solve all problems with *Random Forest!*~~

3. should always experiment with choosing N in Ngrams and weights for terms (one-hot/tfidf etc.)

https://twitter.com/stanfordnlp/status/399551909595344896

When BoW may not be enough?

Ideas?

When BoW may not be enough?

- Small data
  - Zipf's law
  - Rich morphology => not too many training samples
  - ...what if we lemmatize? => sometimes we can't neglect morphology
- Short texts
  - same reasons
  - + intuitively: the larger the text the more good word predictors it has

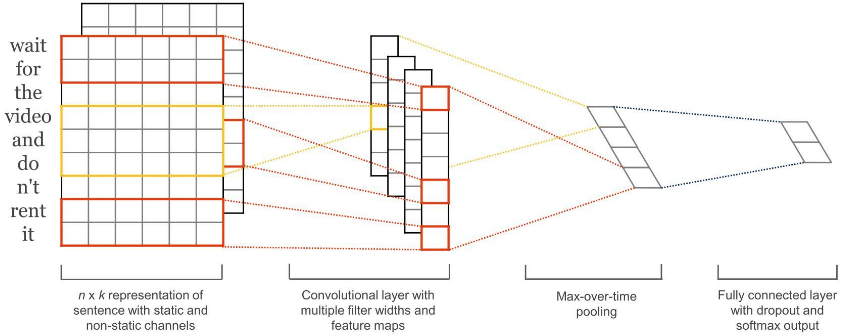
Trash bag // *Wikipedia*

https://upload.wikimedia.org/wikipedia/commons/thumb/c/c8/Garbage_bag.jpg/1200px-Garbage_bag.jpg

Notes on standard text representation approaches

Method #2 sum word vectors (e.g., word2vec) of all words in the texts
with weights proportional to importance weights (e.g. TF-IDF)

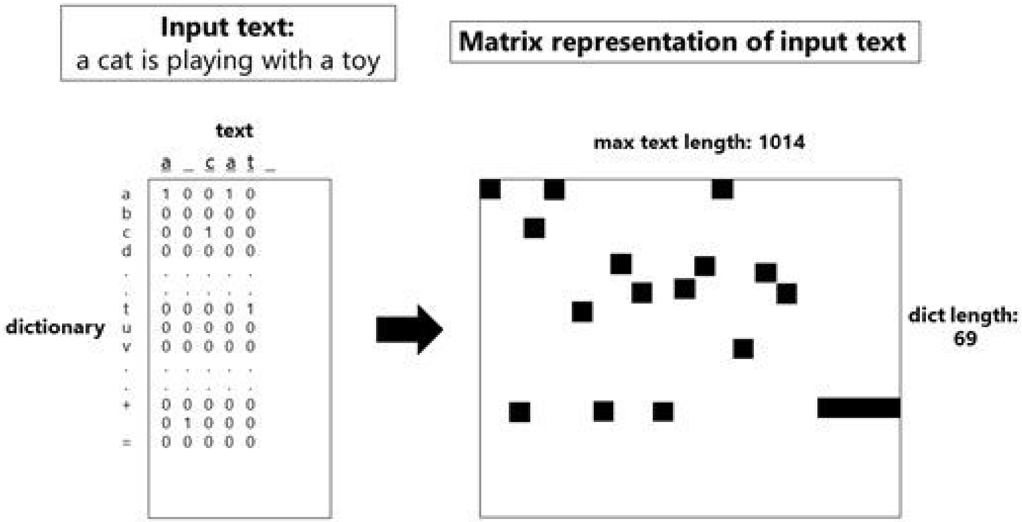
Method #3 concat word vectors (e.g., word2vec) of all words in the texts
into a matrix



What if we go beyond word level?

...that is, represent the text as a sequence of encoded characters (**Method #4**)

e.g. see: http://karpathy.github.io/2015/05/21/rnn-effectiveness/



https://blogs.technet.microsoft.com/machinelearning/2017/02/13/cloud-scale-text-classification-with-convolutional-neural-networks-on-microsoft-azure/

https://i.pinimg.com/originals/20/39/17/203917d3b4cd0fa531801d46a432d272.jpg

More ideas?

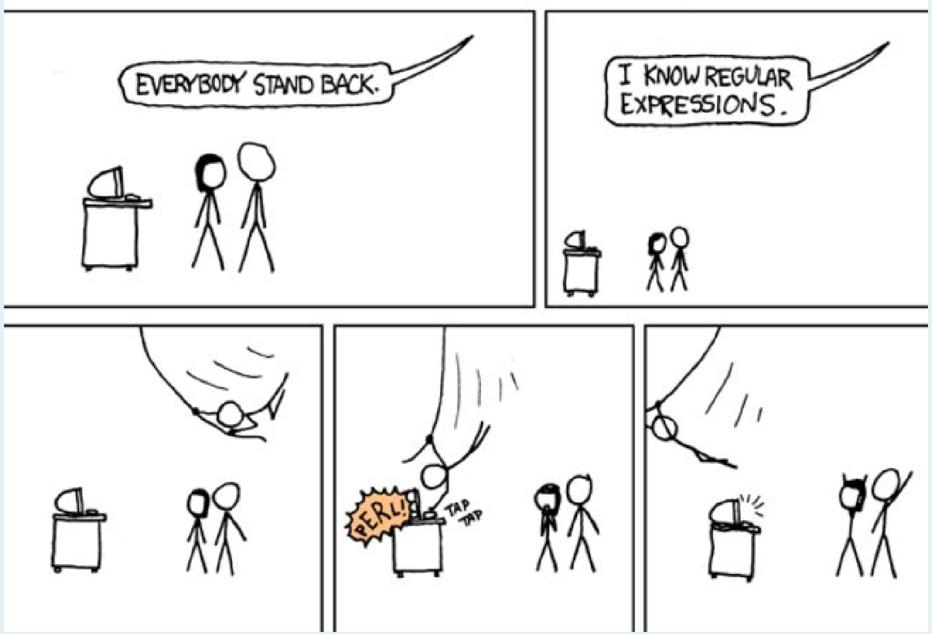
Strings, distances, text representations

Anton Alekseev, Steklov Mathematical Institute in St Petersburg
NRU ITMO, St Petersburg, 2018
anton.m.alexeyev+itmo@gmail.com

Extra topic: regular expressions

If we know something else about our strings

E.g. the substring it contains or its specific format:
phone number, email address, etc.



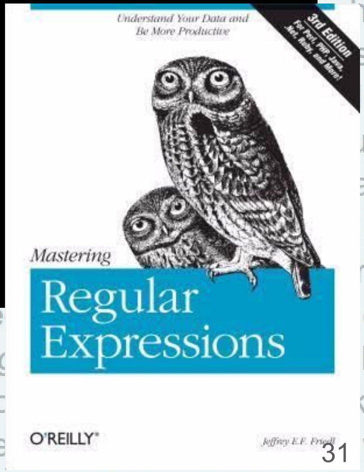
Be careful!

A weapon for a civilized age, however, once you master it, you want to use it everywhere, however

- not suitable for some tasks ([*don’t parse XML with regex*]()),

- requires elegance and support for using in production environment

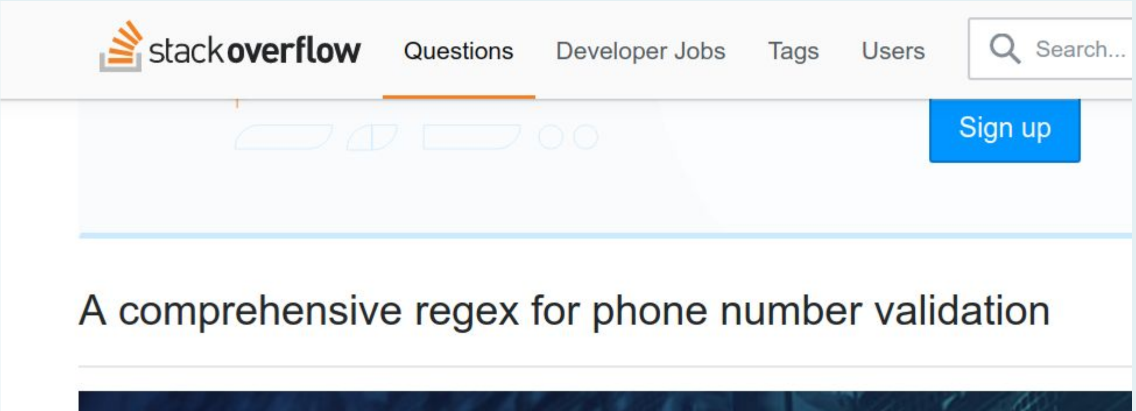
You can't parse [X]HTML with regex. Because HTML can't be parse
that can be used to correctly parse HTML. As I have answered in H
so many times before, the use of regex will not allow you to consum
are a tool that is insufficiently sophisticated to understand the const
HTML is not a regular language and hence cannot be parsed by reg
queries are not equipped to break down HTML into its meaningful p
gettin [redacted] used
of par [redacted] nguag
cannc [redacted] not p
expre [redacted] lar ex
weeps [redacted] bapp.
summ [redacted] regex
and ri [redacted] he fo
in the [redacted] h w
with re [redacted] s w
toil fo [redacted] ult
HTML [redacted] u o
in the [redacted] an
it is to [redacted] ch
all livi [redacted] rop
can anyone survive this scourge using re [redacted] do
dread torture and security holes *using reg* [redacted] HT
between this world and the dread realm c [redacted] ke
corrupt) a mere glimpse of the world of re [redacted] **wi**
programmer's consciousness into a world of ceaseless screaming,



RegEx in Everyday Life

Yet sometimes it is better to use regex as a simple solution for NLP tasks

- Named entities extraction
- Text classification
- grep (instead of using some information retrieval engine!)
- ...



RegEx: characters types

Setting a regex means setting a finite automata firing ‘success’ at certain strings

.	Any character but **\n**
\d	Digit
\D	Not a digit
\w	Letter, digit, _
\W	Not a letter or digit or _
\s	Whitespace char
\S	Not a whitespace char
\b	Word bound
\B	Not a word bound
^ $	The beginning and the end of the string

Each regex sets a **language**:

... - any 3-char strings
\d\d\d - any 3-digit ‘number’ (may start with 0)
921**\s-\s\d\d\d\s-\s** - phone numbers of certain format

But how do we use full stop as a full stop? **Escaping!**

Hello.\s - “Hello! ”, “Hello. ”, “Hellof ”
Hello\.\s - just “Hello. ”

Регулярные выражения: повторения и вариации

<table>
<tr><td>*</td><td>‘Kleene star’,
repetition of the previous character 0+ times</td></tr>
<tr><td>?</td><td>Zero or one characters</td></tr>
<tr><td>+</td><td>Repetition, at least one time</td></tr>
<tr><td>{2}</td><td>Repetition, two times</td></tr>
<tr><td>{1,3}</td><td>Repetition from 1 to 3 times</td></tr>
<tr><td>{2,}</td><td>Repetition more that 1 time</td></tr>
<tr><td>[A-Za-z0-9шыж]</td><td>Any character listed inbraces</td></tr>
<tr><td>[^xyz]</td><td>Neither</td></tr>
<tr><td>ма(ма|ть)</td><td>One of the groups separated with |</td></tr>
<tr><td>[whatever]*?</td><td>? after repetition - “greedy” search</td></tr>
</table>

Regular expressions: tips and tricks

- Reuse! If possible
- If in doubs -- google it + write tests
- Put some regex cheatsheets on the office’s wall
- Regex have dialects: POSIX, PCRE
  choose wisely!
- Always compile regular expressions that are to be later used multiple times (e.g. in a loop)!

- Regex are learnt only in practice, so consider taking some practical exercises. For example, this online course https://regexone.com/