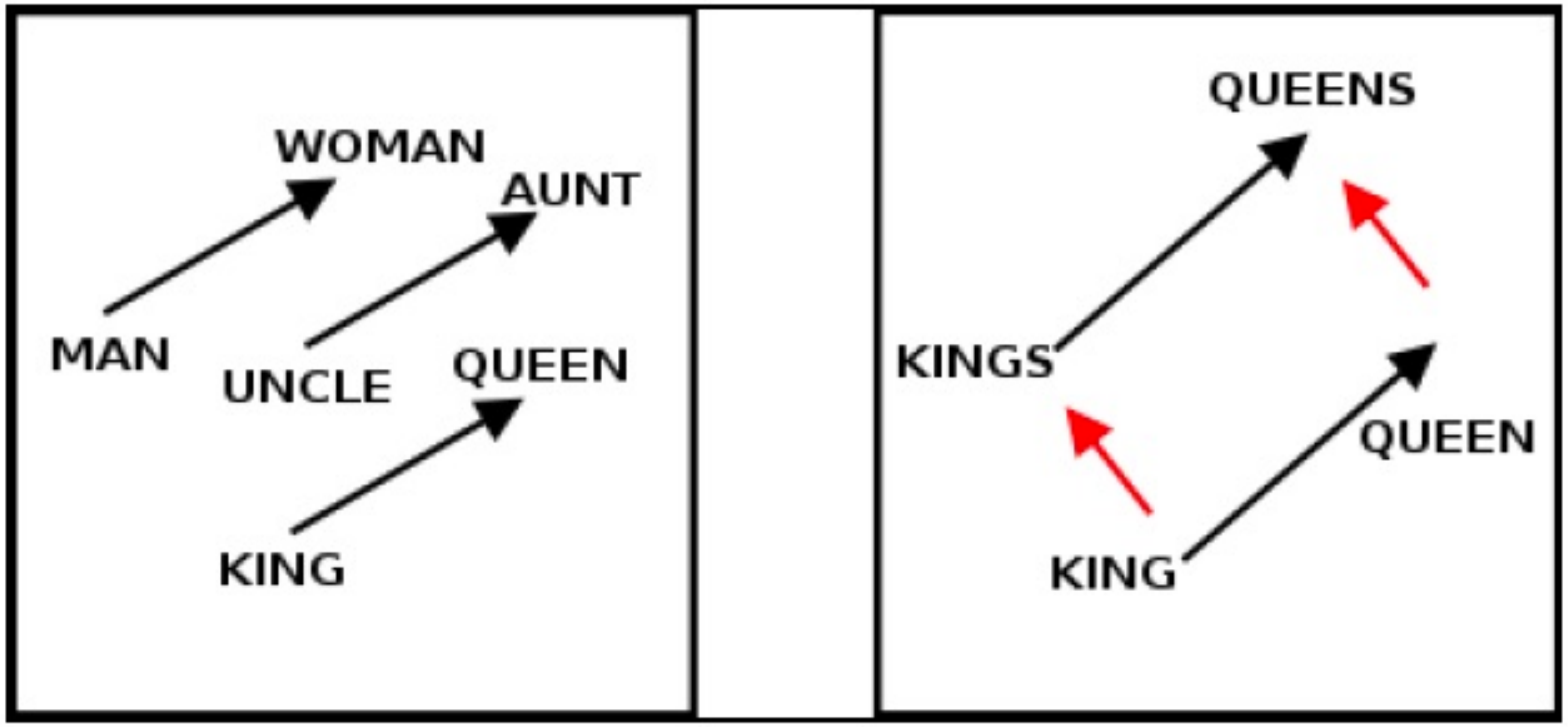
IR, distances, representations

Eugeny Malyutin



Ideas:

- Managers voice: **If you can avoid ML - please do avoid it!**

- Algorithms on strings, distances and etc are unsung DS heroes

- If you can’t avoid ML - use with caution:
  - Data preparation
  - Feature development

Tasks:

- Find the last name in the database. Last name is recorded «from voice».
  — Леха, поищи в нашей базе
  Адольфа **Швардсенеггера**,
  — **Шворцениггера**? Нет такого!

- 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.

- More ideas?

Hamming distance:

- We believe **there are no ‘shifts’** between strings:
  Hamming distance = counting ‘replacements’

$$d_{i,j} = \sum_{n=1}^{l} (|x_i - y_i|)$$

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

- Formally « » is character

R	I	C	H
B	I	C	H

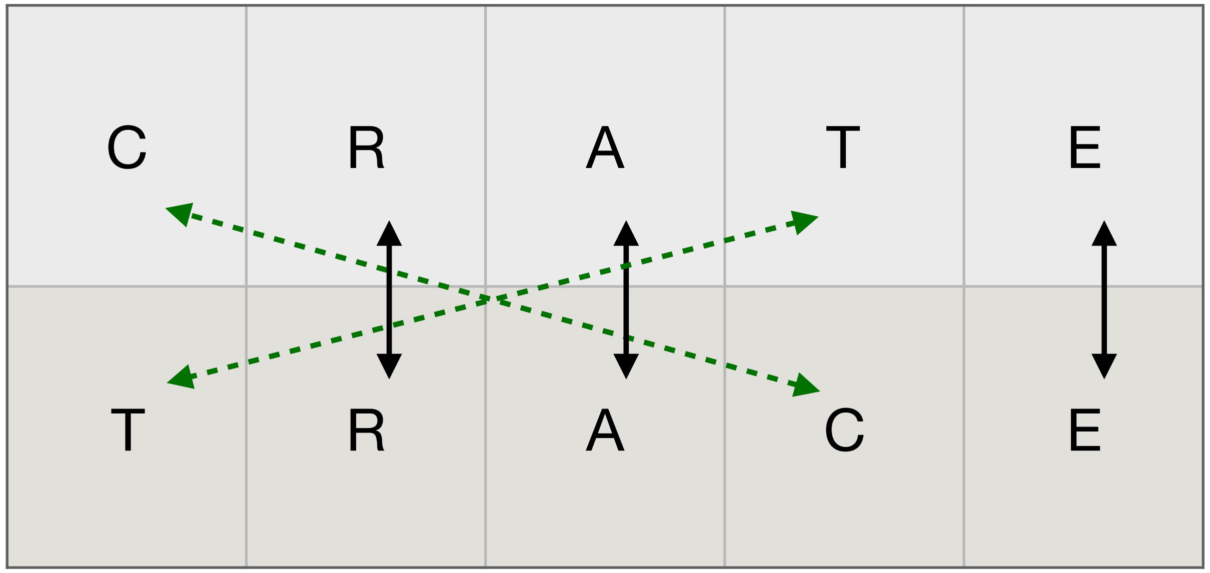
H	A	M	**M**	**I**	**N**	**G**
H	A	M	**I**	**N**	**G**	

Jaro similarity:

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

Where:

- is the length of the string ;
- **m** is the number of ***matching*** *characters*
- **t** is half the number of *transpositions*



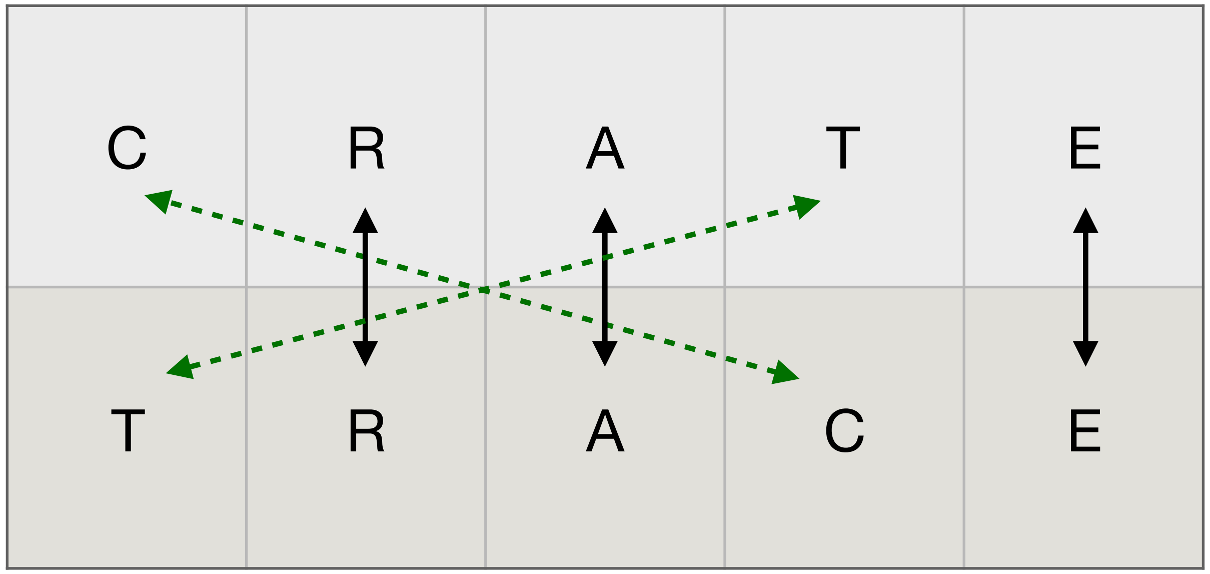
C	R	A	T	E
T	R	A	C	E

Jaro-Winkler similarity:

- **l** - length of the prefices that match exactly (a maximum of 4)
- **p** - scaling coefficient

$$sim_w = sim_j + \ell p(1 - sim_j),$$

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

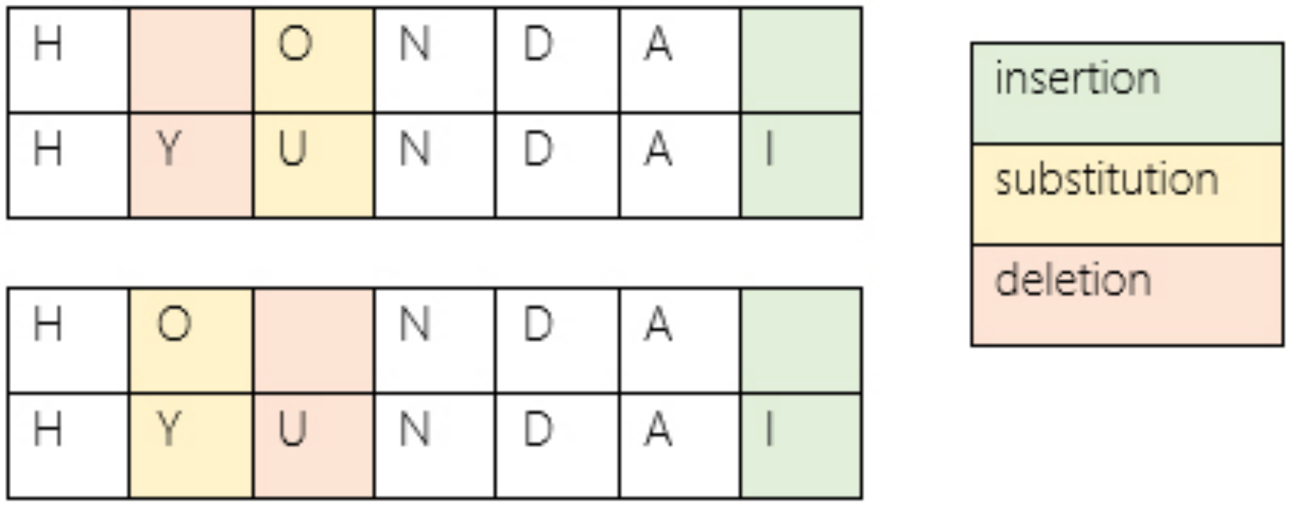


C	R	A	T	E
T	R	A	C	E

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



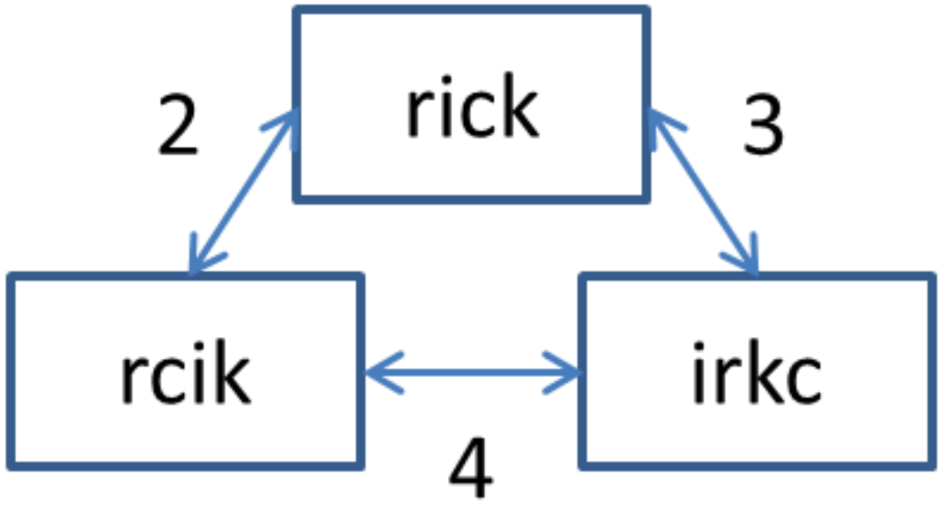
H		O	N	D	A	
H	Y	U	N	D	A	I

H	O		N	D	A	
H	Y	U	N	D	A	I

insertion
substitution
deletion

l_dist = 3

Levenshtein



Levenshtein distance:

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 d(0,0) = 0

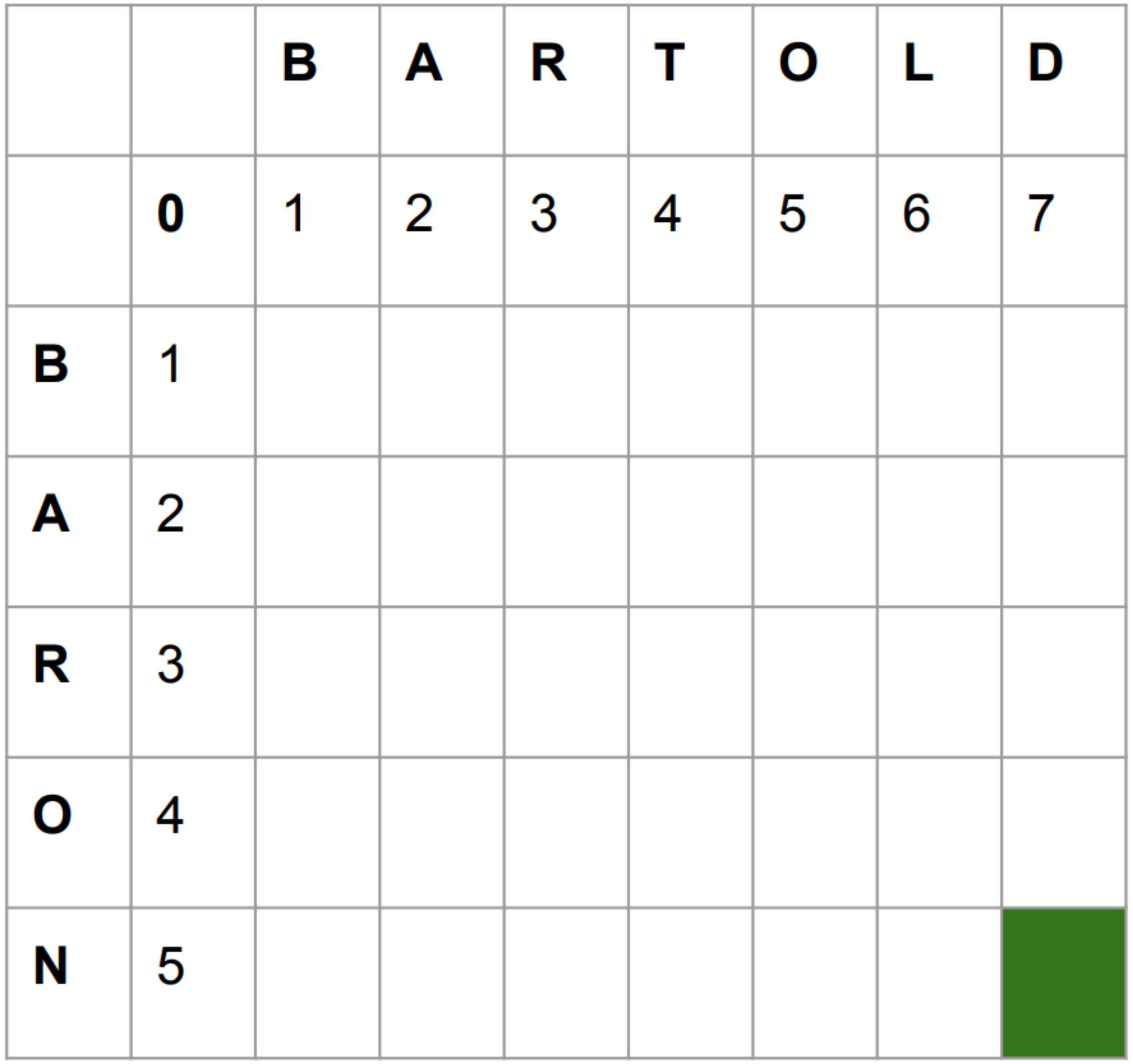
		B	**A**	**R**	**T**	**O**	**L**	**D**
	0	1	2	3	4	5	6	7
B	1							
A	2							
R	3							
O	4							
N	5							🟩

Levenshtein distance:

Wagner–Fischer algorithm

Initially, all empty strings have distance **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:

Wagner–Fischer algorithm

Empty strings have distance **d(0,0) = 0**

Distance between empty one and a non-empty one
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 - **1 +** 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

- You can normalize it into range [0,1] with *max(length(str1), length(str2))* (but still not a metric !)

LCS:

- 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

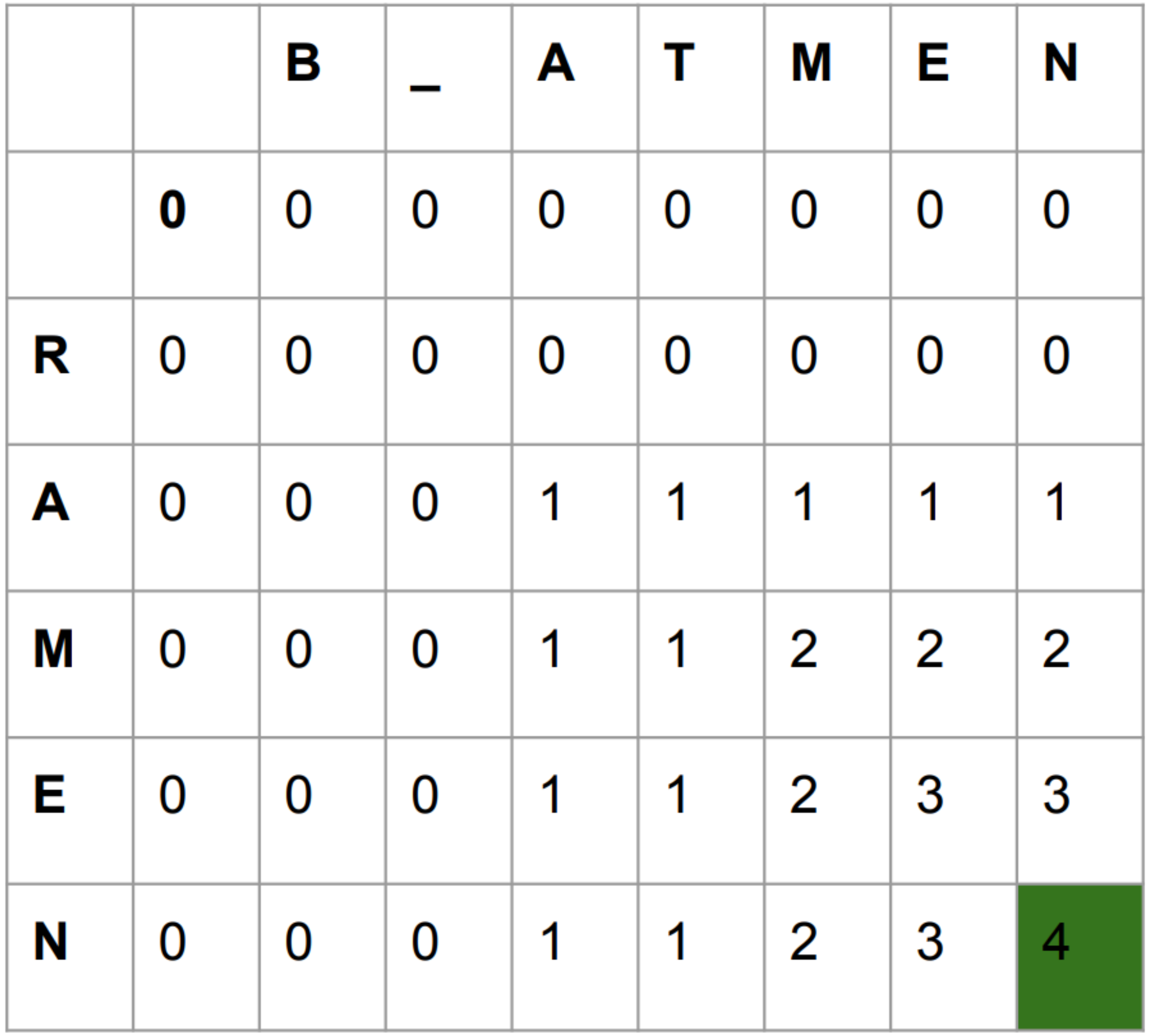
d(0, 0) = 0

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.

Jaccard distance

Jaccard distance: $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$

Suitable as any other «set»-distance;

For tuning:

- duplicates
- bi-uni-tri-…-grams

«ABCD0» (A) vs «ABCDO»(B)

	ab	bc	cd	de	dc	bd
A:	1	1	1	1	0	0
B:	1	0	1	1	1	1

J(A, B) = (3 / 6) = 0.5

Distances in practice:

Never ever implement your own levenshtein/jaccard/BM-25/… in production;

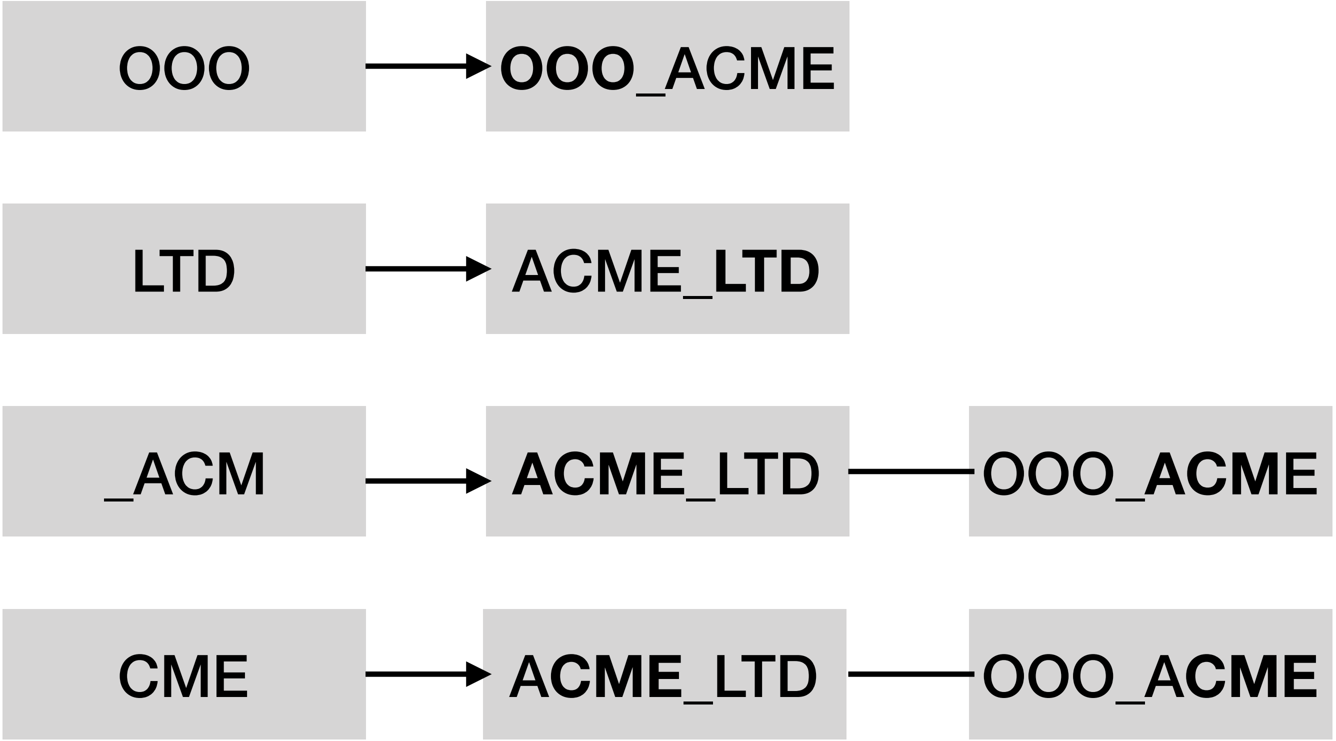
Python:

- nltk.metrics.distance
- python-Levenshtein
- Jellyfish! (+ has soundex!) ...
- Lucene (**Java**) has NgramIndex
- and so on…

N-gram indices

Keep inverted N-gram index to retrieve doc’s with **maximum number of n-grams common** with the query.

Then rank it with more complex approach (hard ML model)



OOO → **OOO**_ACME

LTD → ACME_**LTD**

_ACM → **ACME**_LTD — OOO_**ACME**

CME → **ACME**_LTD — OOO_**ACME**

ML in NLP:

- ML approach:
  - objects: $x \in X$
  - labels: $y \in Y$
  - algorithm: $a(x) : X \to Y$
- **Problems?**

ML in NLP:

- ML approach:
  - objects: $x \in X$
  - labels: $y \in Y$
  - algorithm: $a(x) : X \to Y$
- **Problems?**
- Most algorithms assume $x \in R^n$
- Text don’t look like something from R^n

Vector counters:

- Doc/term matrix
- Words order
- Sparse vectors
- Memory overhead to store dictionary
- Include bi(tri)grams
- Stopwords
- min_tf/min_df, max_tf/max_df
- Binarization

- I love programming;
- Programming also loves me love;

↓

	Love	Programming	also
Doc1	1	1	0
Doc2	2	1	1

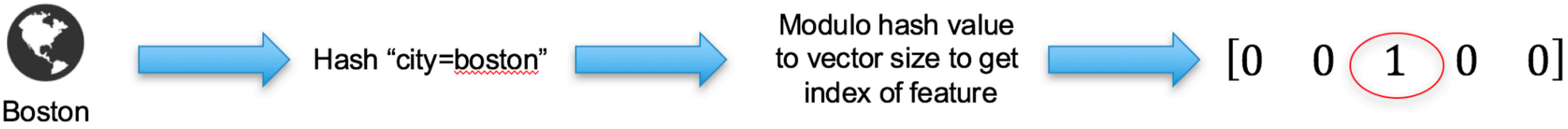
Hashing trick:

- We need to store all vocab. Sometimes it's around 10-20M of strings).
  How many memory will it consume?

- What if we **hash**(word) = idx?
  0 < idx < num_terms

- And then look-up by a hash

- **Pros?**



Boston

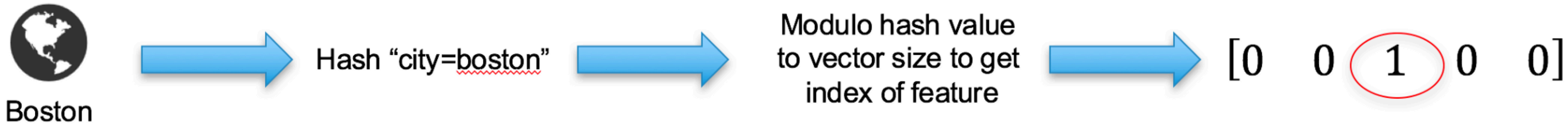
Hash "city=boston"

Modulo hash value
to vector size to get
index of feature

$[0 \quad 0 \quad 1 \quad 0 \quad 0]$

Hashing trick:

- **Pros?**
- No overhead for dictionary store
- Simple production «runtime» inference
- **Cons?**



Boston

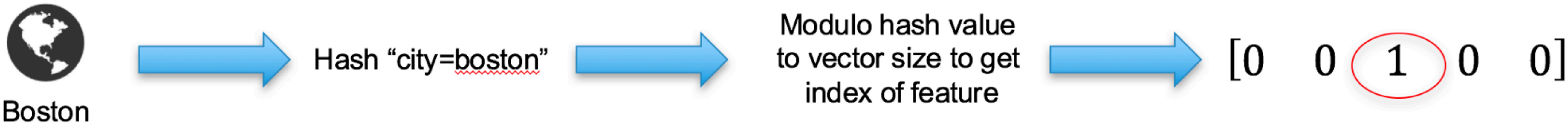
Hash “city=boston”

Modulo hash value to vector size to get index of feature

$[0 \quad 0 \quad 1 \quad 0 \quad 0]$

Hashing trick:

- **Pros?**
- No overhead for dictionary store
- Simple production «runtime» inference
- **Cons?**



Boston

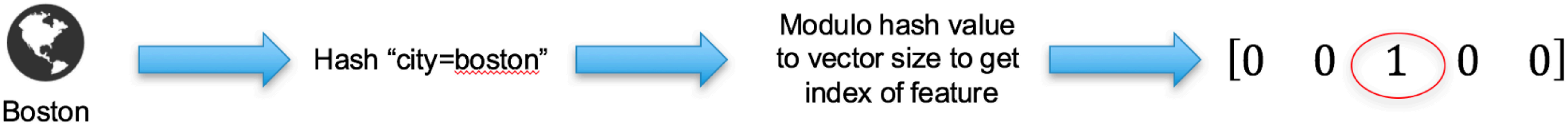
Hash “city=boston”

Modulo hash value to vector size to get index of feature

$[0 \quad 0 \quad 1 \quad 0 \quad 0]$

Hashing trick:

- **Cons?**
- One way mapping: word -> idx, no idx -> word
- Hash collisions
- Memory / precision trade-off



TF-IDF:

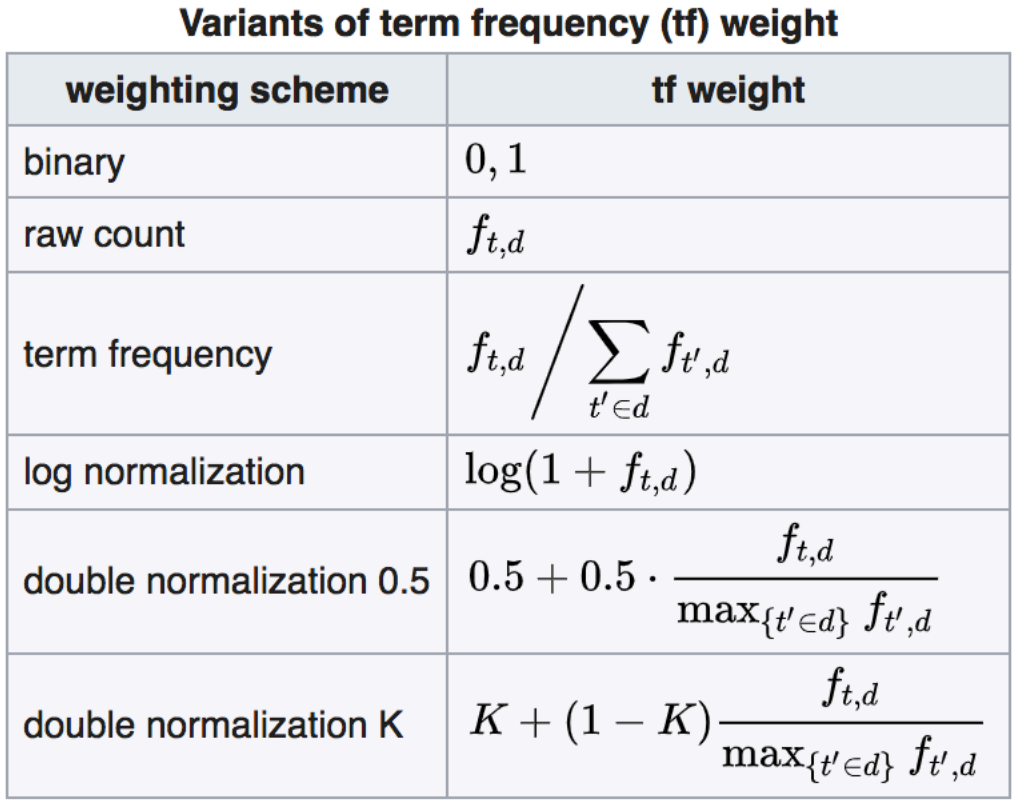
Are there many docs with this word in the collection? (the smaller the number of documents the better)

$$TFIDF = tf(t,d) \times idf(t,D)$$

Is the word met frequently in the document? (more is "better")

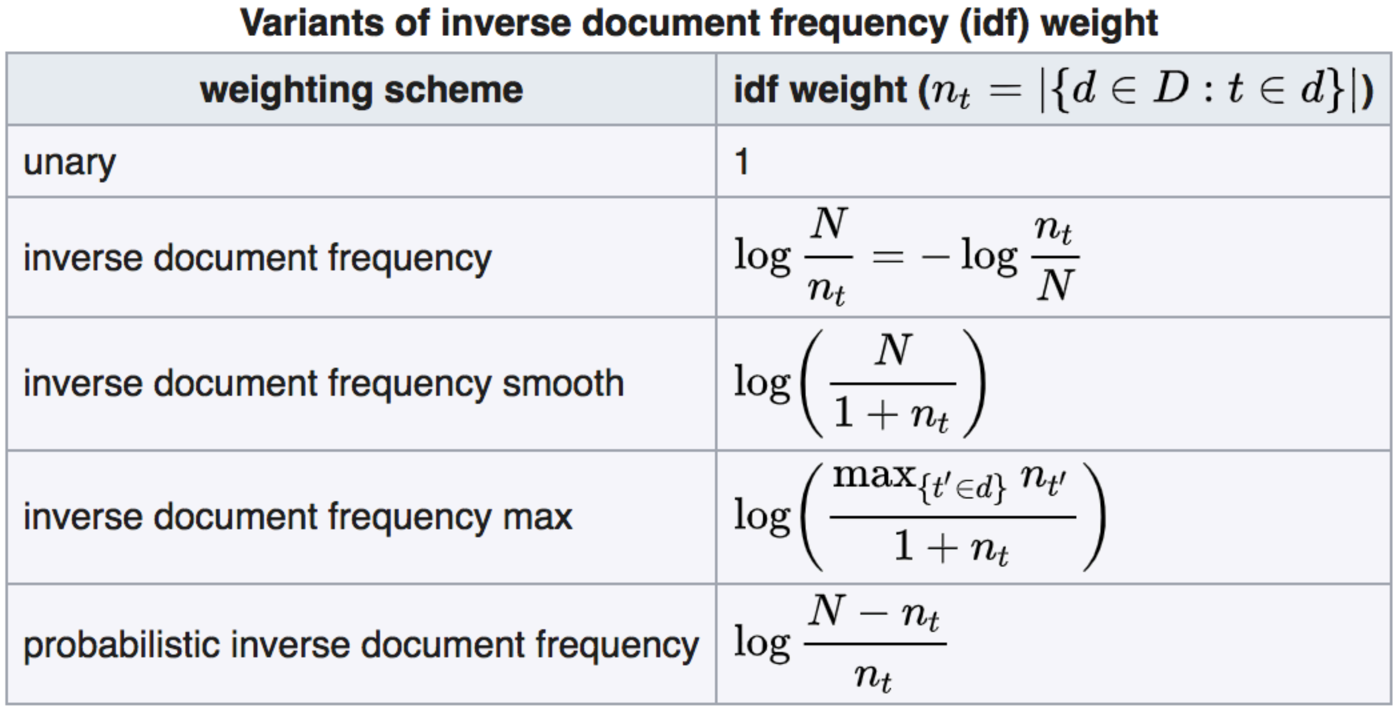
$$f_{t,d} \Big/ \sum_{t' \in d} f_{t',d}$$

$$\mathrm{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$



Variants of term frequency (tf) weight

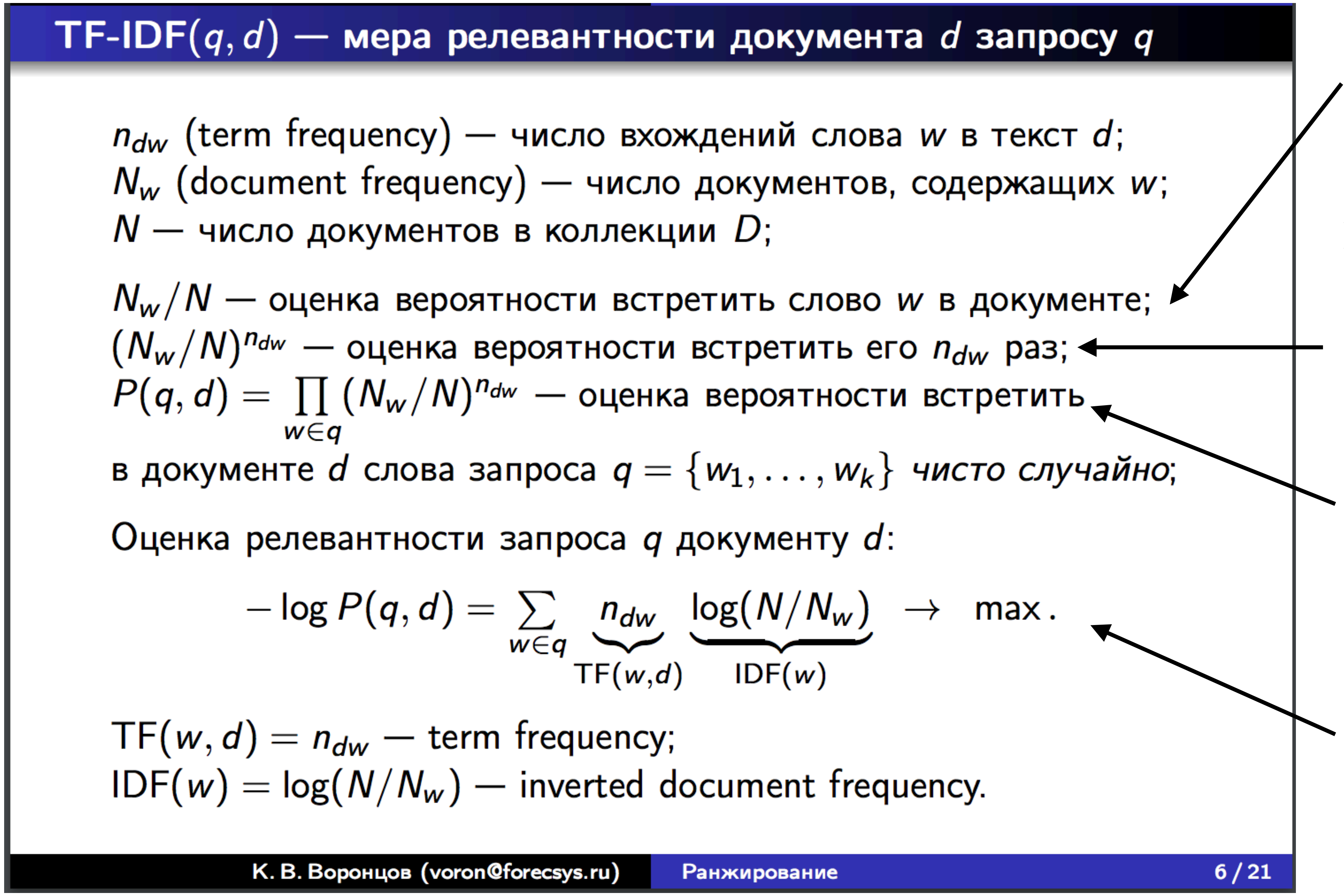
weighting scheme	tf weight
binary	$0, 1$
raw count	$f_{t,d}$
term frequency	$f_{t,d} \Big/ \sum_{t' \in d} f_{t',d}$
log normalization	$\log(1 + f_{t,d})$
double normalization 0.5	$0.5 + 0.5 \cdot \frac{f_{t,d}}{\max_{\{t' \in d\}} f_{t',d}}$
double normalization K	$K + (1 - K)\frac{f_{t,d}}{\max_{\{t' \in d\}} f_{t',d}}$



Variants of inverse document frequency (idf) weight

weighting scheme	idf weight ($n_t = \|\{d \in D : t \in d\}\|$)
unary	1
inverse document frequency	$\log \frac{N}{n_t} = -\log \frac{n_t}{N}$
inverse document frequency smooth	$\log\left(\frac{N}{1 + n_t}\right)$
inverse document frequency max	$\log\left(\frac{\max_{\{t' \in d\}} n_{t'}}{1 + n_t}\right)$
probabilistic inverse document frequency	$\log \frac{N - n_t}{n_t}$

TF-IDF with a theory:



TF-IDF(q,d) — мера релевантности документа d запросу q

n_{dw} (term frequency) — число вхождений слова w в текст d;
N_w (document frequency) — число документов, содержащих w;
N — число документов в коллекции D;

N_w/N — оценка вероятности встретить слово w в документе;
$(N_w/N)^{n_{dw}}$ — оценка вероятности встретить его n_{dw} раз;
$P(q,d) = \prod_{w\in q} (N_w/N)^{n_{dw}}$ — оценка вероятности встретить в документе d слова запроса $q = \{w_1, \dots, w_k\}$ *чисто случайно*;

Оценка релевантности запроса q документу d:

$$-\log P(q,d) = \sum_{w\in q} \underbrace{n_{dw}}_{\mathrm{TF}(w,d)} \underbrace{\log(N/N_w)}_{\mathrm{IDF}(w)} \to \max.$$

$\mathrm{TF}(w,d) = n_{dw}$ — term frequency;
$\mathrm{IDF}(w) = \log(N/N_w)$ — inverted document frequency.

К. В. Воронцов (voron@forecsys.ru) Ранжирование 6 / 21

Probability to see the term **w** in a document

Probability to see the term **w** in a document the number of times it actually shows up in a document (n_dw)

Probability to see query q terms in a document at random

Relevance: the larger, the less the ‘randomness’ of query terms occurence in the document

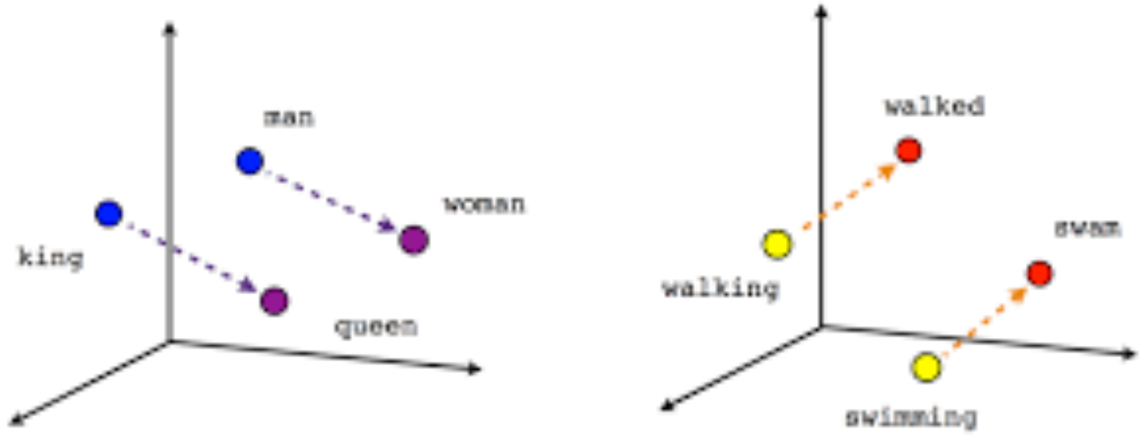
When BoW may not be enough?

- 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

Notes on vector representations:

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

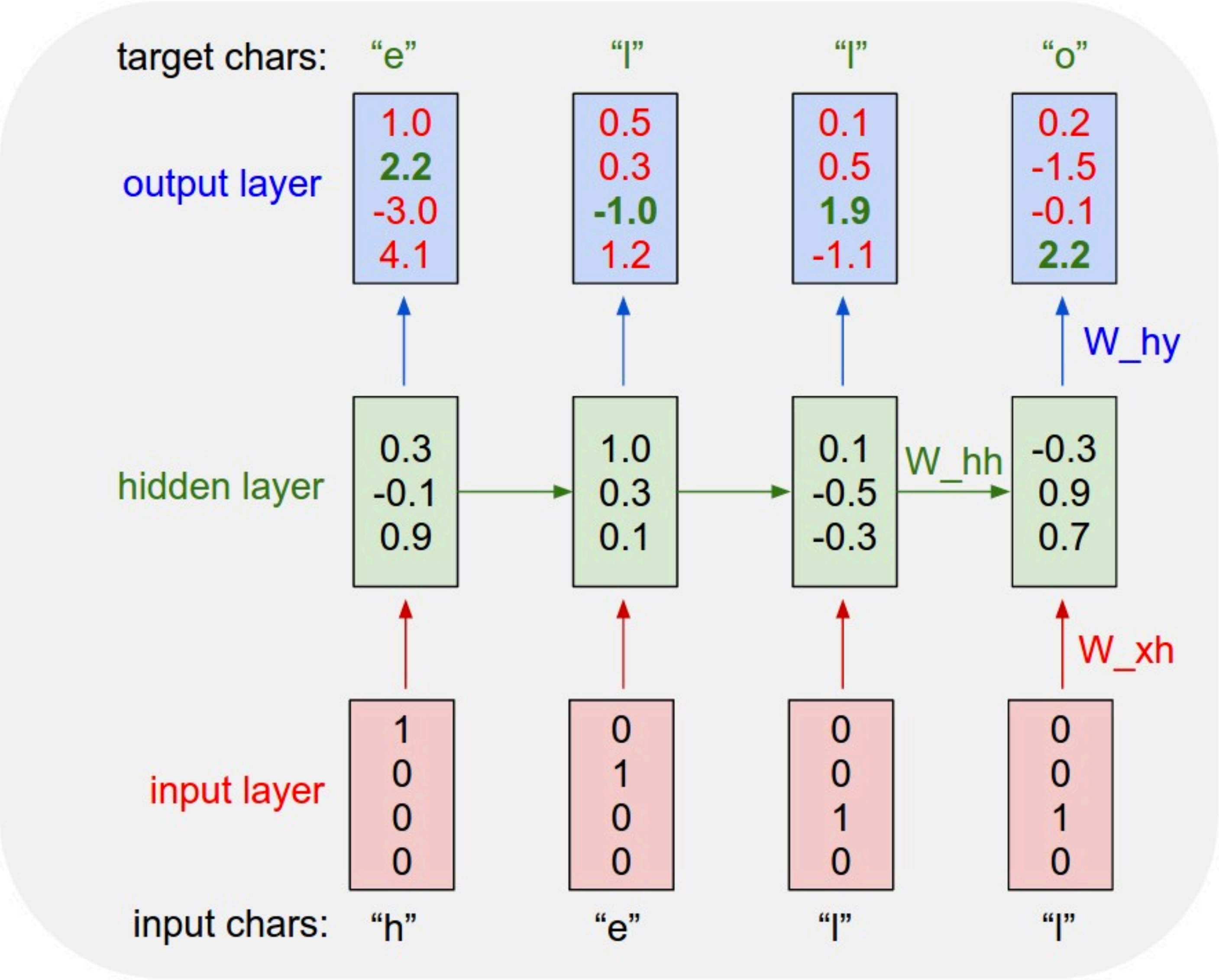


Male-Female

Verb tense

Beyond words 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/



Naive Bayes recap:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Bayes Theorem

$$p(C_k \mid x_1, x_2 \ldots, x_n)$$

Conditional probability **C_k**
by **given features** (x_1..x_n)

Naive Bayes recap:

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)} \propto p(C_k)p(x|C_k) = p(C_k, x_1, x_2, ..., x_n)$$

Conditional probability via Bayes theorem
and joint probability

$$\begin{aligned}
&p(C_k, x_1, x_2, ...x_n) \\
&= p(x_1|x_2, ..., x_n, C_k) \cdot p(x_2, ..., x_n, C_k) \\
&= p(x_1|x_2, ..., x_n, C_k) \cdot p(x_2|x_3, ..., x_n, C_k) \cdot p(x_3, ..., x_n, C_k) \\
&= p(x_1|x_2, ..., x_n, C_k) \cdot p(x_2|x_3, ..., x_n, C_k) ... \cdot p(x_n|C_k) \cdot C_k
\end{aligned}$$

Total joint probability

Naive Bayes recap:

$$\begin{aligned} & p(C_k|x_1, x_2, ..., x_n) \\ \propto\ & p(C_k, x_1, x_2, ...x_n) \\ =\ & p(x_1|C_k) \cdot p(x_2|C_k) ... \cdot p(x_n|C_k) \cdot p(C_k) \\ =\ & p(C_k) \prod_{i=1}^{n} p(x_i|C_k) \end{aligned}$$

The final formula for Naïve Bayes Classifier.

$$p(C_k) = p(y = C_k) = \frac{\sum_{i=1}^{N} I(y_i = C_k)}{N}$$

Maximum likelihood estimation (**MLE**)
for **prior probability** estimation

$$p(x_1 = a_j|y = C_k) = \frac{\sum_{i=1}^{N} I(x_1 i = a_j, y_i = C_k)}{\sum_{i=1}^{N} I(y_i = C_k)}$$

Maximum likelihood estimation (**MLE**)
for **conditional probability** estimation

$$y = arg\, max\, p(y = C_k) \prod p(x|y = C_k)$$

Prediction rule

Naive Bayes recap:

$$p_\lambda(C_k) = p_\lambda(Y = C_k) = \frac{\sum_{i=1}^{N} I(y_i = C_k) + \lambda}{N + K\lambda}$$

$$p(x_1 = a_j | y = C_k) = \frac{\sum_{i=1}^{N} I(x_{1i} = a_j, y_i = C_k) + \lambda}{\sum_{i=1}^{N} I(y_i = C_k) + A\lambda}$$

Laplace Smoothing to solve zero-probability problem.
K - num different y,
A - num different a_j,
\lambda — parameter, usually eq 1

Naive Bayes recap:

- Pros
  - Computationally fast (one pass)
  - Simple to implement
  - Works well with high dimensions
- Cons
  - Naive
  - Accuracy
  - Flaoting problems with zero-close values (google log trick!)

Homework:

- Download sms-spam dataset https://archive.ics.uci.edu/ml/datasets/sms+spam+collection
- Choose and argument metric for quality
- Code «by a hands» naive bayes for spam detection task;
- Choose a measure of a test's accuracy and argument your choice; Perform 5-fold validation for this task;
- Compare your results with sklearn naive_bayes
- I expect your result as self-sufficient (with all comments/graph/etc.) Jupiter notebook in your GitHub in 2 weeks (next lecture).