

Computational Communication Science 2

Week 2 - Lecture

»Bottom up approaches to text analysis«

Anne Kroon

a.c.kroon@uva.nl, @annekroon

April 8, 2024

Digital Society Minor, University of Amsterdam

Today

From text to features: vectorizers

- General idea

- Pruning

Cosine Similarity

Soft cosine similarity

- Word embeddings

- Implementation in Python

From test to large-scale

From text to features: vectorizers

From text to features: vectorizers

General idea

○
 ●●○○○○○○○○○○○○○○○○
 ○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
 ○○
 ○○○○○○

○○○○○

A text as a collections of word

Let us represent a string

```
1 t = "This this is is is a test test test"
2 # like this:
3 print(Counter(t.split()))
```

```
1 Counter({'is': 3, 'test': 3, 'This': 1, 'this': 1, 'a': 1})
```

Compared to the original string, this representation

- is less repetitive
- preserves word frequencies
- but does *not* preserve word order
- can be interpreted as a vector to calculate with (!!!)

Of course, still a lot of stuff to fine-tune... (for example, This/this)

○
○○●○○○○○○○○○○○○○○
○○○○

○○○○○○○○○○○○○○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

From vector to matrix

If we do this for multiple texts, we can arrange the vectors in a table.

$t1 = \text{"This this is is is a test test test"}$

$t2 = \text{"This is an example"}$

	a	an	example	is	this	This	test
$t1$	1	0	0	3	1	1	3
$t2$	0	1	1	1	0	1	0



What can you do with such a matrix? Why would you want to represent a collection of texts in such a way?

○
○○○○●○○○○○○○○○○○
○○○

○○○○○○○○○○○○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

What is a vectorizer

- Transforms a list of texts into a sparse (!) matrix (of word frequencies)
- Vectorizer needs to be “fitted” to the training data (learn which words (features) exist in the dataset and assign them to columns in the matrix)
- Vectorizer can then be re-used to transform other datasets

○
 ○○○○○●○○○○○○○○○○
 ○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
 ○○
 ○○○○○○

○○○○○

The cell entries: raw counts versus tf·idf scores

- In the example, we entered simple counts (the “term frequency”)



But are all terms equally important?

○
 ○○○○○○○●○○○○○○○
 ○○○○

○○○○○○○○○○○○○○○○

○○○○○
 ○○○
 ○○○○○○

○○○○○

The cell entries: raw counts versus tf·idf scores

- In the example, we entered simple counts (the “term frequency”)
- But does a word that occurs in almost all documents contain much information?
- And isn't the presence of a word that occurs in very few documents a pretty strong hint?
- **Solution:** Weigh by *the number of documents in which the term occurs at least once* (the “document frequency”)

⇒ we multiply the “term frequency” (tf) by the inverse document frequency (idf)

(usually with some additional logarithmic transformation and normalization applied, see https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html)

○
 ○○○○○○○○●○○○○○○○
 ○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
 ○○
 ○○○○○○

○○○○○

tf·idf

$$w_{i,j} = tf_{i,j} \times \log \left(\frac{N}{df_i} \right)$$

$tf_{i,j}$ = number of occurrences of i in j

df_i = number of documents containing i

N = total number of documents

○
 ○○○○○○○○○●○○○○○○○
 ○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
 ○○○
 ○○○○○○

○○○○○

Is tf·idf always better?

It depends.

- Ultimately, it's an empirical question which works better (→ machine learning)
- In many scenarios, “discounting” too frequent words and “boosting” rare words makes a lot of sense (most frequent words in a text can be highly un-informative)
- Beauty of raw tf counts, though: interpretability + describes document in itself, not in relation to other documents

○
 ○○○○○○○○○○●○○○○○
 ○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
 ○○
 ○○○○○○

○○○○○

Different vectorizers

1. CountVectorizer (=simple word counts)
2. TfidfVectorizer (word counts (“term frequency”) weighted by number of documents in which the word occurs at all (“inverse document frequency”))

○
 ○○○○○○○○○○●○○○
 ○○○

○○○○○○○○○○○○○○○○

○○○○
 ○○
 ○○○○○

○○○○○

Internal representations

Sparse vs dense matrices

- → tens of thousands of columns (terms), and one row per document
- Filling all cells is inefficient *and* can make the matrix too large to fit in memory (!!!)
- Solution: store only non-zero values with their coordinates! (sparse matrix)
- dense matrix (or dataframes) not advisable, only for toy examples

o
o
oooooooooooo●ooo
oooo

oooooooooooooooooooo

ooooo
oo
ooooooo

ooooo

s p a r s e

0	7	0	0	0	0	6
0	7	6	3	0	4	0
0	4	3	0	0	0	0
4	2	0	0	0	0	0
0	0	0	0	3	2	4

© Matt Edging

DENSE

0	7	0	0	0	0	6
0	7	6	3	0	4	0
0	4	3	0	0	0	0
4	2	0	0	0	0	0
0	0	0	0	3	2	4

<https://matteding.github.io/2019/04/25/sparse-matrices/>

○
 ○○○○○○○○○○○○○●○○
 ○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
 ○○
 ○○○○○○

○○○○○

We learned in week 1 how to tokenize with a list comprehension (and that's often a good idea!).

```
1 from nltk.tokenize import TreebankWordTokenizer
2 tokens = [TreebankWordTokenizer().tokenize(d) for d in docs]
```

But what if we want to *directly* get a DTM instead of lists of tokens?

○
○○○○○○○○○○○○○○○●○
○○○

○○○○○○○○○○○○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

OK, good enough, perfect?

scikit-learn's CountVectorizer (default settings)

- applies lowercasing
- deals with punctuation etc. itself
- minimum word length > 1
- more technically, tokenizes using this regular expression:
`r"(?u)\b\w\w+\b"`¹

```
1 from sklearn.feature_extraction.text import CountVectorizer
2 cv = CountVectorizer()
3 dtm_sparse = cv.fit_transform(docs)
```

¹?u = support unicode, \b = word boundary

○
○○○○○○○○○○○○○○○○●
○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
○○
○○○○○○

○○○○○

OK, good enough, perfect?

CountVectorizer supports more

- stopword removal
- custom regular expression
- or even using an external tokenizer
- ngrams instead of unigrams

see

https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

Best of both worlds

Use the Count vectorizer with a NLTK-based external tokenizer! (see book)

From text to features: vectorizers

Pruning

○
 ○○○○○○○○○○○○○○○○
 ○●○○

○○○○○○○○○○○○○○○○○○

○○○○○
 ○○
 ○○○○○○

○○○○○

General idea

- Idea behind both stopwords removal and tf-idf: too frequent words are uninformative
- (possible) downside stopwords removal: a priori list, does not take empirical frequencies in dataset into account
- (possible) downside tf-idf: does not reduce number of features

Pruning: remove all features (tokens) that occur in less than X or more than X of the documents

CountVectorizer, only stopwords removal

```
1 from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
2 myvectorizer = CountVectorizer(stop_words=mystopwords)
```

CountVectorizer, better tokenization, stopwords removal (pay attention that stopwords list uses same tokenization!):

```
1 myvectorizer = CountVectorizer(tokenizer = TreebankWordTokenizer().tokenize,
  ↳ stop_words=mystopwords)
```

Additionally remove words that occur in more than 75% or less than $n = 2$ documents:

```
1 myvectorizer = CountVectorizer(tokenizer = TreebankWordTokenizer().tokenize,
  ↳ stop_words=mystopwords, max_df=.75, min_df=2)
```

All together: tf-idf, explicit stopwords removal, pruning

```
1 myvectorizer = TfidfVectorizer(tokenizer = TreebankWordTokenizer().tokenize,
  ↳ stop_words=mystopwords, max_df=.75, min_df=2)
```



*What is “best”? Which
(combination of) techniques to
use, and how to decide?*

Cosine Similarity

○
○○○○○○○○○○○○○○○○○○
○○○○

○●○○○○○○○○○○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

Cosine Similarity

Cosine Similarity

- A measure that helps you determine how similar two documents are, irrespective of their size

Applications in Communication Science

- For example, to map *linguistic alignment* of romantic couples over time (Brinberg & Ram, 2021)
- Or, in the political domain, agenda overlap between public opinion and political speech (Hager & Hilbig, 2020)

○
○○○○○○○○○○○○○○○○○○
○○○○

○○●○○○○○○○○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

Cosine Similarity

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}}$$

- It measures the cosine of the angle between two vectors projected in a multi-dimensional space.
- 0 means orthogonal vectors (90 degrees); very dissimilar
- 1 means vectors are the same (0 degrees); similar

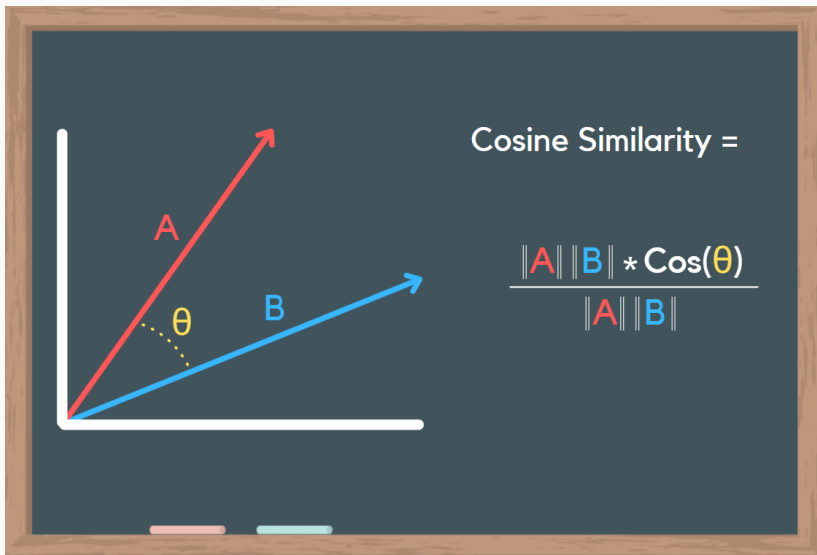
○
○○○○○○○○○○○○○○○○○○
○○○○

○○○●○○○○○○○○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

Cosine Similarity



○
○○○○○○○○○○○○○○○○○○
○○○○

○○○○●○○○○○○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

how can we calculate this in python?

Let's review a practical application ³.

³https://github.com/uva-cw-ccs2/2223s2/blob/master/week03/exercises/OPTIONAL_overtime_similarity.ipynb

```

1 from sklearn.feature_extraction.text import CountVectorizer,
  ↳ TfidfVectorizer
2 import pandas as pd
3
4 doc1 = "When I eat breakfast, I usually drink some tea".lower()
5 doc2 = "I like my tea with my breakfast".lower()
6 doc3 = "She likes cereal and coffee".lower()
7
8 vec = CountVectorizer(stop_words='english')
9 count_matrix = vec.fit_transform([doc1, doc2, doc3])
10
11 print(pd.DataFrame(count_matrix.A,
  ↳ columns=vec.get_feature_names_out()).to_string())

```

	breakfast	cereal	coffee	drink	eat	like	likes	tea	usually
0	1	0	0	1	1	0	0	1	1
1	1	0	0	0	0	1	0	1	0
2	0	1	1	0	0	0	1	0	0

○
○○○○○○○○○○○○○○○○○○
○○○○

○○○○○○●○○○○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

Implementation in Python

```

1 doc1_vector = pd.DataFrame(count_matrix.A,
  ↳ columns=vec.get_feature_names_out()).T[0].to_list()
2 doc2_vector = pd.DataFrame(count_matrix.A,
  ↳ columns=vec.get_feature_names_out()).T[1].to_list()
3
4 print(f"The vector belonging to doc1: {doc1_vector}")
5 print(f"The vector belonging to doc2: {doc2_vector}")

```

```

1 The vector belonging to doc1: [1, 0, 0, 1, 1, 0, 0, 1, 1]
2 The vector belonging to doc2: [1, 0, 0, 0, 0, 1, 0, 1, 0]

```

Now, let's populate the formula. 1. Execute the part of the formula in the numerator. Specifically, take the dot product of the vectors:

$$\sum_{i=1}^n A_i B_i$$

```
1 doc1: [1, 0, 0, 1, 1, 0, 0, 1, 1]
2 doc2: [1, 0, 0, 0, 0, 1, 0, 1, 0]
```

Or, using Python:

```
1 dot_product = sum([num1 * num2 for num1, num2 in zip(doc1_vector,
↳ doc2_vector)])
2 print(dot_product)
```

1	2
---	---

○
○○○○○○○○○○○○○○○○○○
○○○○

○○○○○○○○●○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

Implementation in Python

2. Execute the part of the formula in the denominator. Take the cross product of the two vectors.

$$\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}$$

Calculate this by hand:

$$\text{doc1} = \sqrt{1^2 + 0^2 + 0^2 + 1^2 + 1^2 + 0^2 + 1^2 + 1^2}$$

$$\text{doc1} = \sqrt{1^2 + 0^2 + 0^2 + 0^2 + 1^2 + 0^2 + 1^2 + 0^2}$$

Implementation in Python:

```
1 import math
2 doc1_ = math.sqrt(sum( [i**2 for i in doc1_vector] ) )
3 doc2_ = math.sqrt(sum( [i**2 for i in doc2_vector] ) )
4
```


○
○○○○○○○○○○○○○○○○○○
○○○○

○○○○○○○○○○●○○○○○○○

○○○○○
○○
○○○○○○○

○○○○○

Implementation in Python

3.Finally:

```
1 cos_sim = dot_product / (doc1_ * doc2_)
2 print(cos_sim)
```

```
1 0.5163977794943222
```

```

o
oooooooooooooooooooo
oooo

```

```

oooooooooooo●ooooo

```

```

ooooo
oo
oo
ooooooo

```

```

ooooo

```

Implementation in Python

We can, however, do this much faster using sklearn's `cosine_similarity`.

```

1 from sklearn.metrics.pairwise import cosine_similarity
2 cosine_similarity([doc1_vector, doc2_vector])

```

```

1 array([[1.          , 0.51639778],
2        [0.51639778, 1.          ]])

```

How to use this in practice

What can you do with this?

- This is especially powerful if you want to compare different news articles, movies, song texts, etc.
- For example, which movies are most similar in terms of genre composition?

From text to features: vectorizers

○
○○○○○○○○○○○○○○○○○○○
○○○○

Cosine Similarity

○○○○○○○○○○○○○○●○○○

Soft cosine similarity

○○○○○
○○
○○○○○○○

From test to large-scale

○○○○○

References

	title	Avatar	Pirates of the Caribbean: At World's End	Spectre	The Dark Knight Rises	John Carter
	genre	action adventure fantasy science fiction	adventure fantasy action	action adventure crime	action crime drama thriller	action adventure science fiction
title	genres					
Avatar	action adventure fantasy science fiction	1.000000	0.691870	0.315126	0.084696	0.859850
Pirates of the Caribbean: At World's End	adventure fantasy action	0.691870	1.000000	0.455470	0.122417	0.366490
Spectre	action adventure crime	0.315126	0.455470	1.000000	0.473354	0.366490
The Dark Knight Rises	action crime drama thriller	0.084696	0.122417	0.473354	1.000000	0.098501
John Carter	action adventure science fiction	0.859850	0.366490	0.366490	0.098501	1.000000

Identify movies that are similar in terms of genre ⁴

⁴<https://www.learndatasci.com/glossary/cosine-similarity/>

○
○○○○○○○○○○○○○○○○○○○
○○○○

○○○○○○○○○○○○○○○○○○●○

○○○○○
○○
○○○○○○○

○○○○○

Implementation in Python

```
1 doc1 = "When I eat breakfast, I usually drink some tea".lower()
2 doc2 = "I like my tea with my breakfast".lower()
3 doc3 = "She likes cereal and coffee".lower()
```

What do you expect here? Should there be some level of overlap?

```
1 [[1.          0.51639778 0.          ]
2  [0.51639778 1.          0.          ]
3  [0.          0.          1.          ]]
```

Zero overlap between doc3 and the other documents. Is that correct?

Soft cosine similarity

Soft cosine similarity

...enter soft cosine similarity Sidorov et al. (2014)

“Soft Cosine Measure (SCM) is a method that allows us to assess the similarity between two documents in a meaningful way, even when they have no words in common. It uses a measure of similarity between words, which can be derived using [word2vec] vector embeddings of words.”⁵

⁵https://radimrehurek.com/gensim//auto_examples/tutorials/run_scm.html

○
○○○○○○○○○○○○○○○○○○
○○○○

○○○○○○○○○○○○○○○○○○

○○●○○
○○
○○○○○○○

○○○○○

Soft Cosine Measure (SCM)

SCM

- Even if two sentences do not share the same words, we can calculate similarity by modelling *synonym*
- For example, the words ‘play’ and ‘game’ are different but related Sidorov et al. (2014) ⁶
- How can we capture ‘semantic’ meaning?

How?

- Convert words to *word vectors* and then compute similarities ⁷

⁶<http://www.scielo.org.mx/pdf/cys/v18n3/v18n3a7.pdf>

⁷<https://www.machinelearningplus.com/nlp/cosine-similarity/>



○
○○○○○○○○○○○○○○○○○○
○○○○

○○○○○○○○○○○○○○○○○○

○○○○●
○○
○○○○○○

○○○○○

Word embeddings

Word embeddings

- To use the SCM, you need word embeddings.

Soft cosine similarity

Word embeddings

○
○○○○○○○○○○○○○○○○○○
○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
○○●
○○○○○○○

○○○○○

Understanding SCM

SCM estimates extracts similarity from **word embeddings**.

What are word embeddings?

- No technical details here, just the general idea
- Word embeddings help capturing the meaning of text
- Word embeddings are low-dimensional vector representations that capture semantic meaning
- Used to be state-of-the-art in NLP (but now: contextualized embeddings, e.g., BERT or GPT)
- *“...a word is characterized by the company it keeps...”* (Firth, 1957)

Soft cosine similarity

Implementation in Python

SCM in Python

Calculating Soft Cosine Measure

- To use the SCM, you need embeddings.
- We *can* train embeddings on our own corpus (if we had a lot of data) ...
- But for now we will use pre-trained models ⁹. ...

```
1 import gensim.downloader as api
2
3 fasttext_model300 = api.load('fasttext-wiki-news-subwords-300')
```

⁹<https://github.com/uva-cw-ccs2/2223s2/blob/master/week03/exercises/cosine-similarity-basics.ipynb>

Create a dictionary

Let's review our 3 documents:

```
1 doc1 = "When I eat breakfast, I usually drink some tea".lower()
2 doc2 = "I like my tea with my breakfast".lower()
3 doc3 = "She likes cereal and coffee".lower()
```

Initialize a Dictionary. This step assigns a `token_id` to each word:

```
1 from gensim.utils import simple_preprocess
2 from gensim.corpora import Dictionary
3 dictionary = corpora.Dictionary([simple_preprocess(doc) for doc in [doc1, doc2, doc3]])
```

Now, let's check whether a specific word—for example `coffee`—is in our dictionary:

```
1 'coffee' in dictionary.token2id
```

```
1 True
```

Create a bag-of-words representation

Next, let's represent each document by (token_id, token_count) tuples:

```
1 bag_of_words_vectors = [ dictionary.doc2bow(simple_preprocess(doc))  
    ↪ for doc in [doc1, doc2, doc3]]
```

Build a term similarity matrix and compute a sparse term similarity matrix:

```
1 from gensim.similarities import SparseTermSimilarityMatrix
2 from gensim.similarities import WordEmbeddingSimilarityIndex
3
4 similarity_index = WordEmbeddingSimilarityIndex(fasttext_model300)
5 similarity_matrix = SparseTermSimilarityMatrix(similarity_index,
6 ↪ dictionary)
```

Inspect results

Get SCM using `.inner_product()`:

```
#between doc1 and doc2:
scm_doc1_doc2 = similarity_matrix.inner_product(bag_of_words_vectors[0],
↳ bag_of_words_vectors[1], normalized=(True, True))

#between doc1 and doc3:
scm_doc1_doc3 = similarity_matrix.inner_product(bag_of_words_vectors[0],
↳ bag_of_words_vectors[2], normalized=(True, True))

#between doc2 and doc3;
scm_doc2_doc3 = similarity_matrix.inner_product(bag_of_words_vectors[1],
↳ bag_of_words_vectors[2], normalized=(True, True))

print(f"SCM between:\ndoc1 <-> doc2: {scm_doc1_doc2:.2f}\ndoc1 <-> doc3:
↳ {scm_doc1_doc3:.2f}\ndoc2 <-> doc3: {scm_doc2_doc3:.2f}")
```

○
○○○○○○○○○○○○○○○○○○
○○○○

○○○○○○○○○○○○○○○○○○

○○○○○
○○
○○○○○●○

○○○○○

Inspect results

Do the results make more sense?:

```
1 doc1 = "When I eat breakfast, I usually drink some tea".lower()
2 doc2 = "I like my tea with my breakfast".lower()
3 doc3 = "She likes cereal and coffee".lower()
```

```
1 SCM between:
2 doc1 <-> doc2: 0.29
3 doc1 <-> doc3: 0.15
4 doc2 <-> doc3: 0.28
```

Applications of cosine and soft cosine similarity

Applications of *cosine* and *soft cosine* in the field of Communication Science generally involve some overtime dynamics.

Trace convergence or agenda setting dynamics over time

- Beyond the scope this course to discuss it here, but if you are interested in how you can apply cosine and soft cosine in an overtime analysis, we have prepared a notebook that will help you do just that.
- https://github.com/uva-cw-ccs2/2223s2/main/week03/exercises/OPTIONAL_overtime_similarity.ipynb

From test to large-scale

General approach

Test on a single string, then make a for loop or list comprehension!

Own functions

If it gets more complex, you can write your own function and then use it in the list comprehension:

```

1 def mycleanup(t):
2     # do sth with string t here, create new string t2
3     return t2
4
5 results = [mycleanup(t) for t in allmytexts]
```


Pandas string methods as alternative

If you select column with strings from a pandas dataframe, pandas offers a collection of string methods (via `.str.`) that largely mirror standard Python string methods:

1

```
df['newcolumnwithresults'] = df['columnwithtext'].str.count("bla")
```

To pandas or not to pandas for text?

Partly a matter of taste.

Not-too-large dataset with a lot of extra columns? Advanced statistical analysis planned? Sounds like pandas.

It's mainly a lot of text? Wanna do some machine learning later on anyway? It's large and (potentially) messy? Doesn't sound like pandas is a good idea.

Thank you!!

Thank you for your attention!

- Questions? Comments?

References

References



Brinberg, M., & Ram, N. (2021). Do new romantic couples use more similar language over time? Evidence from intensive longitudinal text messages. *Journal of Communication*, 71(3), 454–477. <https://doi.org/10.1093/joc/jqab012>



Hager, A., & Hilbig, H. (2020). Does public opinion affect political speech? *American Journal of Political Science*, 64(4), 921–937. <https://doi.org/10.1111/ajps.12516>



Sidorov, G., Gelbukh, A., Gómez-Adorno, H., & Pinto, D. (2014). Soft similarity and soft cosine measure: Similarity of features in vector space model. *Computacion y Sistemas*, 18(3), 491–504. <https://doi.org/10.13053/CyS-18-3-2043>