Big Data and Automated Content Analysis (6EC)

Week 4: »Processing textual data« Monday

Anne Kroon a.c.kroon@uva.nl September 25, 2023

UvA RM Communication Science

Today

Basic string operations

Basic string operations

Regular expressions

What is a regexp?

Using a regexp in Python

The bag-of-words (BOW) model

General idea

A cleaner BOW representation

Better tokenization

Stopword removal

Pruning

Stemming and lemmatization

The order of preprocessing steps

Hopefully, everything is clear from last week. If you have questions about today's lecture, please send me an e-mail or sign up for individual consultation.

https://docs.google.com/spreadsheets/d/1vYAhM00Kwn2kc7XJAN2sBeDqZxg-8TjoPGxseAKRWNk/ edit#gid=0

This week, we will get a general overview of working with textual data. This knowledge will help you to get started with cool automated content analyses techniques – which we will start with next week.

- 1. string methods that every string has ("hello".upper())

- 1. string methods that every string has ("hello".upper())
- 2. functions that take a string as input (len("hello"))

1. string methods that every string has ("hello".upper())

The BOW

- 2. functions that take a string as input (len("hello"))
- 3. pandas column string methods (df["somecolumn"].str.upper())

Basic string operations

- 1. string methods that every string has ("hello".upper())
- 2. functions that take a string as input (len("hello"))
- pandas column string methods (df["somecolumn"].str.upper())
- 4. applying string methods or functions to a pandas column
 (df["somecolumn"].apply(len) or
 df["somecolumn"].apply(lambda x: x.upper())

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

Basic string operations

- 1. string methods that every string has ("hello".upper())
- 2. functions that take a string as input (len("hello"))
- pandas column string methods (df["somecolumn"].str.upper())
- 4. applying string methods or functions to a pandas column
 (df["somecolumn"].apply(len) or
 df["somecolumn"].apply(lambda x: x.upper())

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

Basic string operations

00000000

- 1. string methods that every string has ("hello".upper())
- 2. functions that take a string as input (len("hello"))
- pandas column string methods (df["somecolumn"].str.upper())
- 4. applying string methods or functions to a pandas column
 (df["somecolumn"].apply(len) or
 df["somecolumn"].apply(lambda x: x.upper())

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

00000000

An example says more than 1000 words...

```
# probably read from text file(s) instead, you learned that already...
    data = [ "I <b>really</b> liked this movie! It was great. ", " What

→ an awful movie", "Awesome!!!"]

3
    data_stripped = [e.strip() for e in data]
4
    data_lower = [e.lower() for e in data_stripped]
5
    data_clean = [e.replace("<b>",").replace("</b>",") for e in

    data_lower]

7
    # or, more efficient, in one single step:
    data_clean2 = [e.strip().lower().replace("<b>","").replace("</b>","")

    → for e in datal
```

Two examples says even more:

Basic string operations

000000000

1 2 3

4 5

6

8

9

10

```
from string import punctuation
# punctuation is just the string '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{/}~'
text = "This is a test! Let's get rid (of) punct&"
# we make a list of each character in the text but only if it is not
# a punctuation sign. The, we join the elements of the list directly
# to each other without anything between it ("")
cleantext = "".join([c for c in text if c not in punctuation])
```

Combine both

Basic string operations

000000000

2

5

6

8

```
from string import punctuation

def strip_punctuation(text):
    return "".join([c for c in text if c not in punctuation])

data_clean3 = [strip_punctuation(e).strip().lower()\
    .replace("<b>","").replace("</b>","") for e in data]
```

The toolbox at a glance

Slicing

Basic string operations

000000000

mystring[2:5] to get the characters with indices 2,3,4

String methods

- .lower() returns lowercased string
- .strip() returns string without whitespace at beginning and end
- .find("bla") returns index of position of substring "bla" or -1 if not found
- .replace("a", "b") returns string with "a" replaced by "b"
- .count("bla") counts how often substring "bla" occurs
- .isdigit() true if only numbers

Use tab completion for more!

000000000

From test to large-scale analysis: General approach

1. Take a single string and test your idea

```
t = "This is a test test test."
print(t.count("test"))
```

2a. You'd assume it to return 3. If so, scale it up:

```
results = []
   for t in listwithallmytexts:
       r = t.count("test")
3
       print(f"{t} contains the substring {r} times")
       results.append(r)
5
```

2b. If you *only* need to get the list of results, a list comprehension is more elegant:

```
results = [t.count("test") for t in listwithallmytexts]
```

000000000

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

General approach

Basic string operations

000000000

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:

```
def mycleanup(t):
      # do sth with string t here, create new string t2
     return t2
3
4
   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:

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

Basic string operations

00000000

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:

df['newcoloumnwithresults'] = 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.

Regular expressions

Regular expressions

What is a regexp?

Regular Expressions: What and why?

What is a regexp?

- a very widespread way to describe patterns in strings
- Think of wildcards like * or operators like OR, AND or NOT in
- You can use them in many editors (!), in the Terminal, in

Regular Expressions: What and why?

What is a regexp?

- a very widespread way to describe patterns in strings
- Think of wildcards like * or operators like OR, AND or NOT in search strings: a regexp does the same, but is *much* more powerful
- You can use them in many editors (!), in the Terminal, in

Regular Expressions: What and why?

What is a regexp?

- a very widespread way to describe patterns in strings
- Think of wildcards like * or operators like OR, AND or NOT in search strings: a regexp does the same, but is *much* more powerful
- You can use them in many editors (!), in the Terminal, in STATA ... and in Python

A more powerful tool

An example

- We want to remove everything but words from a tweet
- We can do so by calling the .replace() method multiple times (for each unwanted character)
- We can do so with a join+list comprehension: "".join([c for c in tweet if c not in listwithunwantedcharacters])
- But we can also use a regular expression instead: [^a-zA-Z] matches anything that is not a letter

Basic regexp elements

Alternatives

Basic string operations

[TtFf] matches either T or t or F or f

Twitter|Facebook matches either Twitter or Facebook

. matches any character

Repetition

- ? the expression before occurs 0 or 1 times
- * the expression before occurs 0 or more times
- + the expression before occurs 1 or more times

Basic regexp elements

Alternatives

Basic string operations

[TtFf] matches either T or t or F or f

Twitter|Facebook matches either Twitter or Facebook

. matches any character

Repetition

- ? the expression before occurs 0 or 1 times
- * the expression before occurs 0 or more times
- + the expression before occurs 1 or more times

regexp quizz

Basic string operations

Which words would be matched?

- 1. [Pp]ython

regexp quizz

Basic string operations

Which words would be matched?

- 1. [Pp]ython
- 2. [A-Z] +

regexp quizz

Basic string operations

Which words would be matched?

- 1. [Pp]ython
- 2. [A-Z] +
- 3. RT ?:? @[a-zA-Z0-9]+

What else is possible?

Basic string operations

See the table in the book!

Regular expressions

Using a regexp in Python

How to use regular expressions in Python

The module re*

Basic string operations

- re.findall("[Tt]witter|[Ff]acebook",testo) returns a list with all occurances of Twitter or Facebook in the string called testo
- re.findall("[0-9]+[a-zA-Z]+",testo) returns a list with all words that start with one or more numbers followed by one or more letters in the string called testo
- re.sub("[Tt]witter|[Ff]acebook","a social medium",testo)
 returns a string in which all all occurances of Twitter
 or Facebook are replaced by "a social medium"

Use the less-known but more powerful module regex instead to support all dialects used in the book

How to use regular expressions in Python

The module re*

Basic string operations

- re.findall("[Tt]witter|[Ff]acebook",testo) returns a list with all occurances of Twitter or Facebook in the string called testo
- re.findall("[0-9]+[a-zA-Z]+",testo) returns a list with all words that start with one or more numbers followed by one or more letters in the string called testo
- re.sub("[Tt]witter|[Ff]acebook","a social medium",testo)
 returns a string in which all all occurances of Twitter
 or Facebook are replaced by "a social medium"

Use the less-known but more powerful module regex instead to support all dialects used in the book

How to use regular expressions in Python

The module re

```
re.match(" +([0-9]+) of ([0-9]+) points", line) returns
            None unless it exactly matches the string line. If it
            does, you can access the part between () with the
             .group() method.
```

Example:

```
line="
                    2 of 25 points"
1
   result=re.match(" +([0-9]+) of ([0-9]+) points",line)
   if result:
3
     print ("Your points:",result.group(1))
      print ("Maximum points:",result.group(2))
```

Your points: 2

Maximum points: 25

Possible applications

Basic string operations

Data preprocessing

- Remove unwanted characters, words, ...
- Identify *meaningful* bits of text: usernames, headlines, where an article starts, ...
- filter (distinguish relevant from irrelevant cases)

Possible applications

Basic string operations

Data analysis: Automated coding

- Actors
- Brands
- links or other markers that follow a regular pattern
- Numbers (!)

Example 1: Counting actors

```
import re, csv
     from glob import glob
     counts1=[]
3
     counts2=[]
4
     filenames = glob("/home/felicia/articles/*.txt")
5
6
     for fn in filenames:
        with open(fn) as fi:
8
9
           artikel = fi.read()
           artikel = artikel.replace('\n',' ')
10
11
12
               counts1.append(len(re.findall('Israel.*(minister|politician.*|[Aa]ut.
           counts2.append(len(re.findall('[Pp]alest',artikel)))
13
14
     output=zip(filenames, counts1, counts2)
15
     with open("results.csv", mode='w',encoding="utf-8") as fo:
16
         writer = csv.writer(fo)
17
         writer.writerows(output)
18
```

Example 2: Parsing semi-structured data

If your data look like this, you can loop over the lines and use regular expressions to extract the info you need!

```
All Rights Reserved
1
2
                                 2 of 200 DOCUMENTS
3
5
                                   De Telegraaf
6
7
                               21 maart 2014 vrijdag
8
    Brussel bereikt akkoord aanpak probleembanken;
10
    ECB krijgt meer in melk te brokkelen
11
    SECTION: Finance: Blz. 24
12
    LENGTH: 660 woorden
13
14
    BRUSSEL Europa heeft gisteren op de valreep een akkoord bereikt
15
    over een saneringsfonds voor banken. Daarmee staat de laatste
16
```

Practice yourself!

Basic string operations

Take some time to write some regular expressions. Write a script that

- extracts URLS form a list of strings
- removes everything that is not a letter or number from a list of strings

(first develop it for a single string, then scale up)

More tips: http://www.pyregex.com/

The BOW

The BOW

General idea

A text as a collections of word

Let us represent a string

t = "This this is is a test test test"

like this:

Basic string operations

- from collections import Counter
- print(Counter(t.split()))

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

The BOW

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

A text as a collections of word

Let us represent a string

```
t = "This this is is a test test test"
```

The BOW

like this:

```
from collections import Counter
print(Counter(t.split()))
```

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 (!!!)

From vector to matrix

Basic string operations

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

The BOW

t1 = "This this is is a test test test"

t2 = "This is an example"

	а	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?

Basic string operations

The cell entries: raw counts versus tf-idf scores

The BOW

• In the example, we entered simple counts (the "term frequency")



But are all terms equally important?

Basic string operations

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

Basic string operations

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")

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")
- \Rightarrow 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)

Is tf-idf always better?

It depends.

Basic string operations

• Ultimately, it's an empirical question which works better (\rightarrow weeks on machine learning)

The BOW

- 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

Internal representations

Basic string operations

Sparse vs dense matrices

- Most terms are not not contained in a given document
- ullet 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 (!!!)

The BOW

- Solution: store only non-zero values with their coordinates! (sparse matrix)
- dense matrix (or dataframes) not advisable, only for toy examples

Internal representations

Little over-generalizing R vs Python remark ;-)

Among many R users, it is common to manually inspect document-term matrices, and many operations are done directly on them. In Python, they are more commonly seen as a means to an end (mostly, as input for machine learning).

Many R modules¹ convert to dense matrices: really problematic for larger datasets!

¹Things have become a bit better recently

The BOW

A cleaner BOW representation

Room for improvement

Basic string operations

tokenization How do we (best) split a sentence into tokens (terms, words)?

pruning How can we remove unneccessary words?

lemmatization How can we make sure that slight variations of the same word are not counted differently?

.split()

- space → new word
- no further processing whatsoever
- thus, only works well if we do a preprocessing outselves (e.g., remove punctuation)

```
docs = ["This is a text", "I haven't seen John's derring-do. Second
    sentence!"]
```

- tokens = [d.split() for d in docs]
- [['This', 'is', 'a', 'text'], ['I', "haven't", 'seen', "John's", 'derring-do.', 'Second', ' sentence!']]

Basic string operations

Tokenizers from the NLTK pacakge

- multiple improved tokenizers that can be used instead of .split()
- e.g., Treebank tokenizer:
 - split standard contractions ("don't")
 - deals with punctuation
 - BUT: Assumes lists of sentences.
- Solution: Build an own (combined) tokenizer (next slide)!

```
import nltk
     import regex
3
     class MyTokenizer:
         def tokenize(self, text):
5
              tokenizer = nltk.tokenize.TreebankWordTokenizer()
              result = []
              word = r"\p{letter}"
8
              for sent in nltk.sent_tokenize(text):
9
                  tokens = tokenizer.tokenize(sent)
10
11
                  tokens = [t for t in tokens
                            if regex.search(word, t)]
12
13
                  result += tokens
             return result
14
15
     mytokenizer = MyTokenizer()
16
     tokens = [mytokenizer.tokenize(d) for d in docs]
17
18
```



Can you (try to) explain the code?

Basic string operations

scikit-learn's CountVectorizer (default settings)

- applies lowercasing
- deals with punctuation etc. itself
- minimum word length > 1
- more technically, tokenizes using this regular expression: $r''(?11) b w + b''^{2}$

```
from sklearn.feature_extraction.text import CountVectorizer
```

- cv = CountVectorizer()
- dtm sparse = cv.fit transform(docs)

 $^{^{2}}$?u = support unicode, b =word boundary

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 the custom NLTK-based external tokenizer we created before! cv = CountVectorizer(tokenizer=mytokenizer.tokenize)

CountVectorizer supports more

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

see

Basic string operations

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

Best of both worlds

Use the Count vectorizer with the custom NLTK-based external tokenizer we created before! cv = CountVectorizer(tokenizer=mytokenizer.tokenize)

Stopword removal

Basic string operations

What are stopwords?

- Very frequent words with little inherent meaning
- the, a, he, she, ...
- context-dependent: if you are interested in gender, he and she are no stopwords.
- Many existing lists as basis

When using the CountVectorizer, we can simply provide a stopword list.

But we can also remove stopwords "by hand" of course using either a for loop (like we did for punctuation removal) or by modifying the tokennizer (try it!).

- Idea behind both stopword removal and tf-idf: too frequent words are uninformative
- (possible) downside stopword removal: a priori list, does not
- (possible) downside tf-idf: does not reduce number of features

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

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

• Idea behind both stopword removal and tf-idf: too frequent words are uninformative

The BOW

- (possible) downside stopword 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 stopword removal

- - myvectorizer = CountVectorizer(stop_words=mystopwords)

CountVectorizer, other tokenization, stopword removal (pay attention that stopword list uses same tokenization!):

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

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

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

All togehter: tf-idf, explicit stopword removal, pruning

```
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?

Stemming and lemmatization

- ullet Stemming: reduce words to its stem by removing last part (drinking o drink)
- Lemmatization: find word that you would need to look up in a dictionary (drinking \rightarrow drink, but also went \rightarrow go)
- stemming is simpler than lemmatization
- lemmatization often better

Example below: tokenization and lemmatization with spacy in one go:

- 1 import spacy
- 2 nlp = spacy.load('en') # potentially you need to install the language model first

Stemming and lemmatization

- Stemming: reduce words to its stem by removing last part (drinking → drink)
- Lemmatization: find word that you would need to look up in a dictionary (drinking → drink, but also went → go)
- stemming is simpler than lemmatization
- lemmatization often better

Example below: tokenization and lemmatization with spacy in one go:

- 1 import spacy
- 2 nlp = spacy.load('en') # potentially you need to install the language model first
- 3 lemmatized_tokens = [[token.lemma_ for token in nlp(doc)] for doc in docs]

The BOW

The order of preprocessing steps

Option 1

Preprocessing only through Vectorizer

"Just use CountVectorizer or Tfidfvectorizer with the appropriate options."

- pro: No double work, efficient if your main goal is a sparse matrix (for ML?) anyway
- con: you cannot "see" the preprocessed texts

Option 2

Basic string operations

Extensive preprocessing without Vectorizer

"Remove stopwords, punctuation etc. and store in a string with spaces"

```
cleaneddocs = [" ".join(re.findall(r"\w\w+", d)).lower() for d in docs]
cleaneddocswithoutstopwords = [" ".join([w for w in d.split() if w not
    in mystopwords]) for d in cleaneddocs]
```

```
['this is text', 'haven seen john derring do second sentence']
['text', 'seen john derring second sentence']
```

Yes, this list comprehension looks scary - you can make a more elaborate for loop instead

- pro: you can read (and store!) the preprocessed docs
- pro: even the most stupid vectorizer (or wordcloud tool) can split the resulting string later on
- con: potentially double work (for you and the computer)



How would you do it?

Sometimes, I go for Option 2 because

- I like to inspect a sample of the documents
- I can re-use the cleaned docs irrespective of the Vectorizer

But at other times, I opt of Option 1 instead because

- I want to systematically compare the effect of different choices in a machine learning pipeline (then I can simply vary the vectorizer instead of the data)
- I want to use techniques that are geared towards little or no preprocessing (deep learning)

The BOW

How further?

Main takeaway

Basic string operations

- It matters how you transform your text into numbers ("vectorization").
- Preprocessing matters, be able to make informed choices.
- Keep this in mind when we will discuss Machine Learning! It will come back throughout Part II!
- Once you vectorized your texts, you can do all kinds of calculations (random example: get the cosine similarity between two texts)

Basic string operations

n-**grams** Consider using *n*-grams instead of unigrams collocations ngrams that appear more frequently than expected POS-tagging grammatical function ("part-of-speach") of tokens **NER** named entity recognition (persons, organizations, locations)

The BOW

More NLP

Basic string operations

I really recommend looking into spacy (https://spacy.io) for advanced natural language processing, such as part-of-speech-tagging and named entity recognition.

Next steps

Basic string operations

Make sure you understood all of today's concepts.

Re-read the chapters.

I prepared exercises to work on (alone or in teams):

https://github.com/uvacw/teaching-bdaca/blob/main/6ec-course/week04/exercises/
Take-home exam on Monday 2th (after class). The answer sheets and all files have to be handed in no

later than Thursday evening (5–10, 23.59)