

A Practical Introduction to Machine Learning in Python

Day 1 - Monday afternoon

»From text to features«

Damian Trilling
Anne Kroon

d.c.trilling@uva.nl, @damian0604
a.c.kroon@uva.nl, @annekroon

September 25, 2023

Today

From text to features: vectorizers

General idea

Pruning

From text to features: vectorizers

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"
```

like this:

```
1 from collections import Counter
2 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)

A text as a collections of word

Let us represent a string

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

like this:

```
1 from collections import Counter
2 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 (!!!)

From vector to matrix

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

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

`t2 = "This is an example"`

	a	an	example	is	this	This	test
<i>t1</i>	1	0	0	3	1	1	3
<i>t2</i>	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

What is a vectorizer

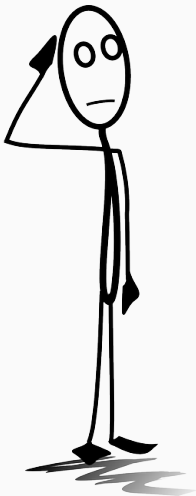
- Transforms a list of texts into a sparse (!) matrix (of word frequencies)
- Vectorizer needs to be “fitted” to the training data (learn which words (features) exist in the dataset and assign them to columns in the matrix)
- Vectorizer can then be re-used to transform other datasets

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)

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)

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"))

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

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://mattedging.github.io/2019/04/25/sparse-matrices/>

Tomorrow we discuss how we can tokenize with a list comprehension (and that's often a good idea!). 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)

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

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

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

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?