A Practical Introduction to Machine Learning in Python
Day 1 - Monday afternoon
»From text to features«

Rupert Kiddle
Marieke van Hoof

r.t.kiddle@vu.nl, @rptkiddle
m.vanhoof@uva.nl, @marieke_vh

September 17, 2024

Gesis

Today

From text to features: vectorizers

From text to features: vectorizers

From text to features: vectorizers

General idea

Bag Of Words (BOW): A text as a collections of words

Let us represent a string

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

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 (!!!)

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

Bag Of Words (BOW): A text as a collections of words

Let us represent a string

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

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 (!!!)

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

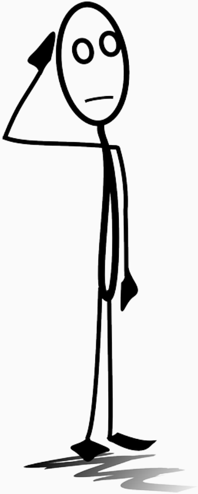
From vector to matrix: Document Term Matrix (DTM)

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
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

What is a vectorizer

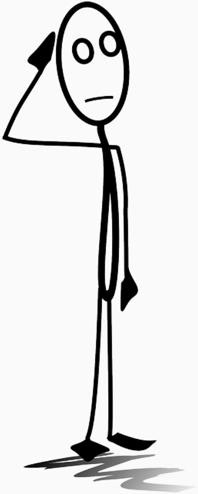
- Transforms a list of texts into a sparse (!) matrix (of word frequencies)
- Vectorizer needs to be "fitted" to the training data (learn which words (features) exist in the dataset and assign them to columns in the matrix)
- Vectorizer can then be re-used to transform other datasets

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")*

$\Rightarrow$ we multiply the "term frequency" (tf) by the inverse document frequency (idf)

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)

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)

tf·idf

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

$$\begin{aligned} tf_{i,j} &= \text{ number of occurrences of } i \text{ in } j \\ df_i &= \text{ number of documents containing } i \\ N &= \text{total number of documents} \end{aligned}$$

Is tf·idf always better?

It depends.

- 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
- Ultimately, it's an empirical question which works better ($\rightarrow$ machine learning)

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"))

Vectorizers take care of...

1. Tokenization: Splitting sentences into tokens (terms, words)
2. Vocabulary: Create a list of unique tokens in the corpus
3. Vectorization: For each document, assigned term frequencies or tf·idf scores

Vectorizers take care of...

1. Tokenization: Splitting sentences into tokens (terms, words)
2. Vocabulary: Create a list of unique tokens in the corpus
3. Vectorization: For each document, assigned term frequencies or tf·idf scores

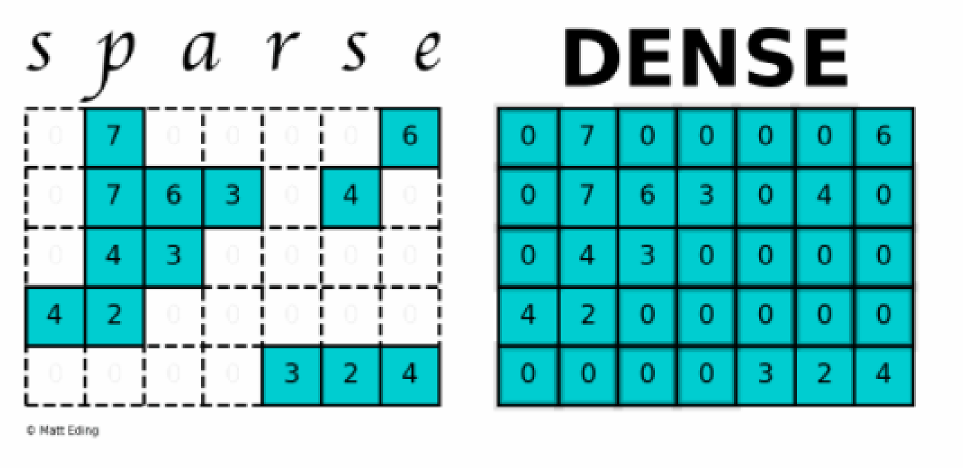
Vectorizers take care of...

1. Tokenization: Splitting sentences into tokens (terms, words)
2. Vocabulary: Create a list of unique tokens in the corpus
3. Vectorization: For each document, assigned term frequencies or tf·idf scores

Internal representations

Sparse vs dense matrices

- $\rightarrow$ 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



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

Room for improvement?

- Tomorrow we discuss how we can tokenize with list comprehensions (and that's often a good idea!)
- But if you want to directly get a DTM instead of a list of tokens you can also achieve a much cleaner representation directly with vectorizers

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]

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
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 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

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

General idea

- 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

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

General idea

- 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

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

General idea

- 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

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

CountVectorizer, only stopword removal

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

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

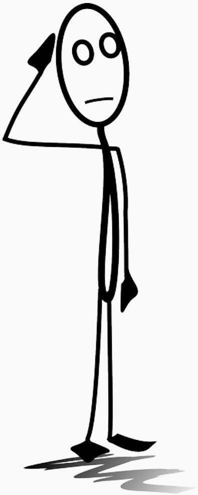
```
myvectorizer = CountVectorizer(tokenizer = TreebankWordTokenizer().
    tokenize, stop_words=mystopwords)
```

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 together: 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?