Introduction to Data Science

TEXT FEATURE EXTRACTION

BRIAN D'ALESSANDRO

TEXT FEATURE STRATEGIES

We’ll cover the two most common text feature engineering paths in use today.

Tokenization	Embedding
Convert words (or word combos) to individual features with either a binary or frequency based value.	Convert words (or word combos) to a dense vector representation from a pre-trained separate process
Highly scalable and easy to use but can suffer from sparsity issues.	Reduces interpretability but is often more robust. Scalable with pre-trained embedding models

TOKENIZING

A very straightforward and scalable way to deal with text is to convert individual words to tokens, and let each token be a separate feature in the data.

Original Data Representation

Doc	Phrase
1	the cow jumped over the moon
2	somewhere over the rainbow
3	over the moon

Feature Dictionary

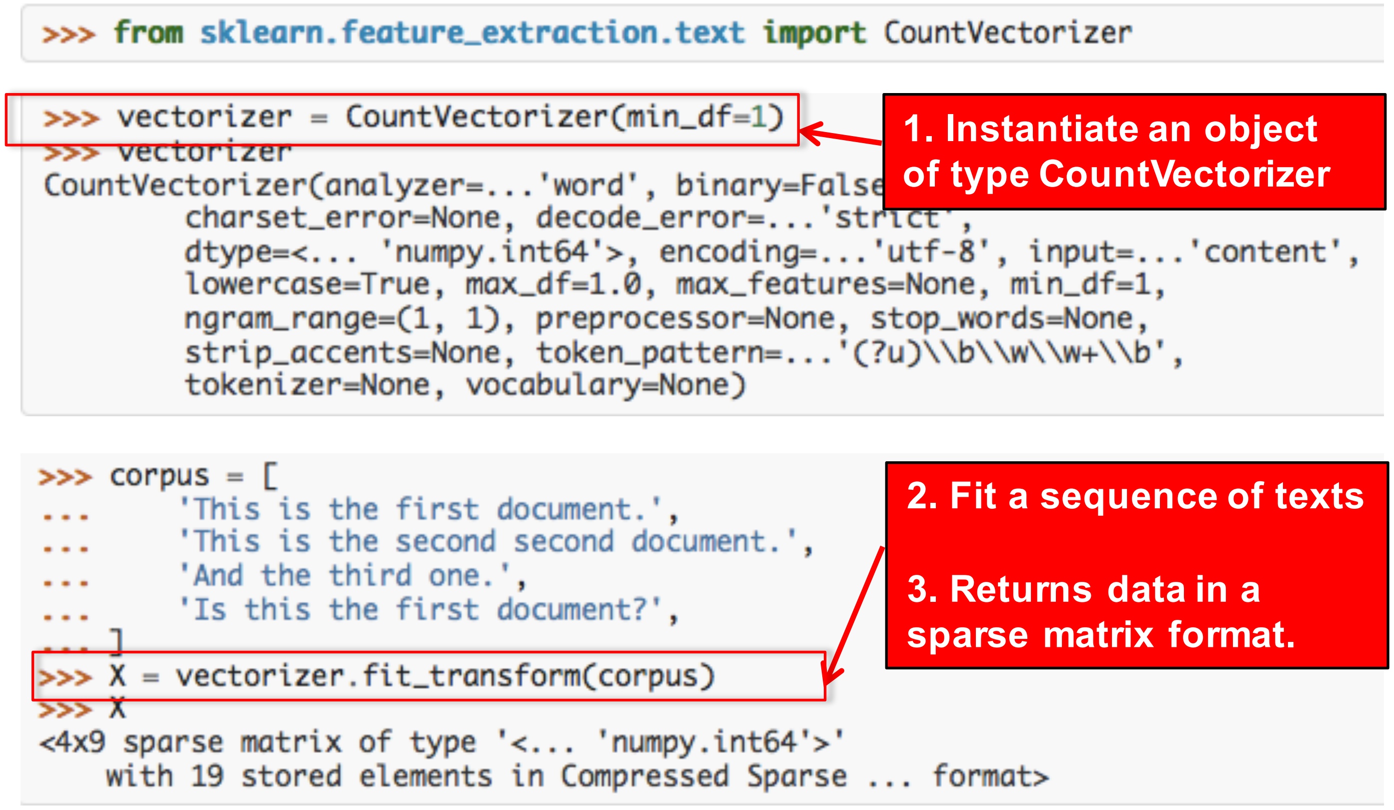
Token	FID
cow	1
jumped	2
moon	3
over	4
rainbow	5
somew here	6
the	7

Tokenized Data Representation (Binary Representation)

	FID						
Doc	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	1	1	1	0	0	1
2	0	0	0	1	1	1	1
3	0	0	1	1	0	0	1

TOKENIZING - PYTHON

http://scikit-learn.org/stable/modules/feature_extraction.html



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

```
>>> vectorizer = CountVectorizer(min_df=1)
>>> vectorizer
CountVectorizer(analyzer=...'word', binary=False
        charset_error=None, decode_error=...'strict',
        dtype=<... 'numpy.int64'>, encoding=...'utf-8', input=...'content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern=...'(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

1. Instantiate an object of type CountVectorizer

```
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third one.',
...     'Is this the first document?',
... ]
>>> X = vectorizer.fit_transform(corpus)
>>> X
<4x9 sparse matrix of type '<... 'numpy.int64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

2. Fit a sequence of texts

3. Returns data in a sparse matrix format.

SPARSE FORMAT

By default, sklearn.feature_extraction.text returns matrices in “sparse” format. In many applications, especially text, there are many features but each instance only has a relatively few non-zero values. We can define a format that does not explicitly store zeros to save space.

Dense Format

	FID						
Doc	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	1	1	1	1	0	0	1
2	0	0	0	1	1	1	1
3	0	0	1	1	0	0	1

Sparse Format (Explicit Values)

Doc	Sparse Features
1	1:1 2:1 3:1 4:1 7:1
2	4:1 5:1 6:1 7:1
3	3:1 4:1 7:1

Sparse Format (Implicit Values)

Doc	Sparse Features
1	1 2 3 4 7
2	4 5 6 7
3	3 4 7

EXTENSIONS - NGRAMS

To gain more flexibility in our feature representation, we might want to tokenize combinations of words, as opposed to the words themselves. Often times word combinations can express more nuance, such as when negative sentiment is being expressd (i.e., “I don’t like”)

“I like rainbows. They look nice.” → *[(“I like”), (“like rainbows”), …, (“They look”), (“look nice”)]*

Each n-gram can then be considered an individual feature.

To build n-grams in sklearn:

```
>>> vect = CountVectorizer(min_df=1, analyzer='char', ngram_range=(1,2))
>>> X = vect.fit_transform(corpus)
```

EXTENSIONS – TF/IDF

In many cases we want more than binary indicators that a particular word/ngram is present in the document. One popular method, TF/IDF, assigns a heuristic importance weight of each word/ngram to each document.

$$TF - IDF = Term\_Freq \;\; x \;\; Inverse\_Document\_Freq$$

Term frequency measures how many times a particular token appears in a particular document, and gives an indication of how important that word is to that document.

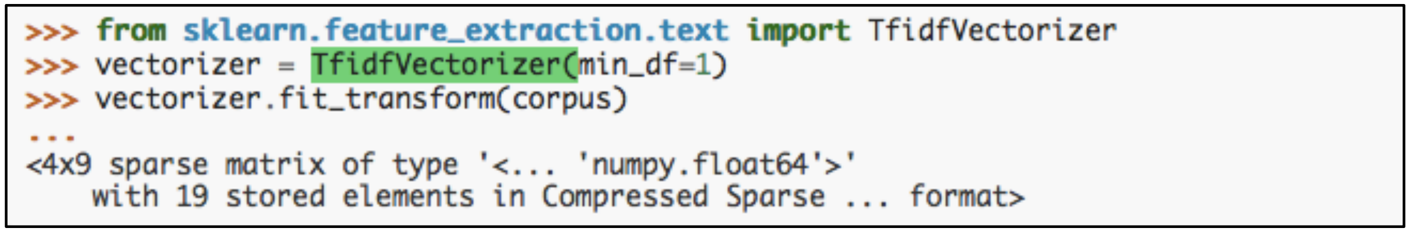
$$TF(term,\ Doc) = \sum_{word \in Doc} \mathbb{I}(word == term)$$

Inverse document frequency measures how often a word appears across all documents. It produces a measure that penalizes words that appear frequently.

$$IDF(term,\ Corpus) = log \frac{|Corpus|}{\sum_{Doc \in Corp} \mathbb{I}(term \in Doc)}$$

TF/IDF - SKLEARN

TF-IDF can be implemented directly with tokenization using the sklearn.feature_extraction.text.TfidfVectorizer class.



```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> vectorizer = TfidfVectorizer(min_df=1)
>>> vectorizer.fit_transform(corpus)
...
<4x9 sparse matrix of type '<... 'numpy.float64'>'
    with 19 stored elements in Compressed Sparse ... format>
```

Variations:

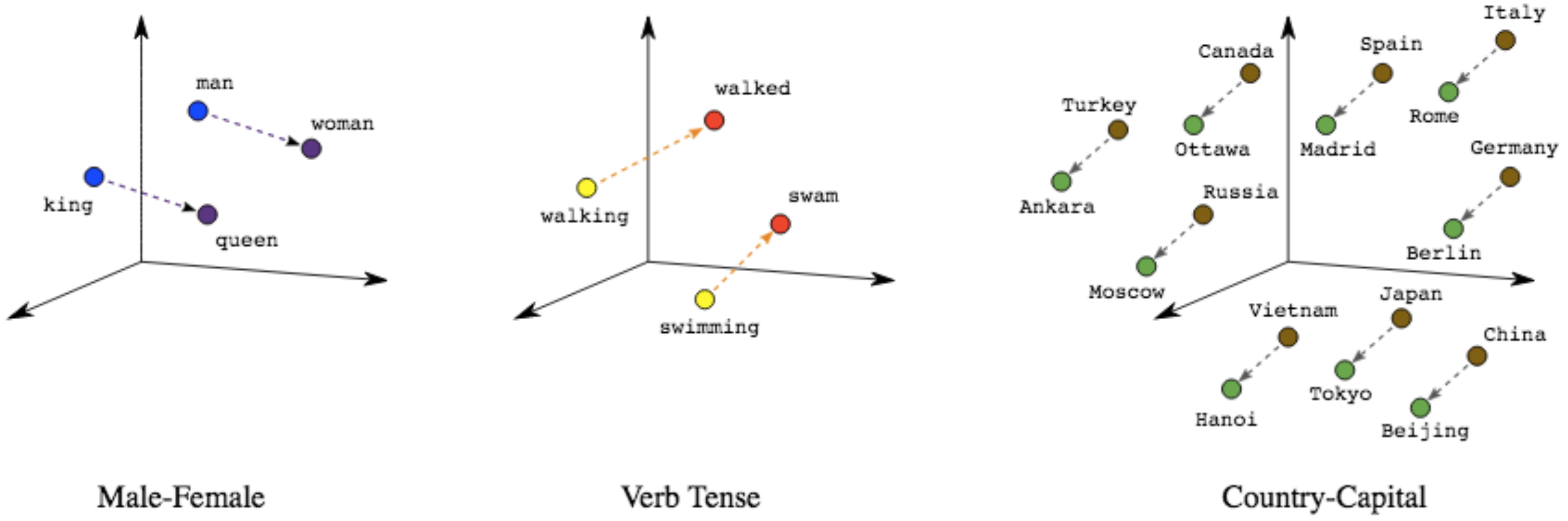
- TF-IDF row vectors can be rescaled to have unit norms
- TF can be calculating using a binary indicator of (freq>0) or log(freg)
- IDF can be augmented to prevent division by zero

The optimality of TF-IDF is of course an empirical question. Testing is always recommended!

WORD EMBEDDINGS

The idea: Represent a word by a dense vector such that words with similar meaning or semantic uses have vectors that are near each other in the vector space (often measured by Cosine Similarity)

f("word") = <0.1, -0.9, 2.3, …., 1.2, 0.6>



Picture source:https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space

WAYS TO GET EMBEDDINGS

The word “embedding” is most often associated with word2vec (and related methods), but there are multiple techniques suited for learning dense vector representations of words.

Latent Semantic Indexing (LSI)

Latent Dirichlet Allocation (LDA)

Word2vec (W2V)

Recurrent Neural Nets (RNN)

More classic techniques:

LSI builds a term x document matrix and performs SVD on the matrix. The resulting singular vectors are used as the word embedding.

LDA is a generative process that infers latent topics from documents. We assume a document is a multinomial distribution over topics, and topics are multinomial distributions over words. The resulting document and word distributions can be used as embeddings

More modern techniques:

W2V uses a shallow neural architecture to model the likelihood of observing a word given its neighboring words. The NNs internal weights are used as the word embedding.

RNN based approaches model the text sequentially against a supervised learning task. The hidden or output vectors of the NN can be used a word embedding

Very comprehensive overview: https://towardsdatascience.com/document-embedding-techniques-fed3e7a6a25d

PRACTICAL WORD2VEC EXAMPLE

For most text problems we can work quickly by using pre-trained embeddings (available in most text processing packages). There are more sophisticated techniques for generating an instance/document level embedding, but simply averaging individual word embeddings is a proven simple approach.

1. For each word in an example, lookup associated w2v vector
2. Perform some aggregation over the example's word vectors to get to a document representation
3. Use the results of step 2 as the document's feature vector

$$\begin{array}{c} D \\ \begin{bmatrix} \dfrac{W_{11}+W_{21}+\ldots+W_{n1}}{n} \\ \\ \\ \\ \dfrac{W_{1n}+W_{2n}+\ldots+W_{nn}}{n} \end{bmatrix} \end{array}$$

https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf