Feature Engineering for NLP in Python

Thursday, 29 October 2020 2:51 PM

Basic features and readability scores

Numerical data

Iris dataset

sepal length	sepal width	petal length	petal width	class
6.3	2.9	5.6	1.8	Iris-virginica
4.9	3.0	1.4	0.2	Iris-setosa
5.6	2.9	3.6	1.3	Iris-versicolor
6.0	2.7	5.1	1.6	Iris-versicolor
7.2	3.6	6.1	2.5	Iris-virginica

One-hot encoding

sex	one-hot encoding	sex_female	sex_male
female	→	1	0
male	→	0	1
female	→	1	0
male	→	0	1
female	→	1	0
...	...	...	...

One-hot encoding with pandas

```
# Import the pandas library
import pandas as pd

# Perform one-hot encoding on the 'sex' feature of df
df = pd.get_dummies(df, columns=['sex'])
```

Textual data

Movie Review Dataset

review	class
This movie is for dog lovers. A very poignant...	positive
The movie is forgettable. The plot lacked...	negative
A truly amazing movie about dogs. A gripping...	positive

Text pre-processing

- Converting to lowercase
  - **Example:** `Reduction` to `reduction`
- Converting to base-form
  - **Example:** `reduction` to `reduce`

Vectorization

review	class
This movie is for dog lovers. A very poignant...	positive
The movie is forgettable. The plot lacked...	negative
A truly amazing movie about dogs. A gripping...	positive

Vectorization

0	1	2	...	n	class
0.03	0.71	0.00	...	0.22	positive
0.45	0.00	0.03	...	0.19	negative
0.14	0.18	0.00	...	0.45	positive

Basic features

- Number of words
- Number of characters
- Average length of words
- Tweets

> **Silverado Records** @SilveradoLabel
> What Country music is everyone listening to today?
>
> #countrymusic #silveradorecords

POS tagging

Word	POS
I	Pronoun
have	Verb
a	Article
dog	Noun

Named Entity Recognition

- Does noun refer to person, organization or country?

TED

Noun	NER
Brian	**Person**
DataCamp	**Organization**

Concepts covered

- Text Preprocessing
- Basic Features
- Word Features
- Vectorization

Number of characters

```
"I don't know." # 13 characters
```

```
# Compute the number of characters
text = "I don't know."
num_char = len(text)

# Print the number of characters
print(num_char)
```

```
13
```

```
# Create a 'num_chars' feature
df['num_chars'] = df['review'].apply(len)
```

Number of words

```
# Split the string into words
text = "Mary had a little lamb."
words = text.split()

# Print the list containing words
print(words)
```

```
['Mary', 'had', 'a', 'little', 'lamb.']
```

```
# Print number of words
print(len(words))
```

```
5
```

Number of words

```
# Function that returns number of words in string
def word_count(string):
    # Split the string into words
    words = string.split()

    # Return length of words list
    return len(words)
```

```
# Create num_words feature in df
df['num_words'] = df['review'].apply(word_count)
```

Average word length

```
#Function that returns average word length
def avg_word_length(x):
    # Split the string into words
    words = x.split()
    # Compute length of each word and store in a separate list
    word_lengths = [len(word) for word in words]
    # Compute average word length
    avg_word_length = sum(word_lengths)/len(words)
    # Return average word length
    return(avg_word_length)
```

Average word length

```
# Create a new feature avg_word_length
df['avg_word_length'] = df['review'].apply(doc_density)
```

Special features

Hashtags and mentions

```
# Function that returns number of hashtags
def hashtag_count(string):
    # Split the string into words
    words = string.split()
    # Create a list of hashtags
    hashtags = [word for word in words if word.startswith('#')]
    # Return number of hashtags
    return len(hashtags)
```

```
hashtag_count("@janedoe This is my first tweet! #FirstTweet #Happy")
```

```
2
```

Other features

- Number of sentences
- Number of paragraphs
- Words starting with an uppercase
- All-capital words
- Numeric quantities

Readability test examples

- **Flesch reading ease**
- **Gunning fog index**
- Simple Measure of Gobbledygook (SMOG)
- Dale-Chall score

Flesch reading ease

- One of the oldest and most widely used tests
- Dependent on two factors:
- **Greater the average sentence length, harder the text is to read**
  - "This is a short sentence."
  - "This is longer sentence with more words and it is harder to follow than the first sentence."
- **Greater the average number of syllables in a word, harder the text is to read**
  - "I live in my home."
  - "I reside in my domicile."
- Higher the score, greater the readability

Flesch reading ease score interpretation

Reading ease score	Grade Level
90-100	5
80-90	6
70-80	7
60-70	8-9
50-60	10-12
30-50	College
0-30	College Graduate

The textatistic library

```
# Import the Textatistic class
from textatistic import Textatistic

# Create a Textatistic Object
readability_scores = Textatistic(text).scores

# Generate scores
print(readability_scores['flesch_score'])
print(readability_scores['gunningfog_score'])
```

```
21.14
16.26
```

Text sources

- News articles
- Tweets
- Comments

Making text machine friendly

- `Dogs` , `dog`
- `reduction` , `REDUCING` , `Reduce`
- `don't` , `do not`
- `won't` , `will not`

Text preprocessing techniques

- Converting words into lowercase
- Removing leading and trailing whitespaces
- Removing punctuation
- Removing stopwords
- Expanding contractions
- Removing special characters (numbers, emojis, etc.)

Tokenization

```
"I have a dog. His name is Hachi."
```

Tokens:

```
["I", "have", "a", "dog", ".", "His", "name", "is", "Hachi", "."]
```

```
"Don't do this."
```

Tokens:

```
["Do", "n't", "do", "this", "."]
```

Tokenization using spaCy

```
import spacy
# Load the en_core_web_sm model
nlp = spacy.load('en_core_web_sm')
# Initiliaze string
string = "Hello! I don't know what I'm doing here."
# Create a Doc object
doc = nlp(string)
# Generate list of tokens
tokens = [token.text for token in doc]
print(tokens)
```

```
['Hello','!','I','do',"n't",'know','what','I',"'m",'doing','here','.']
```

Lemmatization

- Convert word into its base form
  - `reducing` , `reduces` , `reduced` , `reduction` → `reduce`
  - `am` , `are` , `is` → `be`
  - `n't` → `not`
  - `'ve` → `have`

Lemmatization using spaCy

```
import spacy

# Load the en_core_web_sm model
nlp = spacy.load('en_core_web_sm')
# Initiliaze string
string = "Hello! I don't know what I'm doing here."
# Create a Doc object
doc = nlp(string)

# Generate list of lemmas
lemmas = [token.lemma_ for token in doc]
print(lemmas)
```

```
['hello','!','-PRON-','do','not','know','what','-PRON','be','do','here', '.']
```

Text cleaning techniques

- Unnecessary whitespaces and escape sequences
- Punctuations
- Special characters (numbers, emojis, etc.)
- Stopwords

isalpha()

```
"Dog".isalpha()
```

```
True
```

```
"!".isalpha()
```

```
False
```

```
"3dogs".isalpha()
```

```
False
```

```
"?".isalpha()
```

```
False
```

```
"12347".isalpha()
```

```
False
```

A word of caution

- Abbreviations: `U.S.A` , `U.K` , etc.
- Proper Nouns: `word2vec` and `xto10x` .
- Write your own custom function (using regex) for the more nuanced cases.

Removing non-alphabetic characters

```
string = """
OMG!!!! This is like    the best thing ever \t\n.
Wow, such an amazing song! I'm hooked. Top 5 definitely. ?
"""
import spacy

# Generate list of tokens
nlp = spacy.load('en_core_web_sm')
doc = nlp(string)
lemmas = [token.lemma_ for token in doc]
```

Removing non-alphabetic characters

```
...
...
# Remove tokens that are not alphabetic
a_lemmas = [lemma for lemma in lemmas
            if lemma.isalpha() or lemma == '-PRON-']

# Print string after text cleaning
print(' '.join(a_lemmas))
```

```
'omg this be like the good thing ever wow such an amazing song -PRON- be hooked top definitely'
```

Stopwords

- Words that occur extremely commonly
- Eg. articles, be verbs, pronouns, etc.

such as a and the, be verbs such as is and am and pronouns such as he and she.

Removing stopwords using spaCy

```
# Get list of stopwords
stopwords = spacy.lang.en.stop_words.STOP_WORDS
string = """
OMG!!!! This is like    the best thing ever \t\n.
Wow, such an amazing song! I'm hooked. Top 5 definitely. ?
"""
```

Removing stopwords using spaCy

```
...
...
# Remove stopwords and non-alphabetic tokens
a_lemmas = [lemma for lemma in lemmas
            if lemma.isalpha() and lemma not in stopwords]
# Print string after text cleaning
print(' '.join(a_lemmas))
```

```
'omg like good thing wow amazing song hooked definitely'
```

Other text preprocessing techniques

- Removing HTML/XML tags
- Replacing accented characters (such as é)
- Correcting spelling errors

A word of caution

Always use only those text preprocessing techniques that are relevant to your application.

Applications

- Word-sense disambiguation
  - `"The bear is a majestic animal"`
  - `"Please bear with me"`
- Sentiment analysis
- Question answering
- Fake news and opinion spam detection

POS tagging

- Assigning every word, its corresponding part of speech.

```
"Jane is an amazing guitarist."
```

- **POS Tagging:**
  - `Jane` → **proper noun**
  - `is` → **verb**
  - `an` → **determiner**
  - `amazing` → **adjective**
  - `guitarist` → **noun**

POS tagging using spaCy

```
import spacy

# Load the en_core_web_sm model
nlp = spacy.load('en_core_web_sm')
```

```
# Initiliaze string
string = "Jane is an amazing guitarist"
```

```
# Create a Doc object
doc = nlp(string)
```

POS tagging using spaCy

```
...
...
# Generate list of tokens and pos tags
pos = [(token.text, token.pos_) for token in doc]
print(pos)
```

```
[('Jane', 'PROPN'),
 ('is', 'VERB'),
 ('an', 'DET'),
 ('amazing', 'ADJ'),
 ('guitarist', 'NOUN')]
```

POS annotations in spaCy

- `PROPN` → proper noun
- `DET` → determinant
- spaCy annotations at **https://spacy.io/api/annotation**

POS	DESCRIPTION	EXAMPLES
ADJ	adjective	big, old, green, incomprehensible, first
ADP	adposition	in, to, during
ADV	adverb	very, tomorrow, down, where, there
AUX	auxiliary	is, has (done), will (do), should (do)
CONJ	conjunction	and, or, but
CCONJ	coordinating conjunction	and, or, but
DET	determiner	a, an, the

Applications

- Efficient search algorithms
- Question answering
- News article classification
- Customer service

Named entity recognition

- Identifying and classifying named entities into predefined categories.
- Categories include person, organization, country, etc.

```
"John Doe is a software engineer working at Google. He lives in France."
```

- **Named Entities**
- `John Doe` → person
- `Google` → organization
- `France` → country (geopolitical entity)

NER using spaCy

```
import spacy
string = "John Doe is a software engineer working at Google. He lives in France."

# Load model and create Doc object
nlp = spacy.load('en_core_web_sm')
doc = nlp(string)

# Generate named entities
ne = [(ent.text, ent.label_) for ent in doc.ents]
print(ne)
```

```
[('John Doe', 'PERSON'), ('Google', 'ORG'), ('France', 'GPE')]
```

NER annotations in spaCy

- More than 15 categories of named entities
- NER annotations at **https://spacy.io/api/annotation#named-entities**

TYPE	DESCRIPTION
PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Companies, agencies, institutions, etc.
GPE	Countries, cities, states.

A word of caution

- Not perfect
- Performance dependent on training and test data
- Train models with specialized data for nuanced cases
- Language specific

Recap of data format for ML algorithms

For any ML algorithm,

- Data must be in tabular form
- Training features must be numerical

Bag of words model

- Extract word tokens
- Compute frequency of word tokens
- Construct a word vector out of these frequencies and vocabulary of corpus

Bag of words model example

Corpus

```
"The lion is the king of the jungle"
```

```
"Lions have lifespans of a decade"
```

```
"The lion is an endangered species"
```

Bag of words model example

Vocabulary → `a` , `an` , `decade` , `endangered` , `have` , `is` , `jungle` , `king` , `lifespans` , `lion` , `Lions` , `of` , `species` , `the` , `The`

```
"The lion is the king of the jungle"
```

```
[0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 2, 1]
```

```
"Lions have lifespans of a decade"
```

```
[1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0]
```

```
"The lion is an endangered species"
```

```
[0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1]
```

Text preprocessing

- `Lions` , `lion` → `lion`
- `The` , `the` → `the`
- No punctuations
- No stopwords
- Leads to smaller vocabularies
- Reducing number of dimensions helps improve performance

Bag of words model using sklearn

```
corpus = pd.Series([
    'The lion is the king of the jungle',
    'Lions have lifespans of a decade',
    'The lion is an endangered species'
])
```

Bag of words model using sklearn

```
# Import CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer
# Create CountVectorizer object
vectorizer = CountVectorizer()
# Generate matrix of word vectors
bow_matrix = vectorizer.fit_transform(corpus)
print(bow_matrix.toarray())
```

```
array([[0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 3],
       [0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0],
       [1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1]], dtype=int64)
```

Spam filtering

message	label
WINNER!! As a valued network customer you have been selected to receive a $900 prize reward! To claim call 09061701461	spam
Ah, work. I vaguely remember that. What does it feel like?	ham

Steps

1. Text preprocessing
2. Building a bag-of-words model (or representation)
3. Machine learning

Text preprocessing using CountVectorizer

CountVectorizer arguments

- `lowercase` : `False` , `True`
- `strip_accents` : `'unciode'` , `'ascii'` , `None`
- `stop_words` : `'english'` , `list` , `None`
- `token_pattern` : `regex`
- `tokenizer` : `function`

CountVectorizer's main job is to convert a corpus into a matrix of numerical vectors.

Building the BoW model

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

# Create CountVectorizer object
vectorizer = CountVectorizer(strip_accents='ascii', stop_words='english', lowercase=False)

# Import train_test_split
from sklearn.model_selection import train_test_split

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(df['message'], df['label'], test_size=0.25)
```

Building the BoW model

```
...
...
# Generate training Bow vectors
X_train_bow = vectorizer.fit_transform(X_train)

# Generate test BoW vectors
X_test_bow = vectorizer.transform(X_test)
```

Training the Naive Bayes classifier

```
# Import MultinomialNB
from sklearn.naive_bayes import MultinomialNB

# Create MultinomialNB object
clf = MultinomialNB()

# Train clf
clf.fit(X_train_bow, y_train)

# Compute accuracy on test set
accuracy = clf.score(X_test_bow, y_test)
print(accuracy)
```

```
0.760051
```

BoW shortcomings

review	label
`'The movie was good and not boring'`	positive
`'The movie was not good and boring'`	negative

- Exactly the same BoW representation!
- Context of the words is lost.
- Sentiment dependent on the position of 'not'.

n-grams

- Contiguous sequence of n elements (or words) in a given document.
- n = 1 → bag-of-words

```
'for you a thousand times over'
```

- n = 2, n-grams:

```
[
'for you',
'you a',
'a thousand',
'thousand times',
'times over'
]
```

n-grams

```
'for you a thousand times over'
```

- n = 3, n-grams:

```
[
'for you a',
'you a thousand',
'a thousand times',
'thousand times over'
]
```

Therefore, we can use these n-grams to capture more context

Applications

- Sentence completion
- Spelling correction
- Machine translation correction

Building n-gram models using scikit-learn

Generates only bigrams.

```
bigrams = CountVectorizer(ngram_range=(2,2))
```

Generates unigrams, bigrams and trigrams.

```
ngrams = CountVectorizer(ngram_range=(1,3))
```

Shortcomings

- Curse of dimensionality
- Higher order n-grams are rare
- Keep n small

TF-IDF and similarity scores

n-gram modeling

- Weight of dimension dependent on the frequency of the word corresponding to the dimension.
  - Document contains the word `human` in five places.
  - Dimension corresponding to `human` has weight `5`.

Motivation

- Some words occur very commonly across all documents
- Corpus of documents on the universe
  - One document has `jupiter` and `universe` occurring 20 times each.
  - `jupiter` rarely occurs in the other documents. `universe` is common.
  - Give more weight to `jupiter` on account of exclusivity.

Applications

- Automatically detect stopwords
- Search
- Recommender systems
- Better performance in predictive modeling for some cases

Term frequency-inverse document frequency

- Proportional to term frequency
- Inverse function of the number of documents in which it occurs

Mathematical formula

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

$$w_{i,j} \rightarrow \text{weight of term } i \text{ in document } j$$

$$tf_{i,j} \rightarrow term\ frequency\ of\ term\ i\ in\ document\ j$$

$$N \rightarrow number\ of\ documents\ in\ the\ corpus$$

$$df_i \rightarrow number\ of\ documents\ cotaining\ term\ i$$

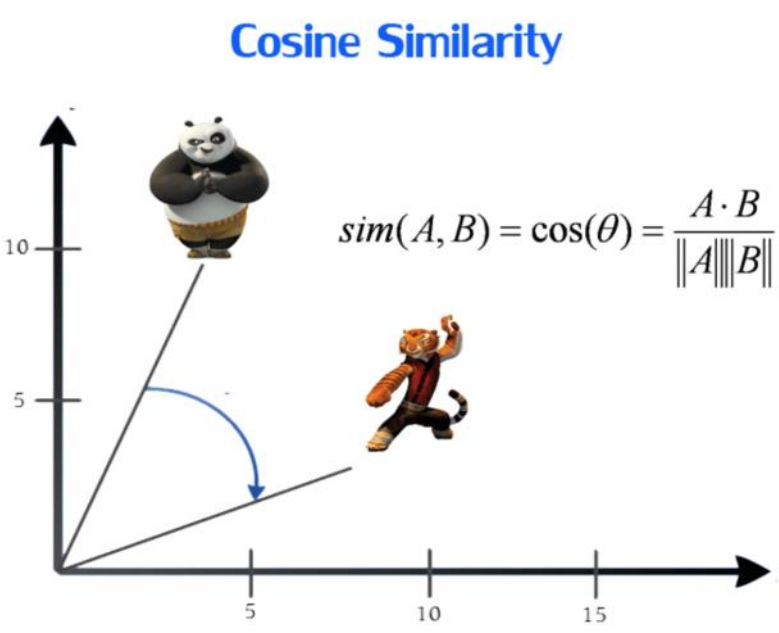
Example:

$$w_{library,document} = 5 \cdot log(\frac{20}{8}) \approx 2$$

tf-idf using scikit-learn

```
# Import TfidfVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
# Create TfidfVectorizer object
vectorizer = TfidfVectorizer()
# Generate matrix of word vectors
tfidf_matrix = vectorizer.fit_transform(corpus)
print(tfidf_matrix.toarray())
```

```
[[0.         0.         0.         0.         0.25434658 0.33443519
  0.33443519 0.         0.25434658 0.         0.25434658 0.
  0.76303975]
 [0.         0.46735098 0.         0.46735098 0.         0.
  0.         0.46735098 0.         0.46735098 0.35543247 0.
  0.        ]
...
```



The dot product

Consider two vectors,

$$V = (v_1, v_2, \cdots, v_n), W = (w_1, w_2, \cdots, w_n)$$

Then the dot product of V and W is,

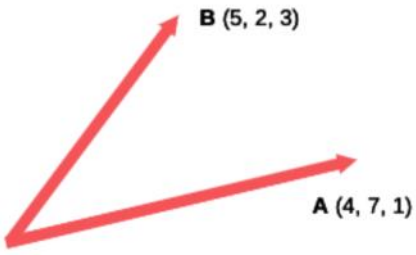
$$V \cdot W = (v_1 \times w_1) + (v_2 \times w_2) + \cdots + (v_n \times w_n)$$

Example:

$$A = (4, 7, 1)\ ,\ B = (5, 2, 3)$$

$$A \cdot B = (4 \times 5) + (7 \times 2) + \cdots (1 \times 3)$$

$$= 20 + 14 + 3 = 37$$



Magnitude of a vector

For any vector,

$$V = (v_1, v_2, \cdots, v_n)$$

The magnitude is defined as,

$$||\mathbf{V}|| = \sqrt{(v_1)^2 + (v_2)^2 + ... + (v_n)^2}$$

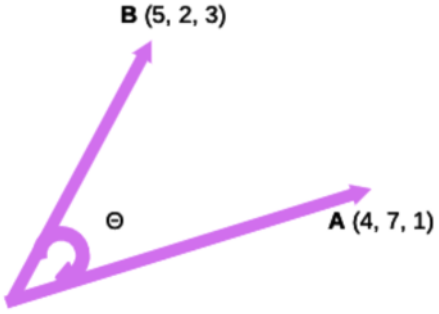
Example:

$$A = (4, 7, 1)\ ,\ B = (5, 2, 3)$$

$$||\mathbf{A}|| = \sqrt{(4)^2 + (7)^2 + (1)^2}$$

$$= \sqrt{16 + 49 + 1} = \sqrt{66}$$

The cosine score



$$A : (4, 7, 1)$$

$$B : (5, 2, 3)$$

The cosine score,

$$cos(A, B) = \frac{A \cdot B}{|A| \cdot |B|}$$

$$= \frac{37}{\sqrt{66} \times \sqrt{38}}$$

$$= 0.7388$$

Cosine Score: points to remember

- Value between -1 and 1.
- In NLP, value between 0 and 1.
- Robust to document length.

Implementation using scikit-learn

```
# Import the cosine_similarity
from sklearn.metrics.pairwise import cosine_similarity
```

```
# Define two 3-dimensional vectors A and B
A = (4,7,1)
B = (5,2,3)

# Compute the cosine score of A and B
score = cosine_similarity([A], [B])

# Print the cosine score
print(score)
```

```
array([[ 0.73881883]])
```

Movie recommender

```
get_recommendations("The Godfather")
```

```
1178                The Godfather: Part II
44030    The Godfather Trilogy: 1972-1990
1914               The Godfather: Part III
23126                           Blood Ties
11297                    Household Saints
34717                   Start Liquidation
10821                            Election
38030                           Goodfellas
17729                   Short Sharp Shock
26293                  Beck 28 - Familjen
Name: title, dtype: object
```

Steps

1. Text preprocessing
2. Generate tf-idf vectors
3. Generate cosine similarity matrix

The recommender function

1. Take a movie title, cosine similarity matrix and indices series as arguments.
2. Extract pairwise cosine similarity scores for the movie.
3. Sort the scores in descending order.
4. Output titles corresponding to the highest scores.
5. Ignore the highest similarity score (of 1).

Generating tf-idf vectors

```
# Import TfidfVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

# Create TfidfVectorizer object
vectorizer = TfidfVectorizer()

# Generate matrix of tf-idf vectors
tfidf_matrix = vectorizer.fit_transform(movie_plots)
```

Generating cosine similarity matrix

```
# Import cosine_similarity
from sklearn.metrics.pairwise import cosine_similarity

# Generate cosine similarity matrix
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)
```

```
array([[1.        , 0.27435345, 0.23092036, ..., 0.        , 0.        ,
        0.00758112],
       [0.27435345, 1.        , 0.1246955 , ..., 0.        , 0.        ,
        0.00740494],
       ...,
       [0.00758112, 0.00740494, 0.        , ..., 0.        , 0.        ,
        1.        ]])
```

The linear_kernel function

- Magnitude of a tf-idf vector is 1
- Cosine score between two tf-idf vectors is their dot product.
- Can significantly improve computation time.
- Use `linear_kernel` instead of `cosine_similarity`.

Generating cosine similarity matrix

```
# Import cosine_similarity
from sklearn.metrics.pairwise import linear_kernel

# Generate cosine similarity matrix
cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)
```

```
array([[1.        , 0.27435345, 0.23092036, ..., 0.        , 0.        ,
        0.00758112],
       [0.27435345, 1.        , 0.1246955 , ..., 0.        , 0.        ,
        0.00740494],
       ...,
       [0.00758112, 0.00740494, 0.        , ..., 0.        , 0.        ,
        1.        ]])
```

The problem with BoW and tf-idf

```
'I am happy'
```

```
'I am joyous'
```

```
'I am sad'
```

Word embeddings

- Mapping words into an n-dimensional vector space
- Produced using deep learning and huge amounts of data
- Discern how similar two words are to each other
- Used to detect synonyms and antonyms
- Captures complex relationships
  - `King - Queen → Man - Woman`
  - `France - Paris → Russia - Moscow`
- Dependent on spacy model; independent of dataset you use

Word embeddings using spaCy

```
import spacy

# Load model and create Doc object
nlp = spacy.load('en_core_web_lg')
doc = nlp('I am happy')
```

```
# Generate word vectors for each token
for token in doc:
  print(token.vector)
```

```
[-1.0747459e+00  4.8677087e-02  5.6630421e+00  1.6680446e+00
 -1.3194644e+00 -1.5142369e+00  1.1940931e+00 -3.0168812e+00
 ...
```

Word similarities

```
doc = nlp("happy joyous sad")
for token1 in doc:
  for token2 in doc:
    print(token1.text, token2.text, token1.similarity(token2))
```

```
happy happy 1.0
happy joyous 0.63244456
happy sad 0.37338886
joyous happy 0.63244456
joyous joyous 1.0
joyous sad 0.5340932
...
```

Document similarities

```
# Generate doc objects
sent1 = nlp("I am happy")
sent2 = nlp("I am sad")
sent3 = nlp("I am joyous")
```

```
# Compute similarity between sent1 and sent2
sent1.similarity(sent2)
```

```
0.9273363837282105
```

```
# Compute similarity between sent1 and sent3
sent1.similarity(sent3)
```

```
0.9403554938594568
```