Working With Text Data

IMLP Chapter 7

Table of Contents

1. Text Data
2. Bag of words
3. Tokenization
4. idf and tf-idf
5. Lemmatization and Stemming

Table of Contents

1. Text Data
2. Bag of Words
3. Tokenization
4. idf and tf-idf
5. Lemmatization and Stemming

Text Data

We have seen features including:

- continuous features that describe a quantity (real or integer data)
- categorical features that are items from a fixed list.

There is a third kind of data that can be found in many applications

These are “free form” data.

- email message as either a legitimate email or spam,
- Facebook posts opinion of a politician on the topic of immigration.
- Tweets
- Customer service message is a complaint or an inquiry
- Phone messages of complaint/inquiry

Strings versus Text Data

The data type String can be used represent text data. There are four kinds of String data you might see:

- Categorical data (black, white, grey etc)

- Free strings that can be semantically mapped to categories (people entering menu items like “salad with ranch”, “salad w ranch”, “salad with ranch dressing”, “salad with honey mustard”, etc)

- Structured string data (address, telephone numbers etc)

- Text data (all other strings)

Example: Movie Reviews

- As a running example, we will use a dataset of movie reviews from the IMDb (Internet Movie Database) website.
- This dataset contains the text of the reviews, together with a label that indicates whether a review is "positive" or "negative." The IMDb website itself contains ratings from 1 to 10.
- To simplify the modeling, this annotation is summarized as a two-class classification dataset where reviews with a score of 7 or higher are labeled as positive, and a score 4 or lower is labeled as negative (neutral reviews are not included in the dataset).

Example: Movie Reviews

Download data on Linux using:

```
wget -nc http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz -P data
! tar xzf data/aclImdb_v1.tar.gz --skip-old-files -C data
```

After unpacking the data, the data set is provided as text files in two separate folders, one for the training data, and one for the test data. Each of these in turn has two subfolders, one called “positive” and one called “negative”:

```
!tree -dL 2 data/aclImdb
data/aclImdb
├── test
│   ├── neg
│   └── pos
└── train
    ├── neg
    ├── pos
    └── unsup
!rm -r data/aclImdb/train/unsup
```

Example: Movie Reviews (Windows)

```
from sklearn.datasets import load_files
reviews_train = load_files("data/aclImdb/train/")
# load_files returns a bunch, containing training texts and training labels
text_train, y_train = reviews_train.data, reviews_train.target
print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))
print("text_train[6]:\n{}".format(text_train[6]))

type of text_train: <class 'list'>
length of text_train: 25000
text_train[6]:
b"This movie has a special way of telling the story, at first i found it rather odd as it
jumped through time and I had no idea whats happening.<br /><br />Anyway the story line was
although simple, but still very real and touching. You met someone the first time, you fell
in love completely, but broke up at last and promoted a deadly agony. Who hasn't go through
this? but we will never forget this kind of pain in our life. <br /><br />I would say i am
rather touched as two actor has shown great performance in showing the love between the
characters. I just wish that the story could be a happy ending."

print("Samples per class (training): {}".format(np.bincount(y_train)))

Out[5]:
Samples per class (training): [12500 12500]
```

Example: Movie Reviews

```
reviews_test = load_files("data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Number of documents in test data:{}".format(len(text_test)))
print("Samples per class (test): {}".format(np.bincount(y_test)))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]

Out[6]:
Number of documents in test data: 25000
Samples per class (test): [12500 12500]
```

Table of Contents

1. Text Data
2. Bag of Words
3. Tokenization
4. idf and tf-idf
5. Lemmatization and Stemming

Bag of words

- Get rid of paragraphs, punctuations, etc and make a bag of words
- Computing the bag-of-words representation for a corpus of documents consists of the following three steps:

1. *Tokenization*. Split each document into the words that appear in it (called *tokens*), for example by splitting them on whitespace and punctuation.

2. *Vocabulary building*. Collect a vocabulary of all words that appear in any of the documents, and number them (say, in alphabetical order)

3. *Encoding*. For each document, count how often each of the words in the vocabulary appear in this document.

Bag of Words

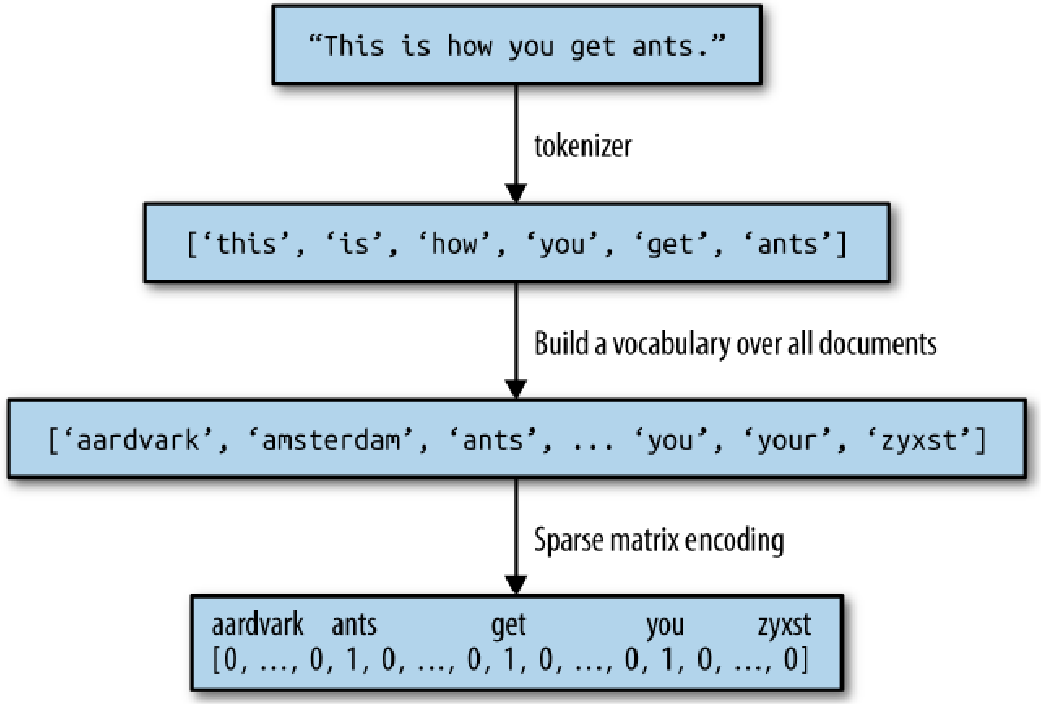
"This is how you get ants."
tokenizer
['this', 'is', 'how', 'you', 'get', 'ants']
Build a vocabulary over all documents
['aardvark', 'amsterdam', 'ants', ... 'you', 'your', 'zyxst']
Sparse matrix encoding
aardvark ants get you zyxst
[0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0]

Figure 7-1. Bag-of-words processing

Bag of Words

```
bards_words =["The fool doth think he is wise,", "but the wise man knows himself to be a fool"]

from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

Fitting the CountVectorizer consists of the tokenization of the training data and building of the vocabulary, which we can access as the vocabulary_ attribute:

```
print("Vocabulary size: {}".format(len(vect.vocabulary_)))
print("Vocabulary content:\n {}".format(vect.vocabulary_))

Vocabulary size: 13
Vocabulary content:

{'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2, 'to': 11, 'knows': 7, 'man': 8, 'fool':
3, 'is': 6, 'be': 0, 'think': 10, 'but': 1}

bag_of_words = vect.transform(bards_words)
print("Dense representation of bag_of_words:\n{}".format( bag_of_words.toarray()))

Dense representation of bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1] [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

Movie Review

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))
feature_names = vect.get_feature_names()
print("Number of features: {}".format(len(feature_names)))
print("First 20 features:\n{}".format(feature_names[:20]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 2000th feature:\n{}".format(feature_names[::2000]))

Number of features: 74849

First 20 features:
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830', '006', '007', '0079',
'0080', '0083', '0093638', '00am', '00pm', '00s', '01', '01pm', '02']
Features 20010 to 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback', 'drawbacks',
'drawer', 'drawers', 'drawing', 'drawings', 'drawl', 'drawled', 'drawling', 'drawn', 'draws',
'draza', 'dre', 'drea']
Every 2000th feature: ['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bete', 'chicanery',
'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freezer', 'goldman', 'hasan',
'huitieme', 'intelligible', 'kantrowitz', 'lawful', 'maars', 'megalunged', 'mostey', 'norrland',
'padilla', 'pincher', 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

Movie Review

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters: ", grid.best_params_)
```

```
Best cross-validation score: 0.89
Best parameters: {'C': 0.1}
```

We obtain a cross-validation score of 89% using C=0.1. We can now assess the generalization performance of this parameter setting on the test set:

```
X_test = vect.transform(text_test)
print("Test score: {:.2f}".format(grid.score(X_test, y_test)))
```

```
Test score: 0.88
```

Table of Contents

1. Text Data
2. Bag of words
3. Tokenization
4. idf and tf-idf
5. Lemmatization and Stemming

Improving Tokenization

Note that following are considered different words:
'draught', 'draughts',
'drawback', 'drawbacks',
'drawer', 'drawers',
'drawing', 'drawings',
'drawl', 'drawled', 'drawling'

The CountVectorizer extracts tokens using a regular expression. By default, the regular expression that is used is "\b\w\w+\b". If you are not familiar with regular expressions, this means it finds all sequences of characters that consist of at least two letters or numbers (\w) and that are separated by word boundaries (\b).

It does not find single-letter words, and it splits up contractions like "doesn't" or "bit.ly", but it matches "h8ter" as a single word.

The CountVectorizer then converts all words to lowercase characters, so that "soon", "Soon", and "sOon" all correspond to the same token (and therefore feature).

This simple technique works reasonably well.. But we could improve

Improving Tokenization

One way to cut back on these is to only use tokens that appear in at least two documents (or at least five documents, and
so on). A token that appears only in a single document is unlikely to appear in the test set and is therefore not helpful. We can set the minimum number of documents a token needs to appear in with the min_df parameter:

```
vect = CountVectorizer(min_df=5).fit(text_train)

X_train = vect.transform(text_train)
```

Improving Tokenization

```
feature_names = vect.get_feature_names()
print("First 50 features:\n{}".format(feature_names[:50]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 700th feature:\n{}".format(feature_names[::700]))
Out[18]:
First 50 features:
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08', '09', '10', '100',
'1000', '100th', '101', '102', '103', '104', '105', '107', '108', '10s', '10th', '11', '110',
'112', '116', '117', '11th', '12', '120', '12th', '13', '135', '13th', '14', '140', '14th',
'15', '150', '15th', '16', '160', '1600', '16mm', '16s', '16th']
Features 20010 to 20030:
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions', 'repetitious',
'repetitive', 'rephrase', 'replace', 'replaced', 'replacement', 'replaces', 'replacing',
'replay', 'replayable', 'replayed', 'replaying', 'replays', 'replete', 'replica']
Every 700th feature:
['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered', 'cheese', 'commitment',
'courts', 'deconstructed', 'disgraceful', 'dvds', 'eschews', 'fell', 'freezer', 'goriest',
'hauser', 'hungary', 'insinuate','juggle', 'leering', 'maelstrom', 'messiah', 'music',
'occasional', 'parking', 'pleasantville', 'pronunciation', 'recipient', 'reviews', 'sas',
'shea', 'sneers', 'steiger', 'swastika', 'thrusting', 'tvs', 'vampyre', 'westerns']
```

Table of Contents

1. Text Data
2. Bag of words
3. Tokenization
4. idf and tf-idf
5. Lemmatization and Stemming

Rescaling with idf (Inverse Document Frequency) value

We can also find the words that have low inverse document frequency—that is, those that appear frequently and are therefore deemed less important. The inverse document frequency values found on the training set are stored in the idf_ attribute:

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Features with lowest idf:\n{}".format( feature_names[sorted_by_idf[:100]]))
```

```
Features with lowest idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with' 'was' 'as' 'on' 'movie'
'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all' 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they'
'there' 'if' 'his' 'out''just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'
'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story' 'which' 'well' 'had'
'me' 'than' 'much' 'their' 'get' 'were' 'other' 'been' 'do' 'most' 'don' 'her' 'also' 'into'
'first' 'made' 'how' 'great' 'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after'
'any' 'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its' 'him']
```

Using Stopwords

Another way that we can get rid of uninformative words is by discarding words that are too frequent to be informative. There are two main approaches: using a language specific list of stopwords, or discarding words that appear too frequently. Scikitlearn has a built-in list of English stopwords in the feature_extraction.text

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Number of stop words: {}".format(len(ENGLISH_STOP_WORDS)))
print("Every 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))

Number of stop words: 318

Every 10th stopword:
['above', 'elsewhere', 'into', 'well', 'rather', 'fifteen', 'had', 'enough',
'herein', 'should', 'third', 'although', 'more', 'this', 'none', 'seemed', 'nobody',
'seems', 'he', 'also', 'fill', 'anyone', 'anything', 'me', 'the', 'yet', 'go',
'seeming', 'front', 'beforehand', 'forty', 'i']module:

vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train with stop words:\n{}".format(repr(X_train)))
```

Rescaling the Data with tf–idf

One of the most common ways to do this is using the *term frequency–inverse document frequency* (tf–idf) method. The intuition of this method is to give high weight to any term that appears
often in a particular document, but not in many documents in the corpus. If a word appears often in a particular document, but not in very many documents, it is likely to be very descriptive of the content of that document.

$$\mathrm{tfidf}(w, d) = \mathrm{tf} * \log\left(\frac{N+1}{N_w+1}\right) + 1$$

where *N* is the number of documents in the training set, *N*w is the number of documents in the training set that the word *w* appears in, and *tf* (the term frequency) is the number of times that the word *w* appears in the query document *d* (the document you want to transform or encode).

Rescaling the Data with tf-idf

```
print("Features with lowest tfidf:\n{}".format(
feature_names[sorted_by_tfidf[:20]]))
print("Features with highest tfidf: \n{}".format( feature_names[sorted_by_tfidf[-
20:]]))

Features with lowest tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred' 'currently'
'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker' 'avoided'
'emphasis' 'commented' 'disappoint' 'realizing' 'downhill' 'inane']

Features with highest tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur' 'vargas'
'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria' 'khouri' 'zizek'
'rob' 'timon' 'titanic']
```

Bag-of-Words with More Than One Word (n-Grams)

There is a way of capturing context when using a bag-of-words representation, by not only considering the counts of single tokens, but also the counts of pairs or triplets of tokens that appear next to each other.

Pairs of tokens are known as *bigrams*, triplets of tokens are known as *trigrams*.

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))

Vocabulary size: 14
Vocabulary: ['be fool', 'but the', 'doth think', 'fool doth', 'he is',
'himself to', 'is wise', 'knows himself', 'man knows', 'the fool', 'the wise',
'think he', 'to be', 'wise man']
```

`ngram_range=(m,M) m – minimum, M-maximum`

Table of Contents

1. Text Data
2. Bag of words
3. Tokenization
4. idf and tf-idf
5. Lemmatization and Stemming

Lemmatization and Stemming

- We saw earlier that the vocabulary often contains singular and plural versions of some words, as in "drawback" and "drawbacks", "drawer" and "drawers", and "drawing" and "drawings". For the purposes of a bag-of-words model, the semantics of "drawback" and "drawbacks" are so close that distinguishing them
- This problem can be overcome by representing each word using its *word stem*, which involves identifying (or *conflating*) all the words that have the same word stem. If this is done by using a rule-based heuristic, like dropping common suffixes, it is usually referred to as *stemming*. For example “work” is stem for “working”. “worked” and “worker” etc
- If instead a dictionary of known word forms is used (an explicit and human-verified system)*, and the role of the word in the sentence is taken into account*, the process is referred to as *lemmatization* and the standardized form of the word is referred to as the *lemma*.

Computing

```
import spacy
import nltk
# load spacy's English-language models
en_nlp = spacy.load('en')
# instantiate nltk's Porter stemmer
stemmer = nltk.stem.PorterStemmer()
# define function to compare lemmatization in spacy with stemming in nltk
def compare_normalization(doc):
# tokenize document in spacy
    doc_spacy = en_nlp(doc)
# print lemmas found by spacy
    print("Lemmatization:")
    print([token.lemma_ for token in doc_spacy])
# print tokens found by Porter stemmer
    print("Stemming:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

Computing

compare_normalization(u **"Our meeting today was worse than yesterday, “ "I'm scared of meeting the clients tomorrow.")**

Lemmatization:

['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be', 'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']

Stemming:

['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m", 'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']