15-388/688 - Practical Data Science: Free text and natural language processing

J. Zico Kolter
Carnegie Mellon University
Fall 2016

Outline

Free text in data science

Bag of words and TFIDF

Word embeddings and word2vec

Language models and N-grams

Libraries for handling free text

Announcements

HW1 solutions out (distributed as images)

Piazza tagging: tag posts with the proper subquestion tag, *not* just the question number in the post title

“Matplotlib” errors don’t actually have anything to do with Matplotlib, they are timeout errors (this is also stated in the return information, but it is quite ambiguous because of the Matplotlib warning … we are trying to address this)

Starting this week, I should be a bit more prompt about notes updates

Outline

Free text in data science

Bag of words and TFIDF

Language models and N-grams

(Next time) word2vec

(Next time) Libraries for handling free text

Outline

Free text in data science

Bag of words and TFIDF

Word embeddings and word2vec

Language models and N-grams

Libraries for handling free text

Free text in data science vs. NLP

A large amount of data in many real-world data sets comes in the form of free text (user comments, but also any “unstructured” field)

(Computational) natural language processing: write computer programs that can understand natural language

This lecture: try to get some meaningful information out of unstructured text data

Understanding language is hard

Multiple potential parse trees:

> "While hunting in Africa, I shot an elephant in my pajamas. How he got into my pajamas, I don't know." – Groucho Marx

Winograd schemas:

> "The city councilmen refused the demonstrators a permit because they [feared/advocated] violence."

Basic point: We use an incredible amount of context to understand what natural language sentences mean

But is it always hard?

Two reviews for a movie (Star Wars Episode 7)

1. “… truly, a stunning exercise in large-scale filmmaking; a beautifully-assembled picture in which Abrams combines a magnificent cast with a marvelous flair for big-screen, sci-fi storytelling.”

2. “It's loud and full of vim -- but a little hollow and heartless.”

Which one is positive?

We can often very easily tell the “overall gist” of natural language text without understanding the sentences at all

But is it always hard?

Two reviews for a movie (Star Wars Episode 7):

1. "… truly, a **stunning** exercise in large-scale filmmaking; a beautifully-assembled picture in which Abrams combines a **magnificent** cast with a **marvelous** flair for big-screen, sci-fi storytelling."

2. "It's loud and full of vim -- but a little **hollow** and **heartless**."

Which one is positive?

We can often very easily tell the "overall gist" of natural language text without understanding the sentences at all

Natural language processing for data science

In many data science problems, we don't need to truly understand the text in order to accomplish our ultimate goals (e.g., use the text in forming another prediction)

In this lecture we will discuss two simple but very useful techniques that can be used to infer some meaning from text *without* deep understanding

1. Bag of words approaches and TFIDF matrices
2. N-gram language models
3. word2vec models (next time)

Note: this lecture may be subject to change in the upcoming years, as massive improvements in "off-the-shelf" language understanding are ongoing (for example, this has already happened to image understanding)

Outline

Free text in data science

Bag of words and TFIDF

Word embeddings and word2vec

Language models and N-grams

Libraries for handling free text

Brief note on terminology

In this lecture, we will talk about “documents”, which mean individual groups of free text

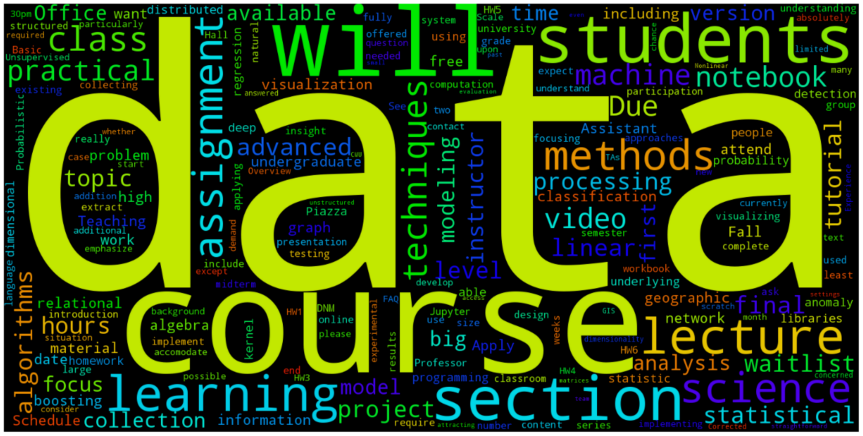
(Could be actual documents, or e.g. separate text entries in a table)

“Words” or “terms” refer to individual words (tokens separated by whitespace) and often also punctuation

“Corpus” refers to a collection of documents

Bag of words

AKA, the word cloud view of documents



Word cloud of class webpage

Represent each document as a vector of word frequencies

Order of words is irrelevant, only matters how often words occur

Bag of words example

"The goal of this lecture is to explain the basics of free text processing"

"The bag of words model is one such approach"

"Text processing via bag of words"

$$X = \begin{array}{c} \begin{matrix} \text{the} & \text{is} & \text{of} & \text{goal} & \text{lecture} & \text{bag} & \text{words} & \text{via} & \text{text} & & \text{approach} \end{matrix} \\ \begin{bmatrix} 2 & 1 & 2 & 1 & 1 & 0 & 0 & 0 & 1 & & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & \ldots & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & & 0 \end{bmatrix} \end{array} \begin{matrix} \\ \text{Document 1} \\ \text{Document 2} \\ \text{Document 3} \end{matrix}$$

Term frequency

"Term frequency" just refers to the counts of each word in a document

Denoted $\mathrm{tf}_{i,j}$ = frequency of word j in document i (sometimes indices are reversed, we use these for consistency with matrix above)

Often (as in the previous slide), this just means the raw count, but there are also other possibilities

1. $\mathrm{tf}_{i,j} \in \{0,1\}$ – does word occur in document or not
2. $\log(1 + \mathrm{tf}_{i,j})$ – log scaling of counts
3. $\mathrm{tf}_{i,j} / \max_j \mathrm{tf}_{i,j}$ – scale by document's most frequent word

Inverse document frequency

Term frequencies tend to be “overloaded” with very common words (“the”, “is”, “of”, etc)

Idea if *inverse document frequency* weight words negatively in proportion to how often they occur in the entire set of documents

$$\text{idf}_j = \log\left(\frac{\#\text{ documents}}{\#\text{ documents with word } j}\right)$$

As with term frequency, there are other version as well with different scalings, but the log scaling above is most common

Note that inverse document frequency is just defined for *words* not for word-document pairs, like term frequency

Inverse document frequency examples

$$X = \begin{bmatrix} 2 & 1 & 2 & 1 & 1 & 0 & 0 & 0 & 1 & & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & \dots & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & & 0 \end{bmatrix}$$

the is of goal lecture bag words via text approach

Document 1
Document 2
Document 3

$$\text{idf}_{\text{of}} = \log\left(\frac{3}{3}\right) = 0$$

$$\text{idf}_{\text{is}} = \log\left(\frac{3}{2}\right) = 0.405$$

$$\text{idf}_{\text{goal}} = \log\left(\frac{3}{1}\right) = 1.098$$

TFIDF

Term frequency inverse document frequency = $\mathrm{tf}_{i,j} \times \mathrm{idf}_j$

Just replace the entries in the X matrix with their TFIDF score instead of their raw counts (also common to remove "stop words" beforehand)

This seems to work much better than using raw scores for e.g. computing similarity between documents or building machine learning classifiers on the documents

the, is, of, goal

$$X = \begin{bmatrix} 0.8 & 0.4 & 0 & 1.1 & \\ 0.4 & 0.4 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \end{bmatrix}$$

Cosine similarity

A fancy name for “normalized inner product”

Given two documents x, y represented by TFIDF vectors (or just term frequency vectors), cosine similarity is just

$$\text{Cosine\_Similarity}(x, y) = \frac{x^T y}{\|x\|_2 \cdot \|y\|_2}$$

Between zero and one, higher numbers mean documents more similar

Equivalent to the (1 minus) the *squared distance* between the two normalized document vectors

$$\frac{1}{2}\|\tilde{x} - \tilde{y}\|_2^2 = 1 - \text{Cosine\_Similarity}(x, y), \text{where } \tilde{x} = \frac{x}{\|x\|_2}, \tilde{y} = \frac{y}{\|y\|_2}$$

Cosine similarity example

“The goal of this lecture is to explain the basics of free text processing”

“The bag of words model is one such approach”

“Text processing via bag of words”

$$M = \begin{bmatrix} 1 & 0.068 & 0.078 \\ 0.068 & 1 & 0.103 \\ 0.078 & 0.103 & 1 \end{bmatrix}$$

Poll: Cosine similarity

What would you expect to happen if the cosine similarity used term frequency vectors instead of TFIDF vectors?

1. Average cosine similarity between all documents would go up

2. Average cosine similarity between all documents would go down

3. Average cosine similarity between all documents would roughly stay the same

Outline

Free text in data science

Bag of words and TFIDF

Word embeddings and word2vec

Language models and N-grams

Libraries for handling free text

Term frequencies as vectors

You you think of individual words in a term-frequencies model as being "one-hot" vectors in an $\#\text{words}$ dimensional space (here $\#\text{words}$ is *total* number of unique words in corpus)

$$\text{“pittsburgh”} \equiv e_{\text{pittsburgh}} \in \mathbb{R}^{\#\text{words}} = \begin{bmatrix} \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix} \begin{matrix} \\ \text{pitted} \\ \text{pittsburgh} \\ \text{pivot} \\ \\ \end{matrix}$$

Document vectors are sums of their word vectors

$$x_{\text{doc}} = \sum_{\text{word} \in \text{doc}} e_{\text{word}}$$

“Distances” between words

No notion of similarity in term frequency vector space:

$$\|e_{\text{pittsburgh}} - e_{\text{boston}}\|_2 = \|e_{\text{pittsburgh}} - e_{\text{banana}}\|_2 = 1$$

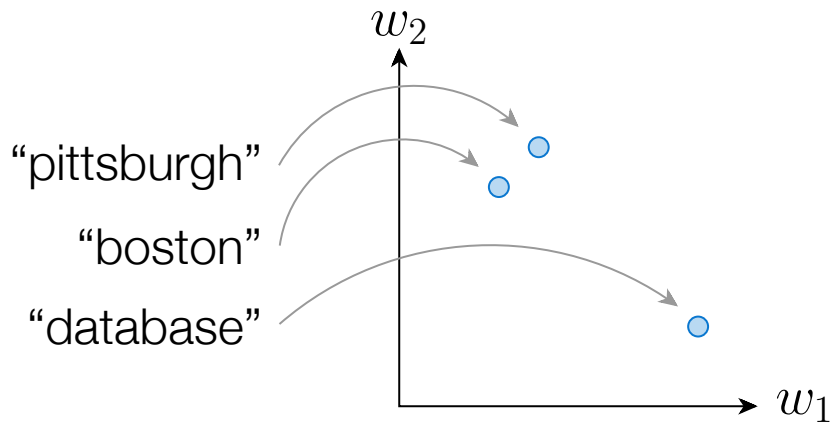
But, some words are inherently more related that others

- “Pittsburgh has some excellent new restaurants”
- “Boston is a city with great cuisine”
- “PostgreSQL is a relational database management system”

Under TFIDF cosine similarity (if we don’t remove stop words), then the second two sentences are more similar than the first and second

Word embeddings

Instead of term frequencies, we would like to represent words in a vector space $\mathbb{R}^k$ that captures some notion of *distance*



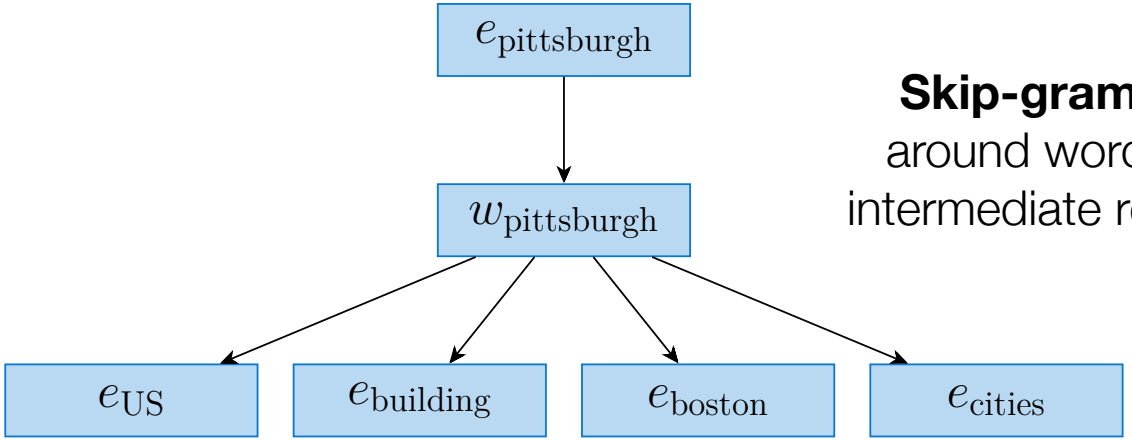
There are many possible ways to create such embeddings, but the word2vec algorithm ignited a lot of recent interest in the approach

Word2vec, briefly

We don't have the tools yet to describe the word2vec algorithm in any detail (we will need machine learning), but we will mention the basics

Given a large body of text, try to "predict" the neighbors of each word by the word itself

"… eastern US cities such as Pittsburgh and Boston are building…"



Skip-gram model: "predict" context around words from the word itself, use intermediate representation as embedding

Pretrained models

The good news is that you don't need to create these models yourself, there exist publicly-available "pretrained" models that have just hard-coded the embeddings for a large number of words

The "original" word2vec model, trained on 100 billion words from Google News documents, with vocabulary size of 3 million unique words:

https://code.google.com/archive/p/word2vec/

Some slightly more manageable ones available here:

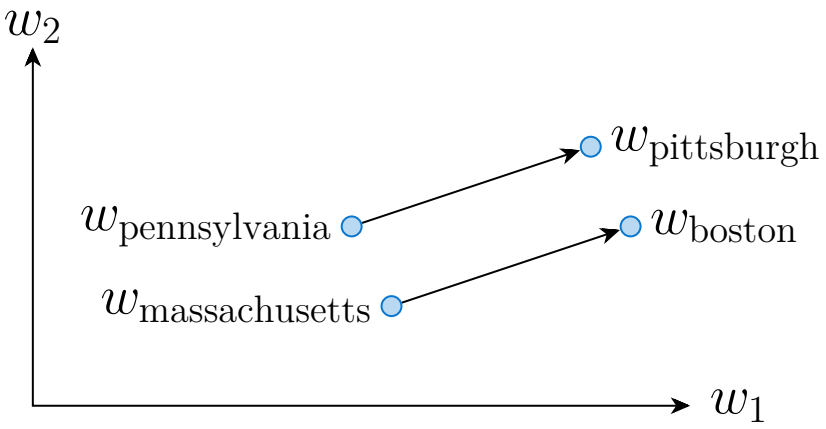
http://vsmlib.readthedocs.io/en/latest/tutorial/getting_vectors.html

Properties of word vectors

Some amazing properties naturally arise from the very simple training process described above

Nearby words: closest vectors to $w_{\text{pittsburgh}}$: “chicago”, “cincinnati”, “boston”, “detroit”

Analogies: closest vectors to $w_{\text{boston}} - w_{\text{massachuetts}} + w_{\text{pennsylvania}}$ (besides “boston”): “pittsburgh”, “philadelphia”, “chicago”, “denver”



Bag of word vectors

We can create "bag of words" from word embedding vectors instead of term frequency vectors (see also, doc2vec model)

$$x_{\text{doc}} = \sum_{\text{word} \in \text{doc}} w_{\text{word}}$$

Can capture the "meaning" of words rather than just counts

- "Pittsburgh has some excellent new restaurants"
- "Boston is a city with great cuisine"
- "PostgreSQL is a relational database management system"

$$M_{\text{tfidf}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.052 \\ 0 & 0.052 & 1 \end{bmatrix} \qquad M_{\text{word2vec}} = \begin{bmatrix} 1 & 0.787 & 0.700 \\ 0.787 & 1 & 0.765 \\ 0.700 & 0.765 & 1 \end{bmatrix}$$

Outline

Free text in data science

Bag of words and TFIDF

Word embeddings and word2vec

Language models and N-grams

Libraries for handling free text

Language models

While the bag of words model is surprisingly effective, it is clearly throwing away a lot of information about the text

The terms “boring movie and not great” is not the same in a movie review as “great movie and not boring”, but they have the exact same bag of words representations (both TFIDF and word2vec)

To move beyond this, we would like to build a more accurate model of how words really relate to each other: language model

Probabilistic language models

We haven't covered probability much yet, but with apologies for some forward references, a (probabilistic) language model aims at providing a probability distribution over every word, given all the words before it

$$P(\text{word}_i | \text{word}_1, \ldots, \text{word}_{i-1})$$

E.g., you probably have a pretty good sense of what the next word should be:

> "Data science is the study and practice of how we can extract insight and knowledge from large amounts of"

$$P(\text{word}_i = \text{“data”} | \text{word}_1, \ldots, \text{word}_{i-1}) = ?$$
$$P(\text{word}_i = \text{“hotdogs”} | \text{word}_1, \ldots, \text{word}_{i-1}) = ?$$

Building language models

Building a language model that captures the true probabilities of natural language is still a distant goal

Instead, we make simplifying assumptions to build approximate but tractable models

n-gram model: the probability of a word depends only on the $n-1$ word preceding it

$$P(\text{word}_i|\text{word}_1,\ldots,\text{word}_{i-1}) \approx P(\text{word}_i|\text{word}_{i-n+1},\ldots,\text{word}_{i-1})$$

This puts a hard limit on the *context* that we can use to make a prediction, but also makes the modeling more tractable

"large amounts of data" vs. "large amounts of hotdogs"

Estimating probabilities

A simple way (but *not* the only way) to estimate the conditional probabilities is simply by counting

$$P(\text{word}_i|\text{word}_{i-n+1},\dots,\text{word}_{i-1}) = \frac{\#(\text{word}_{i-n+1},\dots,\text{word}_{\text{i}})}{\#(\text{word}_{i-n+1},\dots,\text{word}_{i-1})}$$

E.g.:

$$P(\text{“data”}|\text{“large amounts of”}) = \frac{\#(\text{“large amounts of data”})}{\#(\text{“large amounts of”})}$$

Example of estimating probabilities

Very short corpus:

“The goal of this lecture is to explain the basics of free text processing”

Using an 2-gram model

$$P(\text{word}_i | \text{word}_{i-1} = \text{“of”}) = ?$$

Laplace smoothing

Estimating language models with raw counts tends to estimate a lot of zero probabilities (especially if estimating the probability of some new text that was not used to build the model)

Simple solution: allow for any word to appear with some small probability

$$P(\text{word}_i|\text{word}_{i-n+1},\dots,\text{word}_{i-1}) = \frac{\#(\text{word}_{i-n+1},\dots,\text{word}_{\text{i}}) + \alpha}{\#(\text{word}_{i-n+1},\dots,\text{word}_{i-1}) + \alpha D}$$

where α is some number and D is total size of dictionary

Also possible to have "backoffs" that use a lower degree n-gram when the probability is zero

How do we pick n?

Lower n: less context, but more samples of each possible n-gram

Higher n: more context, but less samples

“Correct” choice is to use some measure of held-out cross-validation

In practice: use $n = 3$ for large datasets (i.e., triplets) , $n = 2$ for small ones

Examples

Random samples from language model trained on Shakespeare:

n=1: "in as , stands gods revenge ! france pitch good in fair hoist an what fair shallow-rooted , . that with wherefore it what a as your . , powers course which thee dalliance all"

n=2: "look you may i have given them to the dank here to the jaws of tune of great difference of ladies . o that did contemn what of ear is shorter time ; yet seems to"

n=3: "believe , they all confess that you withhold his levied host , having brought the fatal bowels of the pope ! ' and that this distemper'd messenger of heaven , since thou deniest the gentle desdemona ,"

More examples

n=7: “`so express'd : but what of that ? 'twere good you do so much for charity . i cannot find it ; 'tis not in the bond . you , merchant , have you any thing to say ? but little`”

This is starting to look a lot like Shakespeare, because it is Shakespeare

As we have higher order n-grams, the previous (n-1) words have only appeared very few times in the corpus, so we will always just sample the next word that occurred

Evaluating language models

How do we know how well a language model performs

Common strategy is to estimate the probability of some held out portion of data, and evaluate *perplexity*

$$\text{Perplexity} = 2^{-\frac{\log_2 P(\text{word}_1, \ldots \text{word}_N)}{N}} = \left(\frac{1}{P(\text{word}_1, \ldots \text{word}_N)}\right)^{\frac{1}{N}}$$

where we can evaluate the probability using

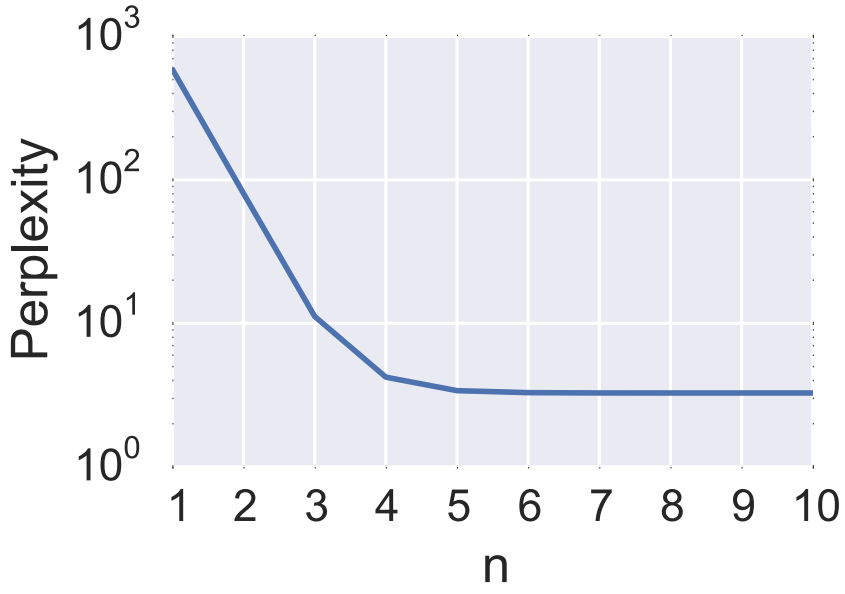
$$P(\text{word}_1, \ldots \text{word}_n) = \prod_{i=n}^{N} P(\text{word}_i | \text{word}_{i-n+1}, \ldots, \text{word}_{i-1})$$

(note that you can compute the log of this quantity directly)

Evaluating perplexity

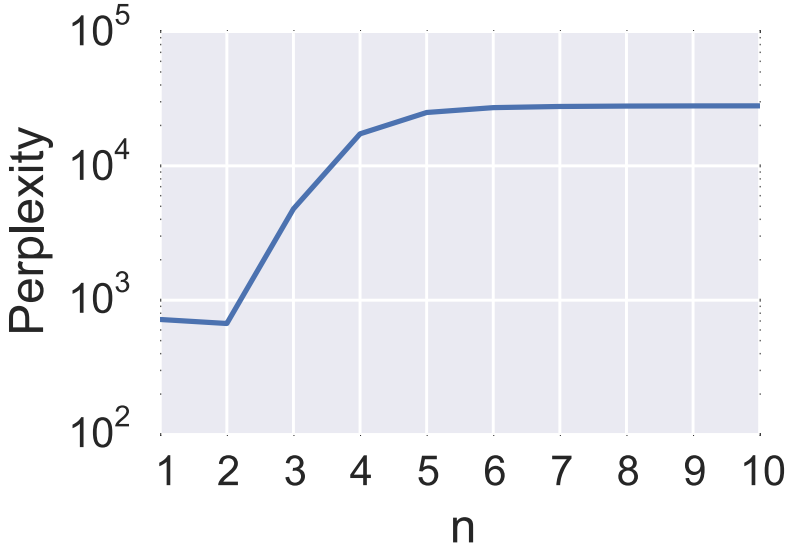
Perplexity on the corpus used to build the model will always decrease using higher n (fewer choices of what comes next means higher probability of observed data)

Note: this is only strictly true when $\alpha = 0$



Evaluating perplexity

What really matters is how well the model captures text from the “same” distribution that was *not* used to train the model



This is a preview of overfitting/model selection, which we will talk about a lot in the machine learning section

Outline

Free text in data science

Bag of words and TFIDF

Word embeddings and word2vec

Language models and N-grams

Libraries for handling free text

NLTK library

The NLTK (natural language toolkit) library (http://www.nltk.org) is a standard Python library for handling text and natural language data

Note: NLTK is a massive library, and is a bit more geared towards things like tagging, parsing, and more complex processes instead of the techniques described previously

Additionally, it actually doesn't contain much of what we want to do (no TFIDF creation, there was an n-gram language model but it was removed due to bugs)

You may want to look at some other options: spacy, CoreNLP

Reading and tagging documents

Load nltk and download necessary files:

```
import nltk
import nltk.corpus
#nltk.download() # just run this once
```

Tokenize a document

```
sentence = "The goal of this lecture isn't to explain complex free text
processing"
tokens = nltk.word_tokenize(sentence)
# ['The', 'goal', 'of', 'this', 'lecture', 'is', "n't", 'to', 'explain',
'complex', 'free', 'text', 'processing']
```

Tag parts of speech

```
pos = nltk.pos_tag(tokens)
# [('The', 'DT'), ('goal', 'NN'), ('of', 'IN'), ('this', 'DT'),
('lecture', 'NN'), ('is', 'VBZ'), ("n't", 'RB'), ('to', 'TO'),
('explain', 'VB'), ('complex', 'JJ'), ('free', 'JJ'), ('text', 'NN'),
('processing', 'NN')]
```

Stop words and n-grams

Get list of English stop words (common words)

```
stopwords = nltk.corpus.stopwords.words("English")
print [a for a in tokens if a.lower() not in stopwords]
# ['goal', 'lecture', "n't", 'explain', 'complex', 'free', 'text',
'processing']
```

Generate n-grams from document

```
list(nltk.ngrams(tokens, 3))
# [('The', 'goal', 'of'), ('goal', 'of', 'this'), ('of', 'this',
'lecture'), ('this', 'lecture', 'is'), ('lecture', 'is', "n't"), ('is',
"n't", 'to'), ("n't", 'to', 'explain'), ('to', 'explain', 'complex'),
('explain', 'complex', 'free'), ('complex', 'free', 'text'), ('free',
'text', 'processing')]

# code below does the same thing, without nltk
zip(*[tokens[i:] for i in range(3)])
```

Gensim library

Then Gensim library (https://radimrehurek.com/gensim) is a “topic modeling” library with a lot of nice built-in models, including:

- An implementation of word2vec (so you can train your own models),
- TFIDF vectors
- Topic modeling algorithms like Latent Dirichlet Alloation (LDA), etc

Fairly interoperable with corpus objects from NLTK

Note that you typically *don’t* want to train your own word2vec models unless you have a *lot* of data

Gensim examples

Import Gensim and set up corpus

```
import gensim as gs

# set up a Corpus in Gensim
documents = [
    "Pittsburgh has some excellent new restaurants",
    "Boston is a city with great cuisine",
    "PostgreSQL is a relational database management system"
]
words = [d.split() for d in documents]
dictionary = gs.corpora.Dictionary(words)
corpus = [dictionary.doc2bow(w) for w in words]
```

Create TDIDF X as CSC sparse matrix, get cosine similarities

```
tfidf = gs.models.TfidfModel(corpus)
X = gs.matutils.corpus2csc(tfidf[corpus])
sims = gs.similarities.MatrixSimilarity(tfidf[corpus])
M = sims.get_similarities(tfidf[corpus])
```

Gensim word2vec

Load pretrained word2vec model from file and use word vectors

```
model = gs.models.KeyedVectors.load_word2vec_format('deps.words',
                                                    binary=False)
# get word vectors
model.wv["pittsburgh"]

# compute most similar entries
model.similar_by_vector(model.wv["pittsburgh"])

# analogies by vector addition
model.similar_by_vector(model.wv["boston"] –
                        model.wv["massachusetts"] +
                        model.wv["pennsylvania"])
```

Construct cosine similarity from word2vec document representation

```
X = np.array([sum(model.wv[w.lower()] for w in doc) for doc in words])
X = X / np.linalg.norm(X,axis=1)[:,None]
print(X.dot(X.T))
```