Working with documents

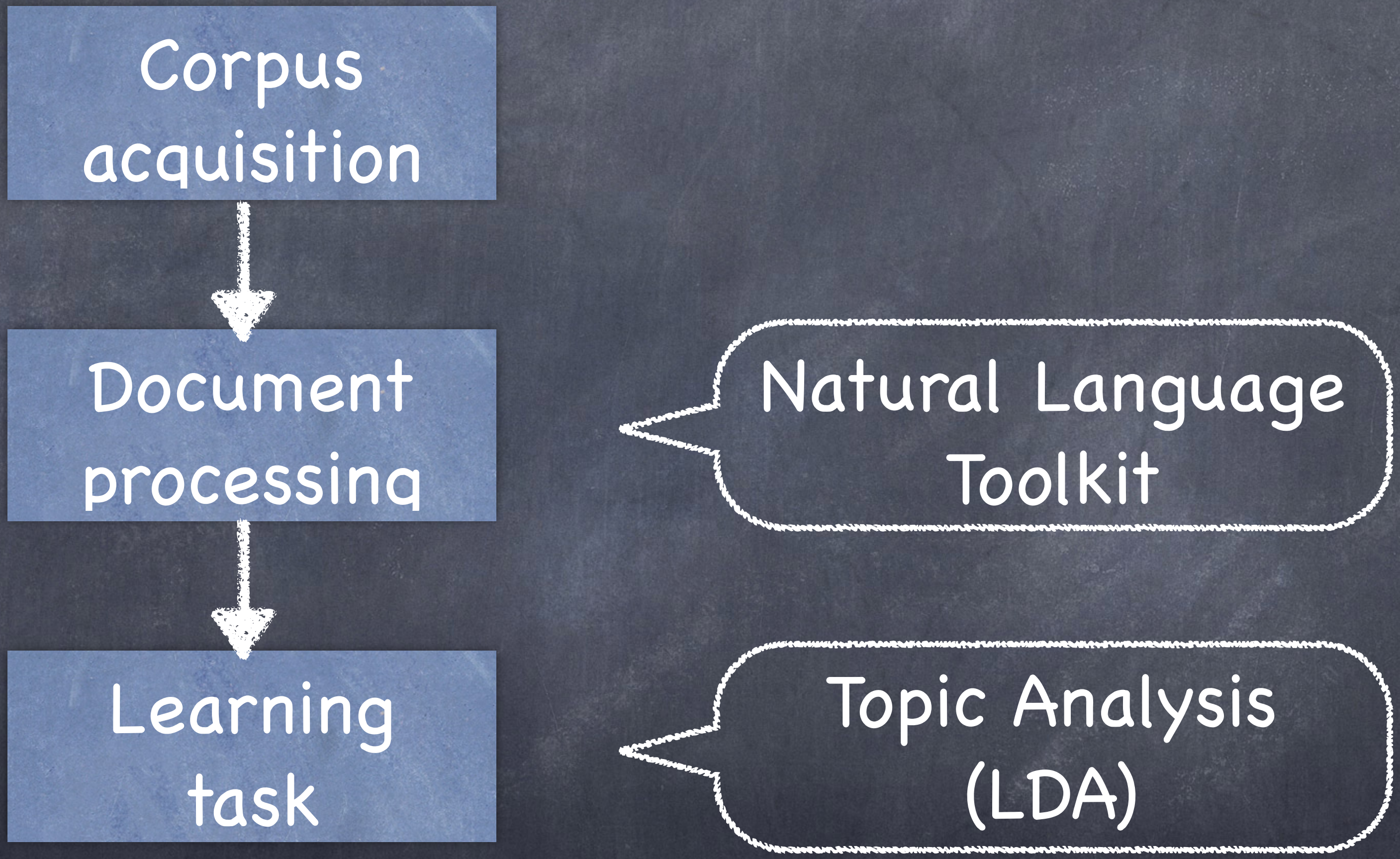
Vanessa Gómez Verdejo

Applications of Machine Learning

Master in Multimedia and Communications

Academic year 2014-2015

Contents



Corpus acquisition

Document processing

Learning task

Natural Language Toolkit

Topic Analysis (LDA)

Corpus acquisition

Any document can be analyzed…

- Web content: web pages, twitters, blogs, …
  - Crawler
  - Available APIs: wikipedia
- Local documents
- Available corpus: scikit-learn, NLTK

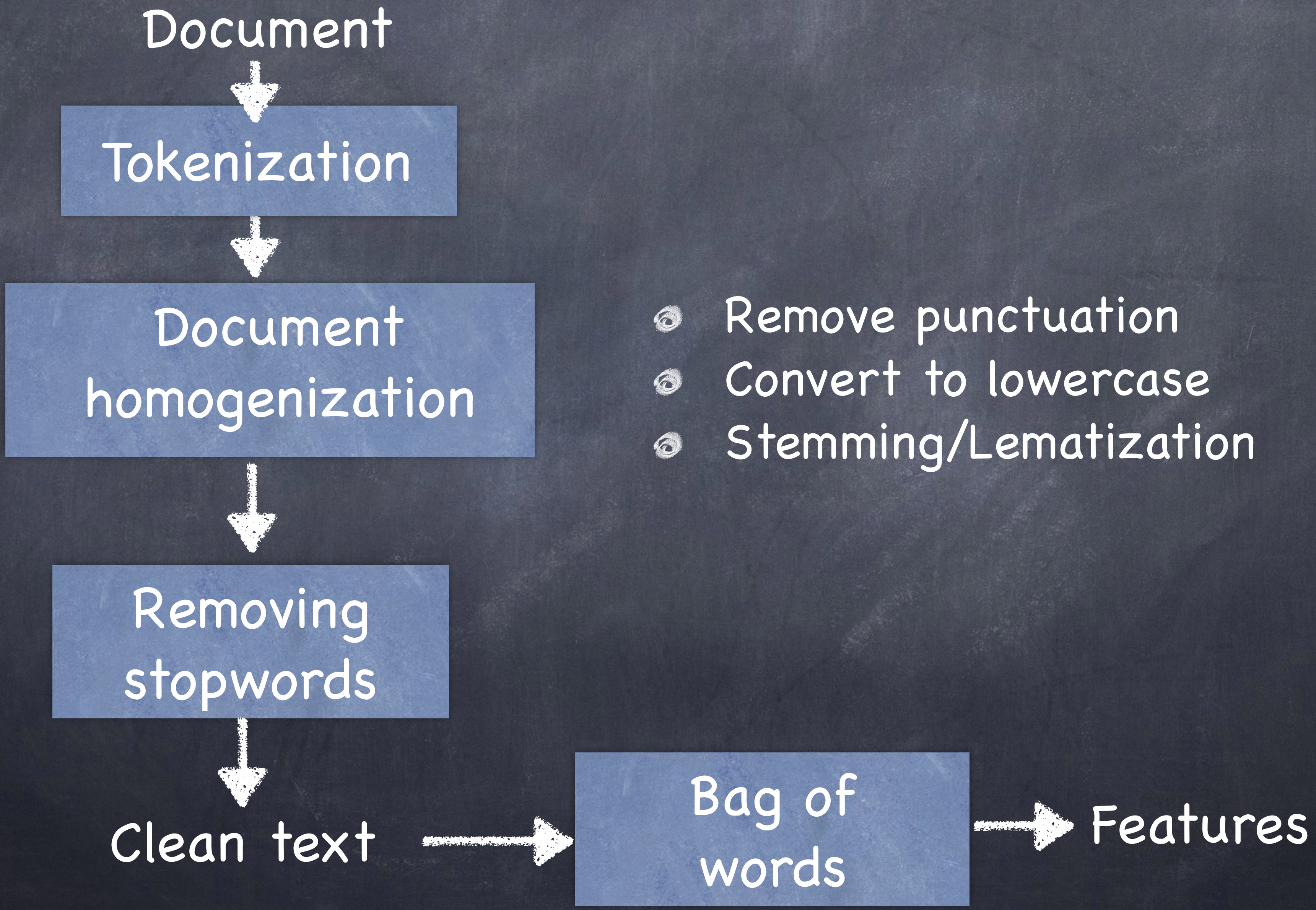
Loading a corpus

- From NLTK (pip install nltk)
  - import nltk
  - nltk.download()
  - mycorpus=nltk.corpus.gutenberg
  - .fileids() -> text_name=corpus.fileids()[0]
  - .raw() -> raw=corpus.raw(text_name)
  - .words() -> words=corpus.words(text_name)

Install it now and download book content (it takes a while)

Document processing

Pipeline



Document

Tokenization

Document homogenization

- Remove punctuation
- Convert to lowercase
- Stemming/Lematization

Removing stopwords

Clean text

Bag of words

Features

Tokenization

- From text to words (elements inside a sentence):
  - sentence=“Hola, mundo.”
  - sentence.split()
    - [‘Hola,’ , ‘mundo.’]
  - from nltk.tokenize import word_tokenize
    - word_tokenize(sentence)
    - [‘Hola’, ‘,’ , ‘mundo’ , ‘.’]

Document homogenization

- EXERCISE 1
  - Convert every word to lowercase
    - clean_text=[w.lower() for w in text]
- EXERCISE 2
  - Remove punctuation
    - clean_text=[w for w in text1 if w.isalnum()]

The class string in python

- s.find(t) index of first instance of string t inside s (-1 if not found)
- s.rfind(t) index of last instance of string t inside s (-1 if not found)
- s.join(text) combine the words of the text into a string using s as the glue
- s.split(t) split s into a list wherever a t is found (whitespace by default)
- s.lower() a lowercased version of the string s
- s.upper() an uppercased version of the string s
- s.title() a titlecased version of the string s
- s.strip() a copy of s without leading or trailing whitespace
- s.replace(t, u) replace instances of t with u inside s
- t in s test if t is contained inside s

Stemming

- We count similar words in different variants as different words
- We need a function that reduces words to their specific word stem.
  - import nltk.stem
  - s= nltk.stem.SnowballStemmer('english')
  - s.stem("imaging") -> u'imag'
  - s.stem("image") -> u'imag'

N-grams…

- Some words tend to occur in groups
  - information processing, machine learning…
- It can be useful that they are analyzed in groups
- There are routines to detect them, but the easiest way is…
  - informationprocessing
  - machinelearning

Removing less important words

- Some words appear very often in all sorts of different contexts.
- They are so frequent that they do not help to distinguish between different texts.
- These words are called stop words.
- The best option would be to remove them

Stopwords

- from nltk.corpus import stopwords

- stopwords = nltk.corpus.stopwords.words('english')

- clean_text=[word for word in document if not word in stopwords]

Corpus processing

Working with the corpus

- Until now, we have worked with a single document
- Extend your code to work with all the documents of the corpus
- Create a list of text, where each row is a previously processed text

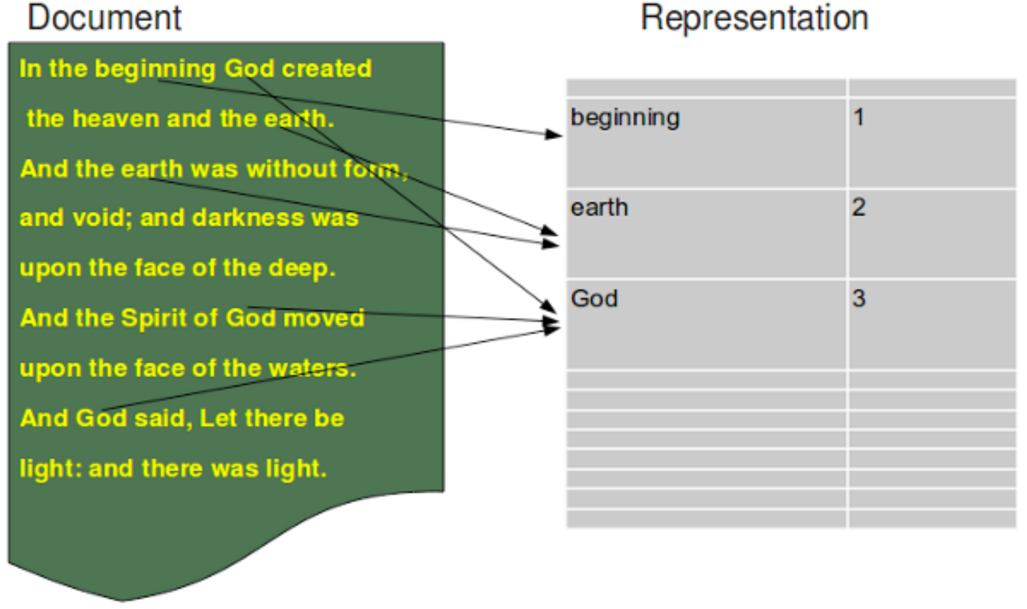
```
content=[
        [u'fulton', u'counti', u’grand', …, u'said', u’friday’]
        [u’austin', u’texa', u’committe', …, u’price', u’abandon’]
        ….
        [u’dear', u’sir', u’let', u’begin', …, u’mind', u’address’]
    ]
```

Working with the corpus

```
content=[]
for text_name in corpus.fileids():
    path = nltk.data.find('corpora/brown/'+text_name)
    f=open(path, 'rU')
    raw = f.read()
    # Here you can process your raw text -> clean_text
     content.append(clean_text)
      f.close()
```

Bag of words: counting words

- From the ML point of view, raw text is useless.
- Only if we manage to transform it into meaningful numbers, can we feed it into our ML algorithms
- **Bag-of-word** approach: for each word in the document, counts its occurrence and notes it in a vector



Document

In the beginning God created
the heaven and the earth.
And the earth was without form,
and void; and darkness was
upon the face of the deep.
And the Spirit of God moved
upon the face of the waters.
And God said, Let there be
light: and there was light.

Representation

beginning	1
earth	2
God	3

Term frequency - Inverse document frequency (TF-IDF)

- BoW: the feature values simply count occurrences of terms in a document.
- High occurrence terms?? They appear in all documents -> USELESS
- Low occurrence terms?? They appear in very few documents -> USEFULNESS
- This can only be solved by:
  - counting term frequencies for each document
  - discounting those that appear in many posts

Term frequency - Inverse document frequency (TF-IDF)

- We want a high value for a given term in a given doc if that term occurs often in that particular doc and very rarely anywhere else

$$TF(word, doc) = \frac{bow(word, doc)}{\#words\ in\ doc}$$

$$IDF(word, doc) = \log \frac{\#doc}{\#doc\ where\ is\ word}$$

$$TF - IDF(w, d) = TF(w, d) \times IDF(w, d)$$

- IDF->0 in common docs & IDF increases in rare docs

Topic Modelling

Topic Modeling

- Topic Modeling attempts to uncover the underlying semantic structure of by identifying recurring patterns of terms in a set of data (topics).
  - Does not parse sentences
  - Does not care about word order, and
  - Does not “understand” grammar or syntax
- Topic models are useful on their own to build visualizations and explore data. They are also very useful as an intermediate step in many other tasks.

Gensim

- Gensim is developed by Radim Řehůřek, who is a machine learning researcher and consultant in the Czech Republic.
- To install it:
  - pip install gensim
  - easy_install gensim

Data structures

- docs -> list of documents and each document is a list of words
- from gensim import corpora
- 1. Represent the words by ids (integer) -> create a **dictionary**
  - dictionary = corpora.Dictionary(docs)
- 2. Vectorize the documents -> **bow/tfidf**
  - corpus_bow = [dictionary.doc2bow(doc) for doc in docs]
- Efficient implementations for long corpus (work document to document): http://radimrehurek.com/gensim/tut1.html

Topics and transformations

- Gensim includes:
  - BOW
  - TF-IDF
  - LSA/LSI
  - Latent Dirichlet Allocation, LDA

TF-IDF

- from gensim import models
- tfidf = models.TfidfModel(corpus_bow) #1-- initialize a model
- From now on, tfidf can be used to convert any vector from the old representation (bow integer counts) to the new representation (TfIdf real-valued weights):
  - doc_bow = [(0, 1), (1, 1)]
  - tfidf[doc_bow] #2-- transform a new vector
- Or to apply a transformation to a whole corpus:
  - corpus_tfidf = tfidf[corpus_bow]

Latent Semantic Indexing (Analysis)

- It transforms documents from either bag-of-words or (preferably) TfIdf-weighted space into a latent space of a lower dimensionality.
- LSI or LSA is able to correlate semantically related terms that are latent in a collection of text
- LSI uses example documents to establish the conceptual basis for each category.
- LSI overcomes two of the most problematic constraints of Boolean keyword queries: multiple words that have similar meanings (synonymy) and words that have more than one meaning (polysemy).

Latent Semantic Indexing

- LSI computes the term and document vector spaces by approximating the TF-IDF matrix, A, into 3 matrices:

$$A = TSD^T \quad T^T T = I_r \quad D^T D = I_r \quad S_{1,1} > S_{2,2} > \ldots > S_{r,r} > 0$$

  - concept vector matrix T (m x r)
  - singular values matrix S (r x r)
  - concept-document vector matrix D (n x r)

- So, matrix A can be approximated by a reduced number of concepts (k)

$$A \approx T_k S_k D_k^T \quad T_k^T T_k = I_k \quad D_k^T D_k = I_k$$

LSA-LSI

- Steps to transform our Tf-Idf corpus via Latent Semantic Indexing into a latent 2-D space
  - lsi = models.LsiModel(corpus_tfidf, id2word=dictionary, num_topics=2) # initialize an LSI transformation
  - # on real corpora, target dimensionality of 200-500 is recommended as a "golden standard"
  - corpus_lsi = lsi[corpus_tfidf] # create a double wrapper over the original corpus bow->tfidf->fold-in-lsi
- It allows incremental updates:
  - lsi.add_documents(another_tfidf_corpus)

LSA-LSI

- Analyzing the topics:
    - lsi.print_topics(2) # both bow->tfidf and tfidf->lsi transformations are actually executed here, on the fly
        - topic #0(1.594): -0.703*"trees" + -0.538*"graph" + -0.402*"minors" +...
            - “trees”, “graph” and “minors” are all related words (and contribute the most to the direction of the first topic)
        - topic #1(1.476): -0.460*"system" + -0.373*"user" + -0.332*"eps" +....

LSA-LSI

- Analyzing document representation over the topics:
  - for doc in corpus_lsi: # both bow->tfidf and tfidf->lsi transformations are actually executed here, on the fly

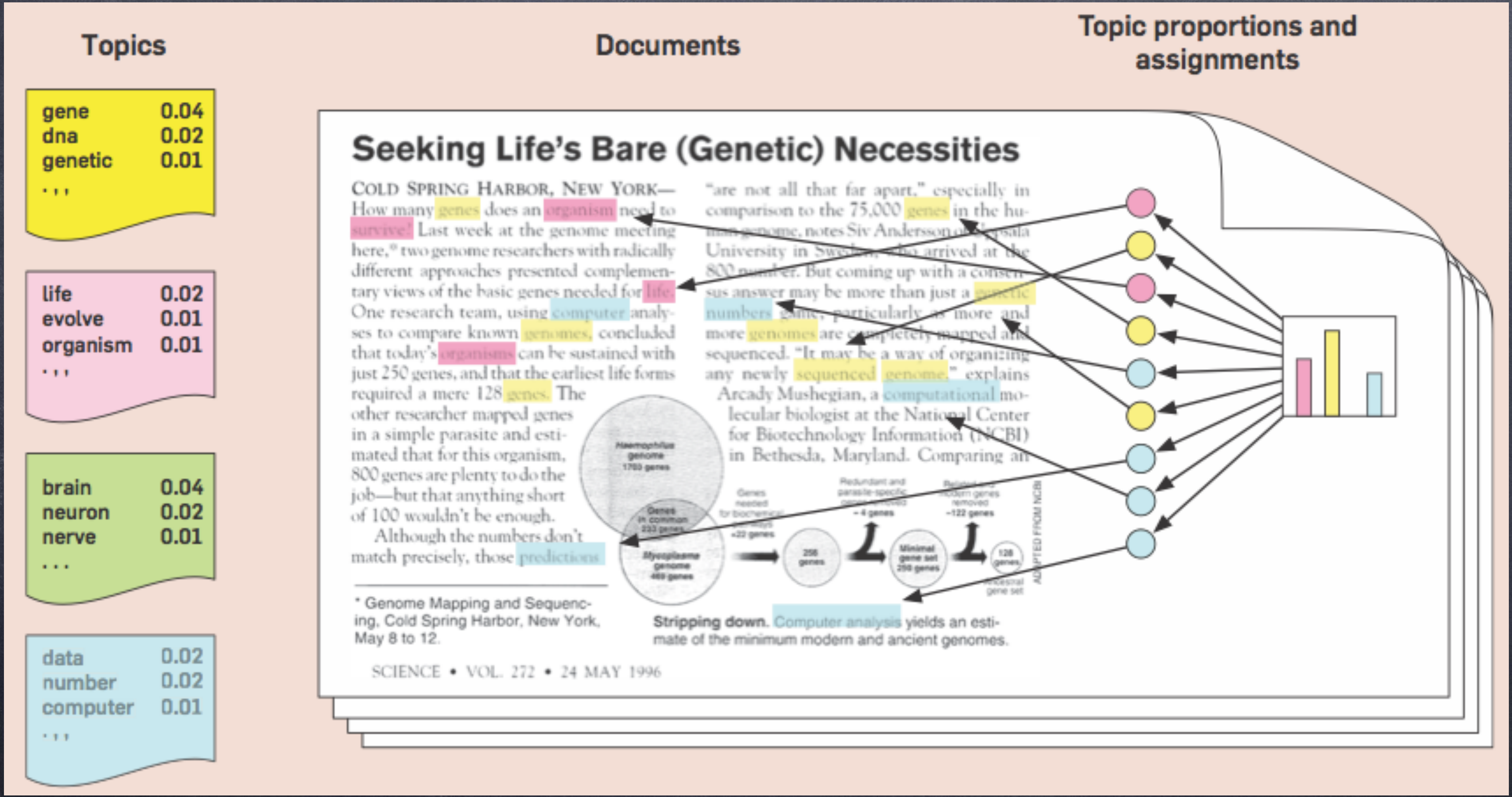
        print(doc)

    - [(0, -0.877), (1, -0.168)] # "The intersection graph of paths in trees"
    - [(0, -0.076), (1, 0.632)] # "System and human system engineering testing of EPS"

Latent Dirichlet Allocation

- It is another transformation from bag-of-words counts into a topic space of lower dimensionality.
- LDA is a probabilistic extension of LSA, so LDA's topics can be interpreted as probability distributions over words.
- These distributions are inferred automatically from a training corpus.
- Documents are in turn interpreted as a (soft) mixture of these topics (again, just like with LSA).

Latent Dirichlet Allocation


Topics
Documents
Topic proportions and assignments
gene 0.04
dna 0.02
genetic 0.01
...
life 0.02
evolve 0.01
organism 0.01
...
brain 0.04
neuron 0.02
nerve 0.01
...
data 0.02
number 0.02
computer 0.01
...
Haemophilus genome 1703 genes
Genes in common 233 genes
Mycoplasma genome 469 genes
Genes needed for biochemical pathways +22 genes
256 genes
Redundant and parasite-specific genes removed −4 genes
Minimal gene set 250 genes
Related and modern genes removed −122 genes
128 genes
Ancestral gene set
ADAPTED FROM NCBI

Seeking Life's Bare (Genetic) Necessities

COLD SPRING HARBOR, NEW YORK— How many genes does an organism need to survive? Last week at the genome meeting here,* two genome researchers with radically different approaches presented complementary views of the basic genes needed for life. One research team, using computer analyses to compare known genomes, concluded that today's organisms can be sustained with just 250 genes, and that the earliest life forms required a mere 128 genes. The other researcher mapped genes in a simple parasite and estimated that for this organism, 800 genes are plenty to do the job—but that anything short of 100 wouldn't be enough.

Although the numbers don't match precisely, those predictions "are not all that far apart," especially in comparison to the 75,000 genes in the human genome, notes Siv Andersson of Uppsala University in Sweden, who arrived at the 800 number. But coming up with a consensus answer may be more than just a genetic numbers game, particularly as more and more genomes are completely mapped and sequenced. "It may be a way of organizing any newly sequenced genome," explains Arcady Mushegian, a computational molecular biologist at the National Center for Biotechnology Information (NCBI) in Bethesda, Maryland. Comparing an

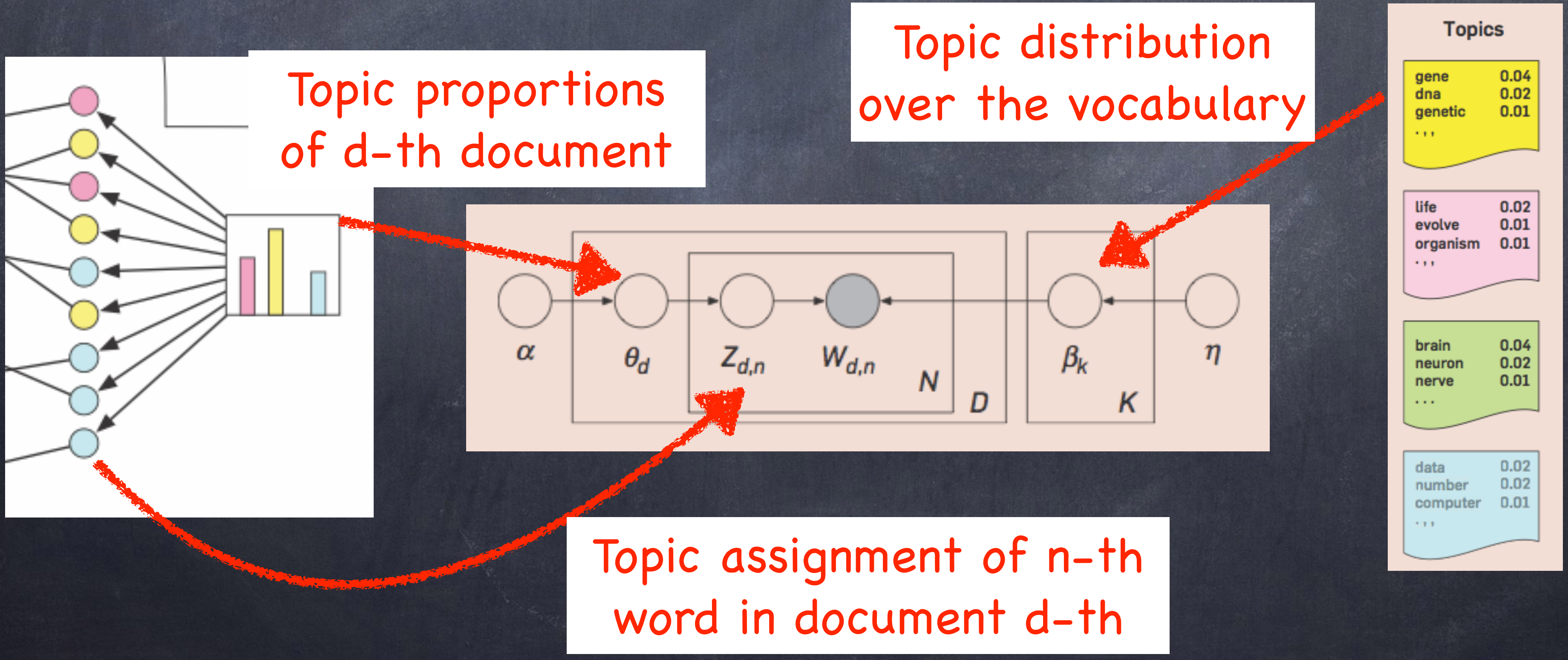
Stripping down. Computer analysis yields an estimate of the minimum modern and ancient genomes.

* Genome Mapping and Sequencing, Cold Spring Harbor, New York, May 8 to 12.

SCIENCE • VOL. 272 • 24 MAY 1996

Latent Dirichlet Allocation

- LDA is a generative probabilistic model:
    - GOAL -> inferring the topic structure (hidden variables) from the words of documents (observed variables)



Topic proportions of d-th document

Topic distribution over the vocabulary

Topic assignment of n-th word in document d-th

Latent Dirichlet Allocation

- The generative process for LDA corresponds to the following joint distribution of the hidden and observed variables

$$p(\beta_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D}) =$$

$$= \prod_{i=1}^{K} p(\beta_i) \prod_{d=1}^{D} p(\theta_d) \left(\prod_{n=1}^{N} p(z_{d,n} | \theta_d) p(w_{d,n} | \beta_{1:K}, z_{d,n}) \right)$$

- Goal: computing the conditional distribution of the topic structure given the observed documents

$$p(\beta_{1:K}, \theta_{1:D}, z_{1:D} | w_{1:D}) = \frac{p(\beta_{1:K}, \theta_{1:D}, z_{1:D}, w_{1:D})}{p(w_{1:D})}$$

Latent Dirichlet Allocation

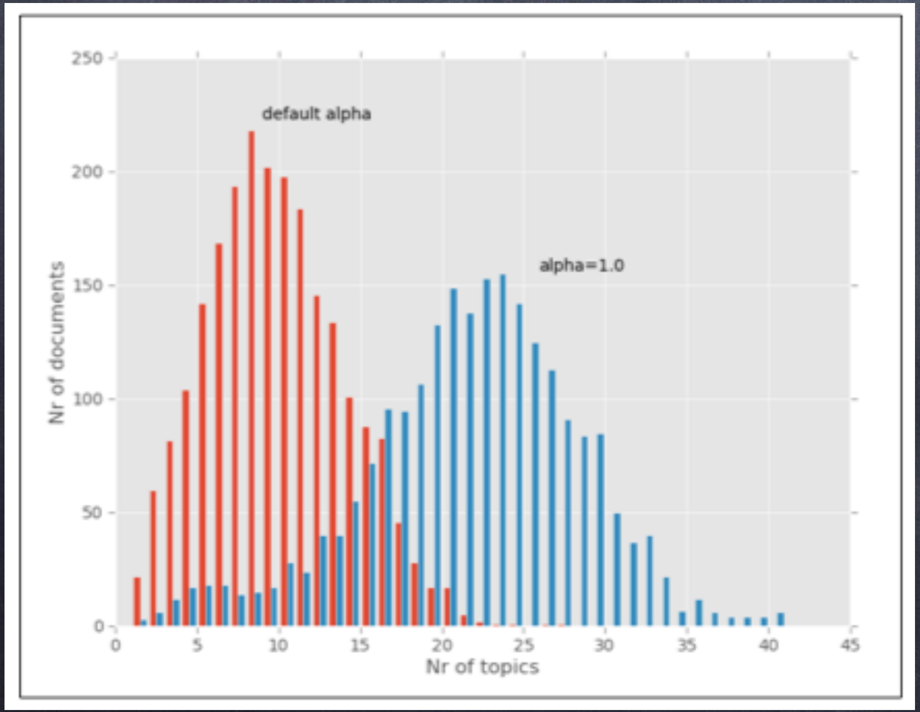
- The denominator is the probability of seeing the observed corpus under any topic model.
- It can be computed by summing the joint distribution over every possible hidden topic structure.
- The number of possible topic structures is exponentially large; this sum is intractable to compute.
- Two LDA families
  - Sampling-based algorithms
  - Variational algorithms

LDA in Gensim

- LDA steps:

-
  ```
  lda = gensim.models.LdaModel(corpus_bow, id2word=dictionary, alpha='auto', num_topics=20)

  # Create an LDA transformation
  ```



It automatically updates the alpha value for us

Bigger values for alpha will result in more topics per document

LDA in Gensim

- lda.print_topics(topics=5, topn=5) # Analize topics (the top 5 words associated with 5 topics)
- ['0.047*link + 0.027*ui + 0.018*main + 0.017*level + 0.016*locale',
- '0.107*tap + 0.047*popup + 0.045*appears + 0.031*request + 0.029*tab',
- '0.120*play + 0.096*ics + 0.084*music + 0.049*bug + 0.030*android',
- '0.106*device + 0.078*google + 0.060*talk + 0.057*voice + 0.044*icon’,
- '0.191*screen + 0.055*button + 0.034*change + 0.032*page + 0.032*lock']

the weights indicate how much
a word influences in a topic

LDA in Gensim

- More things to do....
  - for doc in lda: # training document representation

    print(doc)
- print(lda[doc_bow]) # get topic probability distribution for a document
- lda.update(corpus2) # update the LDA model with additional documents
- print(lda[doc_bow])