The Expectation-Maximization (EM) Algorithm

Kevyn Collins-Thompson

Associate Professor of Information and Computer Science
School of Information, University of Michigan

Maximum likelihood estimate of θ_A , the bias of a two-sided coin

We use "hat" to denote an estimate of a parameter.

$$\widehat{\theta}_A = \frac{\#\ of\ heads\ using\ coin\ A}{\#\ of\ total\ flips\ using\ coin\ A}$$

Actual flips:

HHHHHTHHTTHTTHHHTTHHHHTHHHTTHTTT

Number of "heads": 19

Total number of flips: 32

$\widehat{\theta}_A = 0.59375$

We call this the bias of the coin.

Binomial distribution : Models the number of successes (H)/failures (T) in *n* independent Bernoulli trials. Can show with simple calculus that the above formula (sample proportion of successes) gives the MLE for θ_A.

Review: Maximum Likelihood Estimation of Model Parameters

- The premise of the Maximum Likelihood approach is simple: find parameters $\hat{\theta}$ of an assumed probability distribution for observed data D, that maximize the likelihood (or probability) $p(D|\theta)$ of D.
- In practice we maximize the log likelihood $\log p(D|\theta)$ instead for mathematical convenience (adding regular numbers is easier than multiplying lots of tiny numbers).
- You've already seen maximum likelihood estimation of parameters in a model given some observations of random variables:
  - Simple example: coin toss. Estimate of bias parameter $\hat{\theta}$
  - Typically straightforward if you have a complete dataset and can observe all the variables involved.

Let's continue with coin-flipping…

Your friend has a coin and she would like you to check its bias.

- **Parameter to estimate**: $\hat{\theta}$, the bias of the coin (for a normal unbiased coin $\hat{\theta} = 0.50$)
- **Observed variable *X***: the number of Heads in a sequence of ten random flips.
- **Incomplete data**: none
- What is the value $\hat{\theta}$ that gives highest probability to the *N* observations of # Heads?
- Easy: $\hat{\theta} = \frac{n_{HEADS}}{n_{HEADS}+n_{TAILS}} = \frac{29}{29+21} = 0.58$
  - This is the MLE solution for the binomial distribution (our assumption for how X is distributed).

	X
Flips	**# Heads**
HTHTTHTHTH	5
HHHHTHHHHT	8
THTTTHTHHT	4
HHTHHTHHTH	7
HHTHTHHTTT	5

Update: Your friend comes back later with two coins.

She would like you to estimate the bias for each.
You're allowed to see which coin produced each sequence of flips.

- **Parameters to estimate**: $\hat{\theta}_A$, the bias of coin A and $\hat{\theta}_B$, the bias of coin B.
- **Observed variable *X***: the number of Heads in sequence of ten random flips.
- **Incomplete data**: none
- Easy again! Just estimate separately.

$$\hat{\theta}_A = \frac{n^A_{HEADS}}{n^A_{HEADS} + n^A_{TAILS}} = \frac{14}{30} = 0.47$$

$$\hat{\theta}_B = \frac{n^B_{HEADS}}{n^B_{HEADS} + n^B_{TAILS}} = \frac{15}{20} = 0.75$$

Coin used	Flips	# Coin A Heads	# Coin B Heads
A	HTHTTHTHTH	5	0
B	HHHHTHHHHT	0	8
A	THTTTHTHHT	4	0
B	HHTHHTHHTH	0	7
A	HHTHTHHTTT	5	0

Update 2: The complete data case was easy! **But** what if your friend hadn't told you which coin produced a sequence of flips?

Suppose the only thing she told you was that she selected between coin A and coin B with equal probability, but not which one produced the sequence.

- **Parameters to estimate**: $\hat{\theta}_A$, the bias of coin A and $\hat{\theta}_B$, the bias of coin B.
- **Observed variable *X***: the number of Heads in sequence of ten random flips.
- **Incomplete data**: We're missing which coin produced an observed sequence x_i. We'll call this latent variable Z_i.
- Z_i takes two possible values: Z_A *or* Z_B

- Now what?? It seems impossible that we could still estimate $\hat{\theta}_A$ and $\hat{\theta}_B$ …
- **But we can! Using ….**

Z

Coin used	Flips	# Coin A Heads	# Coin B Heads
?	HTHTTHTHTH	?	?
?	HHHHTHHHHT	?	?
?	THTTTHTHHT	?	?
?	HHTHHTHHTH	?	?
?	HHTHTHHTTT	?	?

? n^A_{HEADS} ? n^B_{HEADS}

The Expectation Maximization algorithm (EM)

- An iterative method to find approximately optimal* estimates of parameters in statistical models, when the data consists of both observed and incomplete data.
- 'Incomplete' data is often in the form of an unobserved latent variable, e.g.:
  - Which of two coins (w/ different biases) produced the series of flips.
  - Which of *k* clusters a data point belongs to.
  - Which of two possible authors wrote an anonymous article.
  - Which topics a web page is about.

* Maximum likelihood or (Bayesian) maximum a posteriori estimates, possibly a local optimum.

The initial step of EM seems so crazy that it just might work…

Initialize: Just start with a random guess for the parameters $\widehat{\boldsymbol{\theta}}_{\boldsymbol{A}}$ and $\widehat{\boldsymbol{\theta}}_{\boldsymbol{B}}$

- We'll call these initial parameter guesses $\widehat{\boldsymbol{\theta}}_{\boldsymbol{A}}^{\boldsymbol{0}}$ and $\widehat{\boldsymbol{\theta}}_{\boldsymbol{B}}^{\boldsymbol{0}}$

E-step:

- Having actual values for $\widehat{\boldsymbol{\theta}}_{\boldsymbol{A}}$ and $\widehat{\boldsymbol{\theta}}_{\boldsymbol{B}}$ now allows us to look at any sequence of flips and estimate coin probabilities for whether A or B was responsible. We denote these `unknown coin' probabilities by $\boldsymbol{p}(\boldsymbol{Z}_{\boldsymbol{i}}|\boldsymbol{X}_{\boldsymbol{i}}, \widehat{\boldsymbol{\theta}}_{\boldsymbol{A}}^{\boldsymbol{t}}, \widehat{\boldsymbol{\theta}}_{\boldsymbol{B}}^{\boldsymbol{t}})$ for iteration *t.*
- And with those coin probabilities, we can now estimate the expected number of Heads for each coin $(n_{HEADS}^{A}, n_{TAILS}^{B})$ for that set of trials.

M-step:

- Now that we have $(n_{HEADS}^{A}, n_{TAILS}^{B})$ we can use those to recompute new bias estimates $\widehat{\boldsymbol{\theta}}_{\boldsymbol{A}}^{\boldsymbol{t+1}}$ and $\widehat{\boldsymbol{\theta}}_{\boldsymbol{B}}^{\boldsymbol{t+1}}$

By repeating these two steps (E and M) our estimates for $\widehat{\boldsymbol{\theta}}_{\boldsymbol{A}}$ and $\widehat{\boldsymbol{\theta}}_{\boldsymbol{B}}$ get better and better until they converge to a solution that's a local maximum! Really!

The E-step

- Think of the goal of the E-step as filling in the missing data with 'pseudocounts'. Once this is done, we have a complete dataset (for estimation purposes)
- So first, how do we estimate the coin probabilities $\mathbf{Z_i}$ given an observed sequence? That is, the likelihood that coin A or coin B produced the sequence.

In the Z column:

- **Z_i = 0 if coin A was used for the i-th series**
- **Z_i = 1 if coin B was used**

Z

Coin used	Flips	# Coin A Heads	# Coin B Heads
?	HTHTTHTHTH	?	?
?	HHHHTHHHHT	?	?
?	THTTTHTHHT	?	?
?	HHTHHTHHTH	?	?
?	HHTHTHHTTT	?	?

The E-step: concrete example

- Current parameter estimates (guessing) are $\widehat{\boldsymbol{\theta}}_{\boldsymbol{A}} = 0.2$ and $\widehat{\boldsymbol{\theta}}_{\boldsymbol{B}} = 0.6$.
- Thus we know how to calculate $\boldsymbol{P}(\boldsymbol{X_i} \mid \boldsymbol{Z_A})$ and $\boldsymbol{P}(\boldsymbol{X_i} \mid \boldsymbol{Z_B})$ for any trial $\boldsymbol{X_i}$
- We observe the sequence $\boldsymbol{X_i}$ = HHHHHHHTTT
- The probability of a given sequence of heads is given by the binomial distribution:
- $\boldsymbol{P}(\boldsymbol{X_i} \mid \boldsymbol{Z_A}) = \boldsymbol{P}(\boldsymbol{HHHHHHHTTT} \mid \boldsymbol{coin\ A\ was\ chosen}) = \frac{\boldsymbol{10!}}{\boldsymbol{7!3!}}(\boldsymbol{0.2})^{\boldsymbol{7}}(\boldsymbol{0.8})^{\boldsymbol{3}}$
- $\boldsymbol{P}(\boldsymbol{X_i} \mid \boldsymbol{Z_B}) = \boldsymbol{P}(\boldsymbol{HHHHHHHTTT} \mid \boldsymbol{coin\ B\ was\ chosen}) = \frac{\boldsymbol{10!}}{\boldsymbol{7!3!}}(\boldsymbol{0.6})^{\boldsymbol{7}}(\boldsymbol{0.4})^{\boldsymbol{3}}$
- Now we need to calculate the coin probabilities
  $\boldsymbol{p}(\boldsymbol{Z_A}|\boldsymbol{X_i})$ and $\boldsymbol{p}(\boldsymbol{Z_B}|\boldsymbol{X_i})$
- Can we do this once we know $\boldsymbol{P}(\boldsymbol{X_i} \mid \boldsymbol{Z_B})$? Yes! We use Bayes Rule…

The E-step (coin probabilities)

Here's Bayes Rule, plus using our knowledge that $P(Z_A) = P(Z_B) = \frac{1}{2}$

$$P(Z_A|X_i) = \frac{P(X_i|Z_A)P(Z_A)}{P(X_i|Z_A)P(Z_A) + P(X_i|Z_B)P(Z_B)} = \frac{P(X_i|Z_A)}{P(X_i|Z_A) + P(X_i|Z_B)}$$

Now plugging in the values we have for $P(X_i|Z_A)$ and $P(X_i|Z_B)$:

$$P(Z_A|X_i, \hat{\theta}_A, \hat{\theta}_B) = \frac{\frac{10!}{7!\,2!}(0.2)^7(0.8)^3}{\frac{10!}{7!\,2!}(0.2)^7(0.8)^3 + \frac{10!}{7!\,2!}(0.6)^7(0.4)^3} = \boxed{0.00366}$$

A similar calculation gives the probability that it was coin B

$$P(Z_B|X_i, \hat{\theta}_A, \hat{\theta}_B) = \boxed{0.99634}$$

This makes sense: given our bias estimates $\hat{\theta}_A$ = 0.2 and $\hat{\theta}_B$ =0.6, the sequence HHHHHHHTTT was much more likely to be produced by coin B than coin A.

The E-step (coin probabilities)

We can repeat the previous calculation to fill in the left column of the table.

Notice that instead of the choice BETWEEN either A and B, we have a 'soft' choice based on the coin probabilities, since we're not sure:

Coin used probability	Flips	# Coin A Heads	# Coin B Heads
(A: 0.1163 B: 0.8836)	HTHTTHTHTH	?	?
(A: 0.0006 B: 0.9993)	HHHHTHHHHT	?	?
(A: 0.4414 B: 0.5586)	THTTTHTHHT	?	?
(A: 0.0036 B: 0.9964)	HHTHHTHHTH	?	?
(A: 0.1164 B: 0.8836)	HHTHTHHTTT	?	?

(This example assumes current estimates are $\widehat{\boldsymbol{\theta}}_{\boldsymbol{A}}$ = 0.2 and $\widehat{\boldsymbol{\theta}}_{\boldsymbol{B}}$ =0.6.

The E-step (expected counts)

- With these coin probabilities we can now estimated expected # Heads for A and B

Coin used probability	Flips	# Coin A Heads	# Coin B Heads
(A: 0.1163 B: 0.8836)	HTHTTHTHTH	0.5818	4.4181
(A: 0.0006 B: 0.9993)	HHHHTHHHHT	0.0049	7.9951
(A: 0.4414 B: 0.5586)	THTTTHTHHT	1.7655	2.2345
(A: 0.0036 B: 0.9964)	HHTHHTHHTH	0.0255	6.9745
(A: 0.1164 B: 0.8836)	HHTHTHHTTT	0.5818	4.4182

n^{A}_{HEADS}= 2.96 n^{B}_{HEADS}= 26.04

- And our dataset is now 'complete'

The M-step

- The M-step is actually just the same maximum likelihood parameter estimation we did before with complete data.
- So it’s easy: we’re just using our ‘pretend’ complete dataset to re-estimate the parameters.

The M-step

- Just proceed as in the complete data case, with the same MLE estimator:

$$\hat{\theta}_A^{t+1} = \frac{n_{HEADS}^A}{n_{HEADS}^A + n_{TAILS}^A} = \frac{2.96}{2.96 + 26.04} = 0.102$$

$$\hat{\theta}_B^{t+1} = \frac{n_{HEADS}^B}{n_{HEADS}^B + n_{TAILS}^B} = \frac{26.04}{2.96 + 26.04)} = 0.898$$

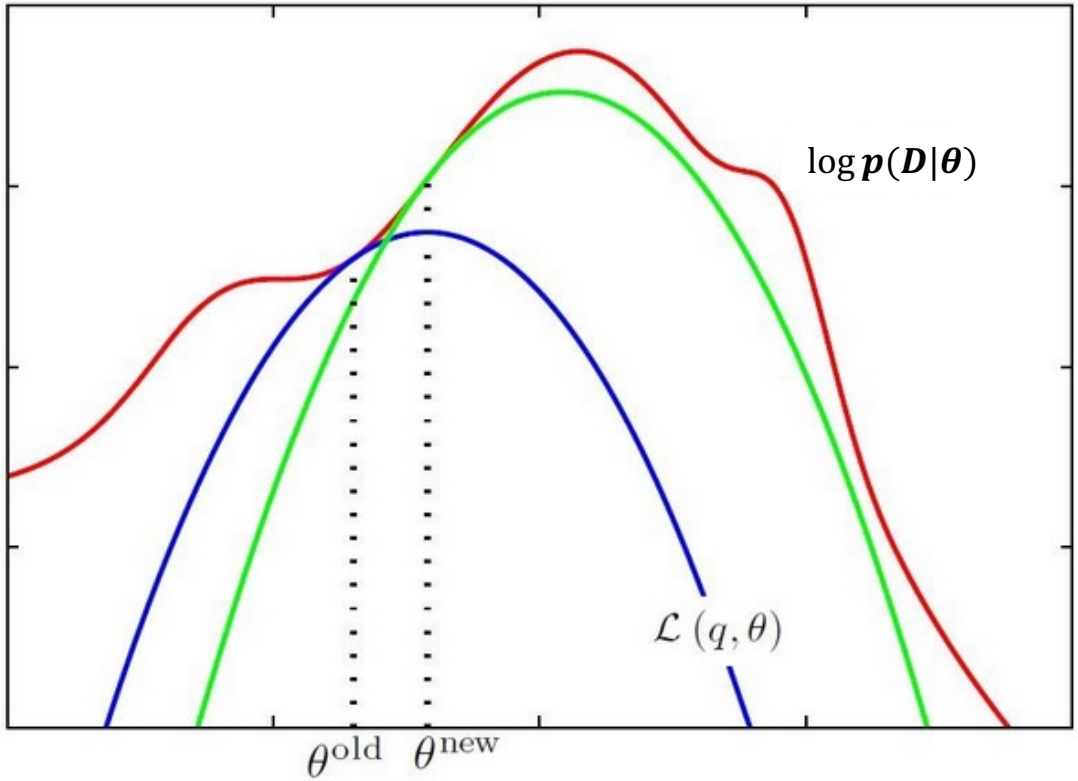
Coin probability	Flips	# Coin A Heads	# Coin B Heads
(A: 0.1163 B: 0.8836)	HTHTTHTHTH	0.5818	4.4181
(A: 0.0006 B: 0.9993)	HHHHTHHHHT	0.0049	7.9951
(A: 0.4414 B: 0.5586)	THTTTHTHHT	1.7655	2.2345
(A: 0.0036 B: 0.9964)	HHTHHTHHTH	0.0255	6.9745
(A: 0.1164 B: 0.8836)	HHTHTHHTTT	0.5818	4.4182
		n_{HEADS}^A= 2.96	n_{HEADS}^B= 26.04

With these updated parameter estimates $\hat{\theta}_A^{t+1}$ and $\hat{\theta}_B^{t+1}$, we can then repeat the E-step..

Review of EM

- EM can be used to estimate the parameters of a probability distribution when:
  1. We observe samples from the distribution where some variables are missing; (incomplete data, latent variables)
  2. The parameter estimation would be easy if we had values for those variables (complete data).
- Example : two-coin flipping problem
  - We observed samples from a two-binomial mixture where the coin A or B that actually generated each sample was missing.
  - Goal: estimate the parameters $\boldsymbol{\theta_A}$ and $\boldsymbol{\theta_B}$ of each binomial (coin bias)
  - The two-binomial parameter estimation problem would be easy with complete data (as we saw).
  - So we filled in the missing data to create a 'pretend' complete dataset.

Behind the magic: the E- and M-steps can be viewed as implementing a form of coordinate ascent



- The complete-data log-likelihood increases after each EM step
- Each EM step improves the parameter estimates!
- Convergence to local optimum guaranteed.
- Maximizes lower bound on marginal likelihood

- Replace D with q (pseudo data)

Source: https://towardsdatascience.com/inference-using-em-algorithm-d71cccb647bc

Code implementation

- No generic EM support in scikit-learn
- But EM is implemented to estimate parameters for e.g. Gaussian mixture models (`sklearn.mixture`)
- See this week's notebook

```
def coin_em(rolls, theta_A=None, theta_B=None, maxiter=10):
    # Initial Guess
    theta_A = theta_A or random.random()
    theta_B = theta_B or random.random()
    thetas = [(theta_A, theta_B)]
    # Iterate
    for c in range(maxiter):
        print("#%d:\t%0.2f %0.2f" % (c, theta_A, theta_B))
        heads_A, tails_A, heads_B, tails_B = e_step(rolls, theta_A, theta_B)
        theta_A, theta_B = m_step(heads_A, tails_A, heads_B, tails_B)

    thetas.append((theta_A,theta_B))
    return thetas, (theta_A,theta_B)
```

[Source: Example is based on original code from UMich EECS 445 tutorial by Jake Abernathy and Jia Deng, combined with additional material by Karl Rosaen.]

Code implementation

```
def e_step(rolls, theta_A, theta_B):
    """Produce the expected value for heads_A, tails_A, heads_B, tails_B
    over the rolls given the coin biases"""

    heads_A, tails_A = 0,0
    heads_B, tails_B = 0,0
    for trial in rolls:
        likelihood_A = coin_likelihood(trial, theta_A)
        likelihood_B = coin_likelihood(trial, theta_B)
        p_A = likelihood_A / (likelihood_A + likelihood_B)
        p_B = likelihood_B / (likelihood_A + likelihood_B)
        heads_A += p_A * trial.count("H")
        tails_A += p_A * trial.count("T")
        heads_B += p_B * trial.count("H")
        tails_B += p_B * trial.count("T")
    return heads_A, tails_A, heads_B, tails_B

def m_step(heads_A, tails_A, heads_B, tails_B):
    """Produce the values for theta that maximize the expected number of heads/tails"""

    theta_A = heads_A / (heads_A + tails_A)
    theta_B = heads_B / (heads_B + tails_B)
    return theta_A, theta_B
```

Coin probabilities via Bayes' Rule

Use coin probabilities to expect expected counts of Heads/Tails

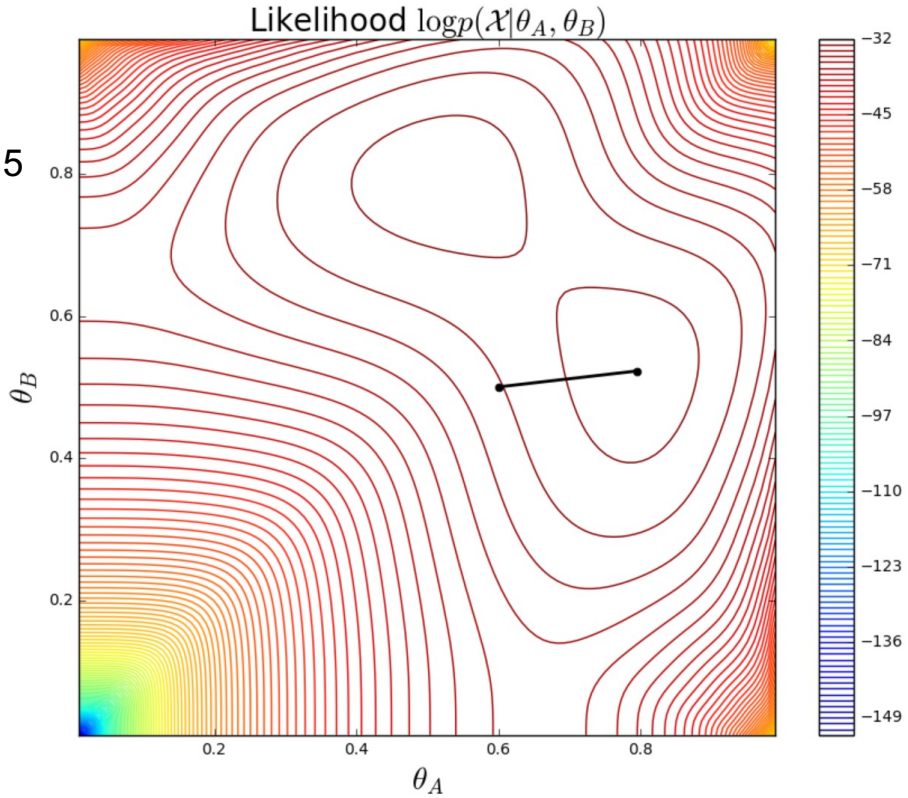
Data likelihood as a function of model parameters θ_A and θ_B

```
rolls = [ "HTTTHHTHTH", "HHHHTHHHHH", "HTHHHHHTHH",
          "HTHTTTHHTT", "THHHTHHHTH" ]
thetas, _ = coin_em(rolls, 0.6, 0.5, maxiter=6)
```

```
#0:      0.60 0.50
#1:      0.71 0.58
#2:      0.75 0.57
#3:      0.77 0.55
#4:      0.78 0.53
#5:      0.79 0.53
```

initial guess: θ_A =0.6, θ_B = 0.5

- Here's the hill-climbing that EM does for our two-coin flip example, showing the data log likelihood as a function of the model parameters θ_A and θ_B
- Note the symmetry of the two optimal solutions corresponding to exchanging θ_A and θ_B



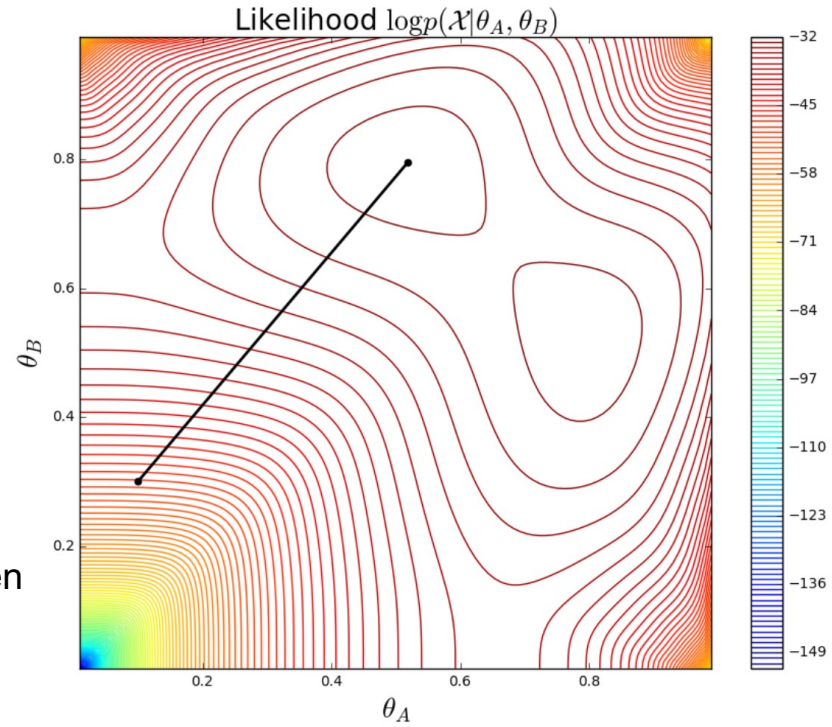
Let's see what happens if we pick a different initial guess for θ_A and θ_B

```
thetas2, _ = coin_em(rolls, 0.1, 0.3, maxiter=6)
```

```
#0:      0.10 0.30
#1:      0.43 0.66
#2:      0.50 0.75
#3:      0.51 0.78
#4:      0.52 0.79
#5:      0.52 0.79
```

initial guess: θ_A =0.1, θ_B =0.3

```
plot_coin_likelihood(rolls, thetas2)
```



- With a different initial parameter guess, EM converges to a *different* local optimum.
- This is *very* typical of EM. The likelihood landscape is typically complex and bumpy (even more than this simple example)
- EM should be run e.g. 10 times with different initial parameter guesses, then keep the solution with max. likelihood (here they're equal)

Questions to ask yourself about any statistical modeling problem

- What are the parameters I’m trying to estimate?
- What’s the observed data?
- What’s the missing data?

Example: the k-means clustering algorithm is the EM algorithm!

This explains the “magic” of the k-means algorithm!

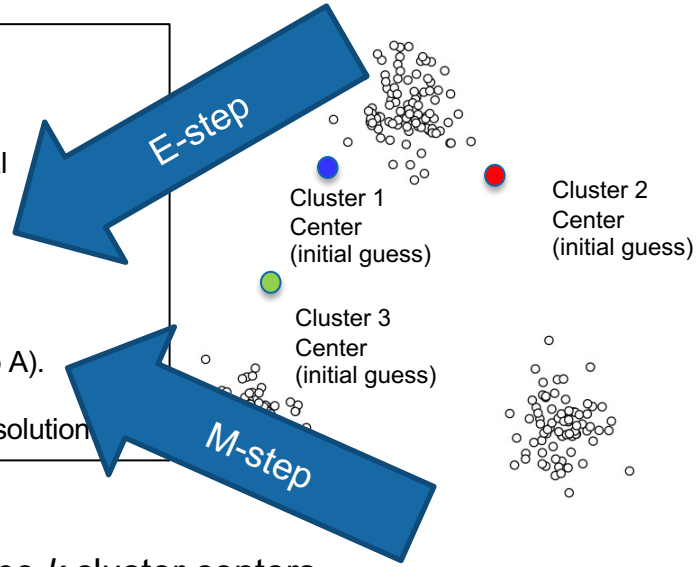
The k-means algorithm

Initialization Pick the number of clusters *k* you want to find. Then pick *k random* points to serve as an initial guess for the cluster centers.

Step A Assign each data point to the nearest cluster center.

Step B Update each cluster center by replacing it with the mean of all points assigned to that cluster (in step A).

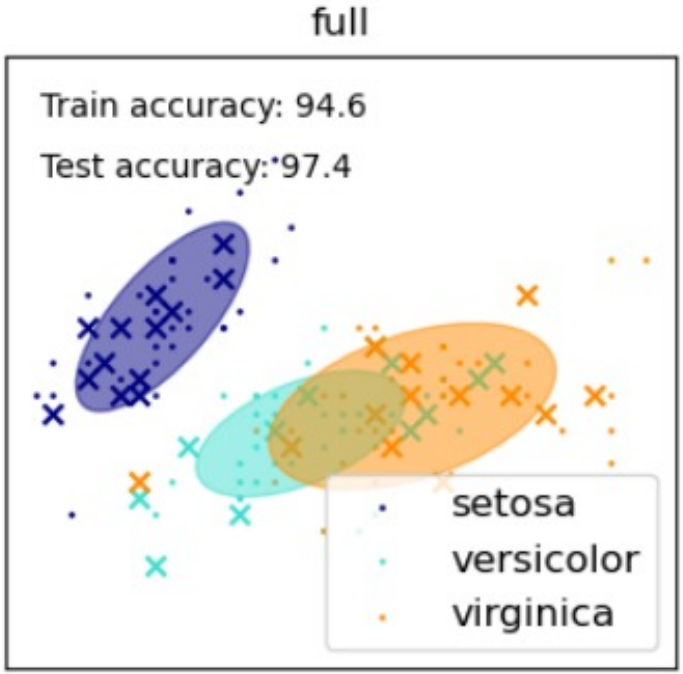
Repeat steps A and B until the centers converge to a stable solution



Parameters to estimate: (x, y) coordinates of the k cluster centers.
Observed data: The data points.
Missing data: The cluster membership (labels) of each data point.

Example: Estimating Gaussian mixture models can be done with EM

- Given the assumption that the data points were generated by a mixture of *k* Gaussians, find the parameters of the Gaussians.
- Generalizes the *k*-means algorithm by also estimating covariance structure of the data, not just means (centers)
- Parameters to estimate:
  - The (x, y) locations of the *k* Gaussian means
  - The covariance matrices of the *k* Gaussians.
- Observed data: The data points.
- Missing data: The cluster membership (labels) of each data point.



Mixture of k=3 Gaussians
on the iris dataset

Source: https://scikit-learn.org/stable/modules/mixture.html

Example: Topic models can be computed using the EM algorithm

- Probabilistic methods: Latent Dirichlet Allocation (LDA)
- Parameters to estimate:
  - The K topics $\beta_1, \beta_2, \ldots, \beta_K$
  - Per-document topic proportions $\theta_1, \theta_2, \ldots, \theta_D$
- Observed data:
  - Words in documents
- Missing data:
  - Topic assignment of each word $Z_{d,i}$ for *d* = 1..D and *i* = 1.. N
- Basic EM does not scale well to LDA <u>but</u> researchers have developed fast, incremental forms of EM (e.g. https://ieeexplore.ieee.org/document/7302081)

Best practices for use of EM

- EM will always converge.. but can get "stuck" in local optima. So try multiple randomized initializations.
- Basic EM can be very slow to converge if there are many latent variables.
- Various methods to accelerate have been developed (see paper below for some examples) and may be worth implementing depending on the scenario.
- EM appears in many guises across domains (especially NLP) and has been generalized in various ways (e.g. Generalized EM)

Source: https://arxiv.org/ftp/arxiv/papers/1301/1301.6730.pdf

Kevyn Collins-Thompson

kevynct@umich.edu

School of Information

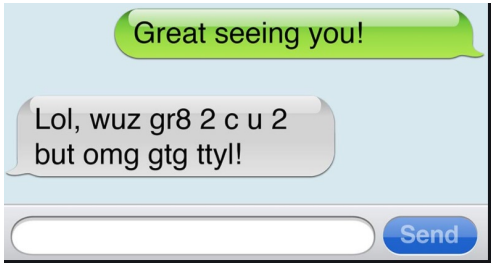
Text representations: working with textual data

Kevyn Collins-Thompson

Associate Professor of Information and Computer Science
School of Information, University of Michigan

Most text is unstructured with no predefined features or internal formatting rules

- Large variations in:
  - Style
  - Vocabulary
  - Noise/quality
  - Formatting
- We’re going to need good text preprocessing tools to apply topic modeling algorithms…



German Town Fears Ruin by U.S. Effort to Stop Russian Pipeline

Three U.S. senators and the Trump administration want to halt Gazprom’s Nord Stream 2 pipeline by imposing sanctions on a Baltic port that is supplying the project.

‘GolfGate’ Dinner in Ireland Sparks Political Backlash

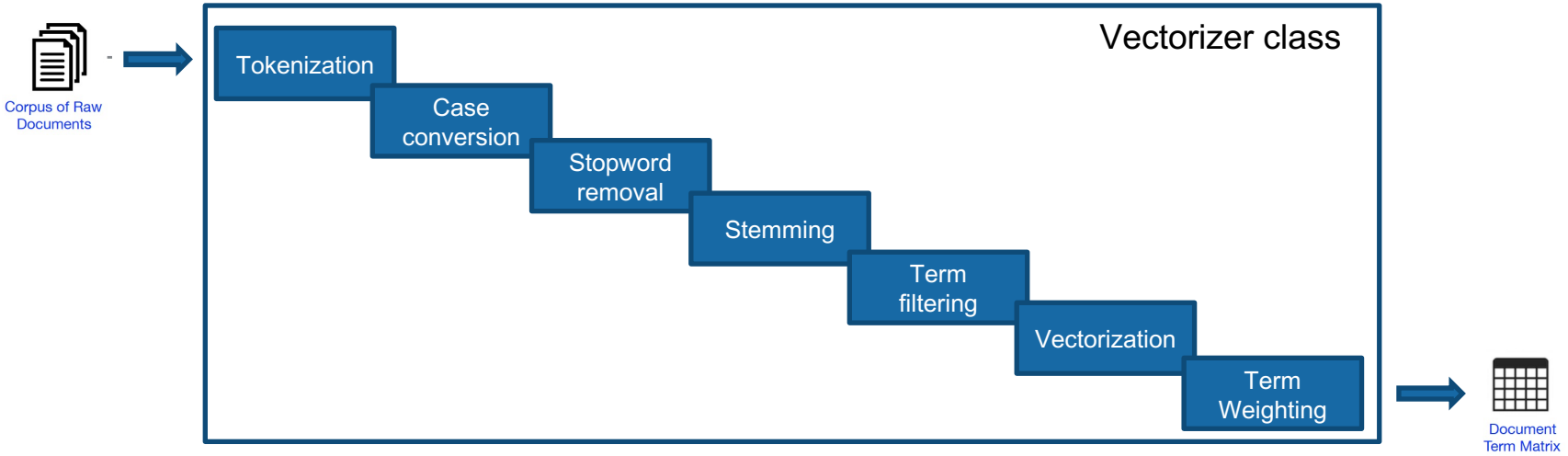
Two senior politicians have resigned, and the European Union’s trade commissioner is under pressure to quit after flouting coronavirus restrictions at a large private dinner.

‘Baby Bond’ Proposed in N.J. in Bid to Narrow the Wealth Gap

A plan would set aside a $1,000 state-financed nest egg at birth for most children, giving them a financial lift when they reach 18 and enter adulthood.

We need to convert text into numeric data.. but how?

We'll see how scikit-learn makes it easy – it's basically a one-step process.
BUT under the hood a lot has to happen…



Once we have the document-term matrix, we're ready to go with scikit-learn!

1. Tokenization: split the text into individual tokens

- For English typically this splits words on whitespace or certain punctuation

`'Breaking: The U.S. stock market is up 3% in a.m. trading!'`

`'Breaking','The','U.S.','stock','market','is','up','3%','in','a.m.','trading'`

- But doesn't work for most Asian languages (and beyond)
- Some characters can have special significance (e.g. # for tweets)

2. Character normalization / case conversion

- Often just conversion to lower case
- But could easily be more complex
  - e.g. how to handle accents? passé, Qualität

`'Breaking','The','U.S.','stock','market','is','up','3%','in','a.m.','trading'`

`'breaking','the','u.s.','stock','market','is','up','3%','in','a.m.','trading'`

3. Stopword removal

- A ‘stopword’ is a term that is very common
- e.g. In English: ‘the’, ‘of’, ‘a’, ‘an’, ‘but’, etc.
- Assumption: very common words carry little semantic meaning
- This assumption is incorrect! (e.g. phrase search: ‘the office’)
- But it helps greatly reduce the amount of text to be processed.
- Important: varies with domain. ‘biology’ may be a stopword in a biological paper corpus.
- May also consider all ‘very short’ terms (1 char) as stopwords.

‘breaking’,‘u.s.’,’stock’,’market’,’up’,’3%’,’in’,’a.m.’,’trading’

‘breaking’,‘u.s.’,’stock’,’market’,’3%’,’a.m.’,’trading’

4. Stemming

- Normalizing words to their root form
- A variant of this is called lemmatization
- Not included in scikit-learn: NLTK is standard Python library for this.

thinks → think
thinking → think
thinker → think

argue → argu
argument → argu
arguing → argu
argus → argu

‘breaking’,‘u.s.’,’stock’,’market’, ’3%’,’a.m.’,’trading’

‘break’,‘u.s.’,’stock’,’market’, ’3%’,’a.m.’,’trade’

5. Min/max frequency-based term filtering

- Term must occur a minimum of M times in the corpus
- Term must appear in min/max of M% of all corpus documents

‘break’,‘u.s.’,’stock’,’market’,’3%’,’a.m.’,’trade’

‘break’,‘u.s.’,’stock’,’market’,’trade’

From tokens to numeric features

6. Vectorization

- As we process documents in a corpus we accumulate a dictionary of all unique terms, creating a corpus vocabulary V
- With this dictionary, each document's terms are mapped to their index in V.

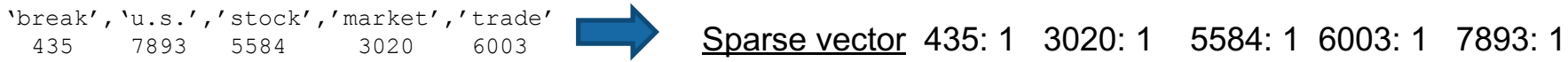
```
'break','u.s.','stock','market','trade'

 435    7893   5584     3020    6003
```

- This maps the document to a sparse vector d of dimension |V|. (Most words do not appear in most documents.)

From tokens to numeric features

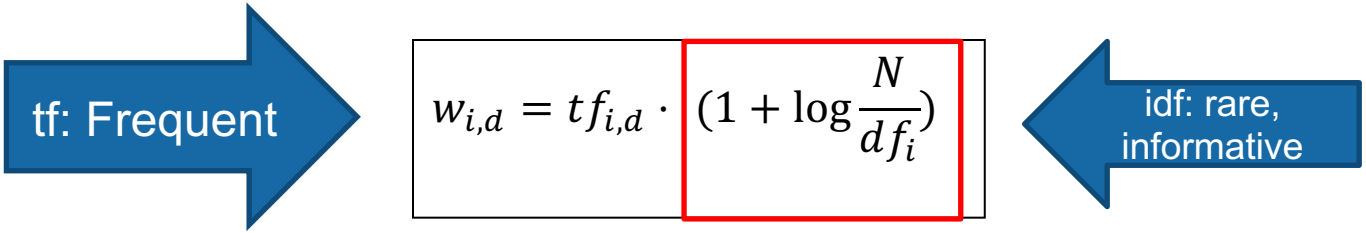
Vectorization (continued)

- This is called the 'bag-of-words' model
- Each document is represented by a (sparse) term vector *d* in a |V|-dimensional coordinate space.
- The vector element d[i] holds a statistic about the occurrence of term i in document d.
  - Could be binary (0 or 1): did the term occur at least once in d?
  - Could be integer (0 or greater): how many times did the term occur in d? (count of term frequency, or **tf**)
  - Could be a more sophisticated term weight (we will look at **tf.idf**)



Text representation: heuristic tf.idf weights combine frequency and informativeness

- Each term *i* in document *d* gets *a tf.idf* weight



$tf_{i,d}$ = frequency of term *i* in document *d*

N = total number of documents in collection

df_i = the number of documents that contain term *i*

- tf.idf combines the benefits of both weightings:
  - tf rewards terms with high frequency *within* a doc
  - idf rewards terms that are rare, discriminative *across* the whole corpus
- Note that idf assumes a collection, so it is collection-dependent!

Bag-of-words in scikit-learn

- Transforming a list of strings (documents) into a numeric document-term matrix is easy with the `CountVectorizer` class
  - Input: Documents, as a list of strings. Each string is a separate document.
  - Use `fit_transform` to process
  - Output: A sparse NumPy 2d array.
    - Rows correspond to documents
    - Columns correspond to terms
- `CountVectorizer` fills in the count of each term in the document.

```
simple_documents_train = ['The cat, dog, and duck were friends. \
                           The cat and duck met at the dog\'s house despite the dog\'s objections.',
                    'Computers have power supplies that regulate power consumption.',
                    'Plug in the monitor and turn on the computer. \
                      the monitor is now ready for use.',
                    'You will find the plug on the right side of the screen.',
                    'My friend likes coffee and cats.',
                    'His dog gets along well with my friend.']
```

```
from sklearn.feature_extraction.text import CountVectorizer

tf_vectorizer = CountVectorizer(stop_words='english')
tf_documents = tf_vectorizer.fit_transform(simple_documents_train)
```

Bag-of-words in scikit-learn

```
print("doc-term matrix shape:", tf_documents.shape)
print("doc-term matrix initial row (first document vector):\n",
      tf_documents[0:1] )
```

```
doc-term matrix shape: (6, 26)
doc-term matrix initial row (first document vector):
   (0, 0)        2
  (0, 7)         3
  (0, 8)         2
  (0, 10)        1
  (0, 14)        1
  (0, 12)        1
  (0, 6)         1
  (0, 16)        1
```

- Accessing corpus term information
  - All terms: `get_feature_names()`
  - Map from term to vector index: `vocabulary_`

```
### How many unique terms in the vocabulary?
tf_feature_names = tf_vectorizer.get_feature_names()
vocab = tf_vectorizer.vocabulary_

print("Number of unique terms in vocabulary:", len(tf_feature_names))
print("And here are the terms:\n")
for i in range(0, len(tf_feature_names)):
    print(tf_feature_names[i])

### Which column corresponds to a given term?
### Remember that the first column index is zero.
print("The term frequency of \'cats' can be found in column",
      vocab["cats"])
print("The term frequency of \'monitor' can be found in column",
      vocab["monitor"])
```

```
Number of unique terms in vocabulary: 26
And here are the terms:

cat
cats
coffee
computer
computers
consumption
despite
dog
duck
friend
friends
gets
house
likes
met
monitor
objections
plug
power
ready
regulate
right
screen
supplies
turn
use
The term frequency of 'cats' can be found in column 1
The term frequency of 'monitor' can be found in column 15
```

Accessing the vocabulary and feature names

```
tfidf_feature_names = tfidf_vectorizer.get_feature_names()
```

```
### How many unique terms in the vocabulary?
tfidf_feature_names = tfidf_vectorizer.get_feature_names()
vocab = tfidf_vectorizer.vocabulary_

print("Number of unique terms in vocabulary:", len(tfidf_feature_names))
print("Sample terms:")
for i in range(5050, 5060):
    print(tfidf_feature_names[i])
print("doc-term matrix initial row (first document vector):\n",
      tfidf_documents[0:1] )
```

```
Number of unique terms in vocabulary: 10000
Sample terms:
cso
cso uiuc
csrc
csrc ncsl
cult
cultural
culture
cup
cure
curious
```

```
doc-term matrix init
  (0, 1854)     0.17
  (0, 1916)     0.20
  (0, 7072)     0.11
  (0, 7033)     0.11
  (0, 6565)     0.13
  (0, 6366)     0.14
  (0, 8060)     0.18
  (0, 9982)     0.10
  (0, 9002)     0.18
  (0, 5618)     0.16
  (0, 7296)     0.15
  (0, 1825)     0.07
  (0, 6807)     0.07
  (0, 3335)     0.08
  (0, 4334)     0.14
  (0, 8477)     0.13
  (0, 8751)     0.16
  (0, 4422)     0.21
  (0, 3834)     0.15
  (0, 1853)     0.09
  (0, 8914)     0.12
  (0, 8271)     0.10
  (0, 5438)     0.19
  (0, 3256)     0.06
  (0, 4461)     0.11
  (0, 5501)     0.14
  (0, 6850)     0.15
  (0, 7031)     0.15
  (0, 9028)     0.19
  (0, 5437)     0.16
  (0, 1913)     0.14
  (0, 5099)     0.11
  (0, 8632)     0.14
  (0, 4490)     0.52
  (0, 9922)     0.15
```

Bag-of-words with tf.idf weights in scikit-learn

- Transforming a list of strings (documents) into a numeric document-term matrix using tf.idf weights is done using the `TfidfVectorizer` class. Almost identical in usage to `CountVectorizer`.
  - Input: Documents, as a list of strings. Each string is a separate document.
  - Use `fit_transform` to process
  - Output: A sparse NumPy 2d array.
    - Rows correspond to documents
    - Columns correspond to terms
- `TfidfVectorizer` fills in the normalized tf.idf weight of each term in the document.
- Generally for text classification, tf.idf is a better representation for text than tf or binary.

```
tfidf_vectorizer = TfidfVectorizer(max_features = 10000, # only top 10k by freq
                                   lowercase = False, # keep capitalization
                                   ngram_range = (1,2), # include 2-word phrases
                                   min_df=10,  # note: absolute count of doc
                                   max_df=0.95,   # note: % of docs
                                   stop_words='english') # default English stopwords

tfidf_documents = tfidf_vectorizer.fit_transform(documents_train)
```

More refined text preprocessing by setting CountVectorizer properties

```
from sklearn.feature_extraction.text import TfidfVectorizer

dataset = fetch_20newsgroups(subset = 'train',
                             shuffle=True,
                             random_state=42,
                             remove=('headers', 'footers', 'quotes'))
documents_train = dataset.data

tfidf_vectorizer = TfidfVectorizer(max_features = 10000, # only top 10k by freq
                                   lowercase = False, # keep capitalization
                                   ngram_range = (1,2), # include 2-word phrases
                                   min_df=10,  # note: absolute count of doc
                                   max_df=0.95,   # note: % of docs
                                   stop_words='english') # default English stopwords

tfidf_documents = tfidf_vectorizer.fit_transform(documents_train)
```

Parameter	Function
max_features	Total vocabulary size limit: only consider the top max_features terms by term frequency for the vocabulary.
lowercase	Convert all characters to lowercase before tokenizing
ngram_range	Number of tokens to consider for a term. (1,1) means only unigrams (single words). (1,2) means unigrams and bigrams, etc.
min_df	Ignore terms with document frequency strictly lower than min_df (can be absolute integer or fractional % of corpus)
max_df	Ignore terms with document frequency strictly greater than max_df (can be absolute integer or fractional % of corpus)
stopwords	Controls the stopword list used to prune very common terms. You can pass a custom list (recommended for specialized domains)

- Note that some properties like max_df allow you to specify an integer count, or a real number as fraction of the corpus.
- There are many options: see the scikit-learn documentation for details.
- **Even small changes to these settings can make a BIG difference in text classifier performance!**

Kevyn Collins-Thompson

kevynct@umich.edu

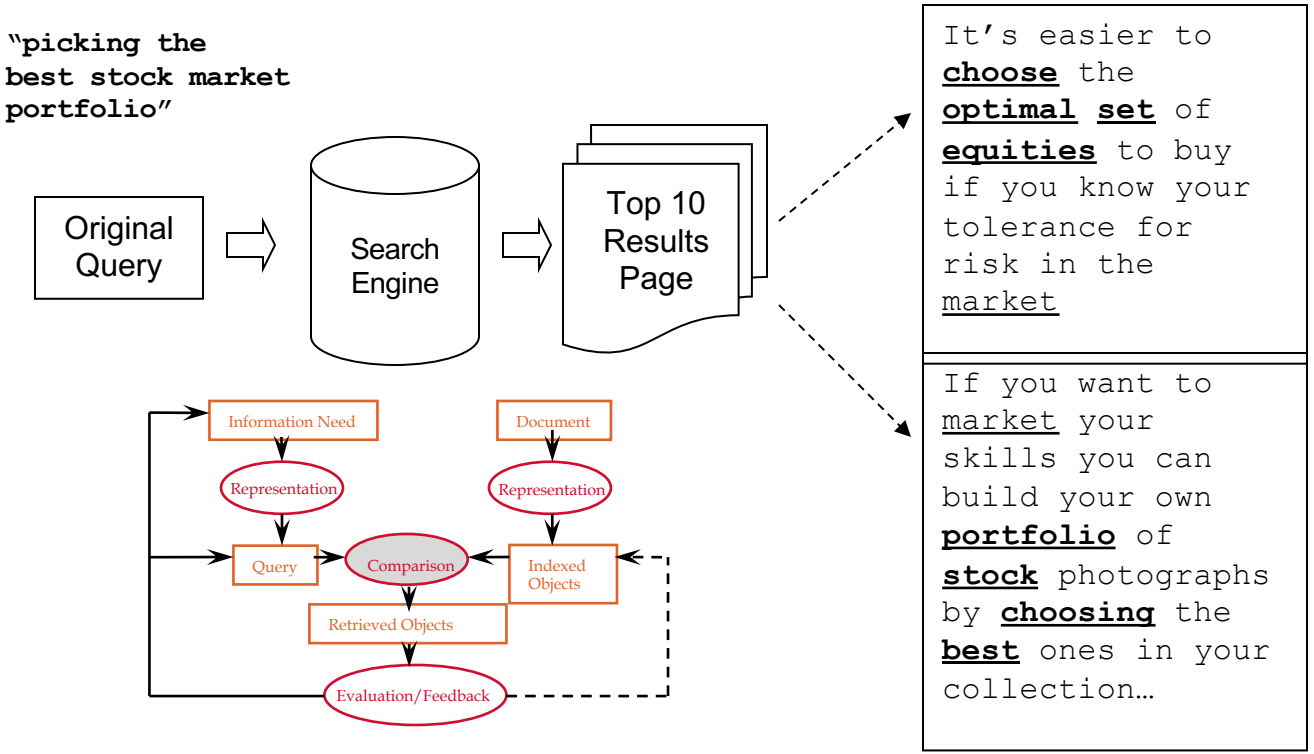
School of Information

Latent Semantic Indexing

Kevyn Collins-Thompson

Associate Professor of Information and Computer Science
School of Information, University of Michigan

Search engine challenge: Many ways to express a concept. What if query and document terms don't match? Or match incorrectly because of an incorrect word sense (polysemy)?



Latent Semantic Indexing [Deerwester, Dumais, Furnas, Landauer, Harshman. 1990]

- Idea: "**Assume there is some underlying latent semantic structure in the data that is partially obscured by the randomness of word choice with respect to retrieval. Use statistical techniques to estimate this latent structure, and get rid of the obscuring noise. A description of terms and documents based on the latent semantic structure is used for indexing and retrieval**."
- Construct a "semantic" space wherein terms and documents that are closely associated are placed near one another.
- **How? Take a large matrix of term-document association data and apply SVD.**
- Singular-value decomposition allows the arrangement of the space to reflect the major associative patterns in the data, and ignore the smaller, less important influences.
- As a result, terms that did not actually appear in a document may still end up close to the document, if that is consistent with the major patterns of association in the data.
- Position in the space then serves as a new kind of semantic indexing.
- Retrieval proceeds by using the terms in a query to identify a point in the space, and documents in its neighborhood are returned to the user.

Source: http://lsa.colorado.edu/papers/JASIS.lsi.90.pdf

LSI has been investigated for use in many different domains

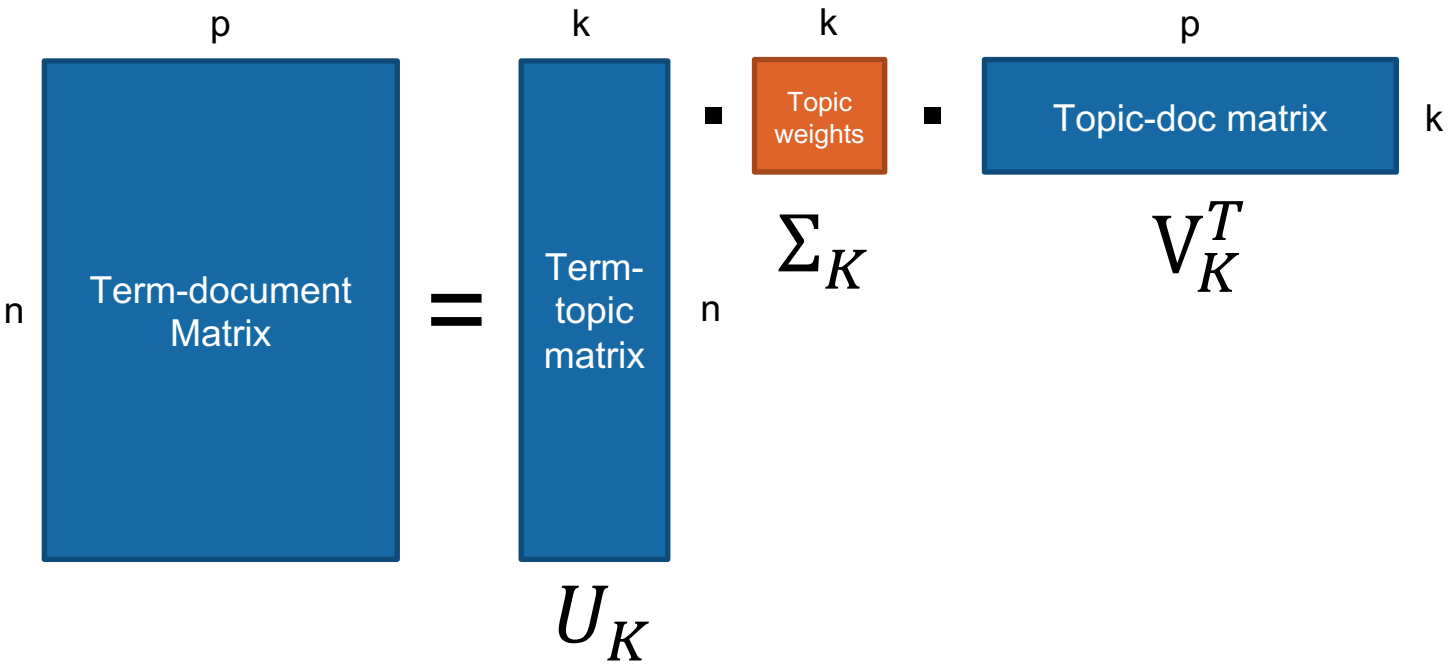
- Can be used on any corpus involving conceptual identifiers
  - Words in any language
  - Identification codes
  - Speech morphemes ...
- Semantic matching for full-text search
- Cross-language retrieval
- TOEFL/GRE synonym finding
- Matching papers to reviewers
- Finding and organizing search results
- Grouping documents into clusters
- Spam filtering
- Speech recognition
- Patent searches
- Automated essay evaluation
- Sometimes called Latent Semantic Analysis (LSA)

LSI = TruncatedSVD on tf.idf weights

- Input:
  - Sparse n x p term-document matrix X
  - Number of desired topics k
- Entries of X are tf.idf term weights
- No centering or normalization of X (sparse)
- It may be that n << p.
- Use TruncatedSVD on *X*

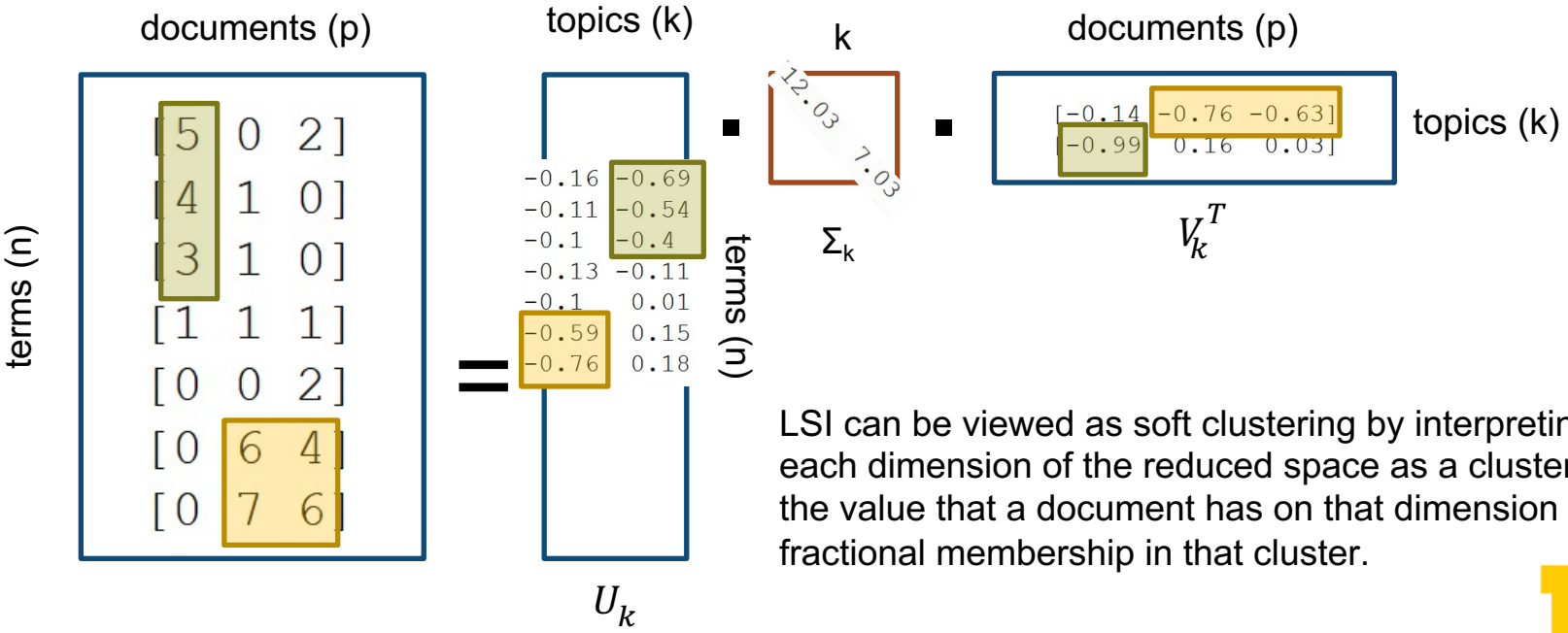
Truncated SVD applied to sparse text data = Latent Semantic Indexing

Interpretation of Truncated SVD of term-document matrix for *k* topics



SVD for text data = Latent Semantic Indexing

Interpretation of Truncated SVD of term-document matrix for *k* topics



LSI can be viewed as soft clustering by interpreting each dimension of the reduced space as a cluster and the value that a document has on that dimension as its fractional membership in that cluster.

LSI with scikit-learn
Step 1: text preparation using tf.idf vectorizer

```
from sklearn.preprocessing import Normalizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

# We use the variable that holds the newsgroup corpus that we loaded earlier.
# Initialize the TfidfVectorizer we want for this LSI example
tfidf_vectorizer = TfidfVectorizer(ngram_range = (1,1),
                                   min_df=2,
                                   max_df=0.95,
                                   stop_words='english') # default English st

tfidf_documents = tfidf_vectorizer.fit_transform(documents_train)
tfidf_feature_names = tfidf_vectorizer.get_feature_names()
```

See accompanying notebook. This example uses 20newsgroups dataset.
Assumes document list has been loaded into `documents_train`.

LSI with scikit-learn
Step 2: run truncated SVD

```
# LSI does truncated SVD on the tf.idf.
# The matrix we got back from the vectorizer is a
# document-term matrix, i.e. one row per document.
# To match the examples and development of LSI in
# our lectures, we're going to
# take the transpose of the document-term matrix to give
# TruncatedSVD the term-document matrix as input.
n_topics = 200
lsi = TruncatedSVD(n_components=n_topics, random_state=0)

# This is the matrix U_k:  num_term_features x num_topics
reduced_term_matrix = lsi.fit_transform(np.transpose(tfidf_documents))

# and this is the matrix V_k^T  num_topics x num_documents
reduced_document_matrix = lsi.components_

# these are the the values along the diagonal of matrix \Sigma.
singular_values = lsi.singular_values_

print(len(documents_train))
print(tfidf_documents.shape)
print(reduced_term_matrix.shape)
print(lsi.components_.shape)
print(lsi.singular_values_.shape)
```

```
11314
(11314, 39115)
(39115, 200)
(200, 11314)
(200,)
```

a. Run TruncatedSVD on X with k = `n_topics` $X_k = U_k \Sigma_k V_k^T$

b. Get the reduced term matrix U_k

c. Get the reduced doc matrix V_k^T

d. Get the singular values

For large collections (thousands of documents) n_topics typically 100-400

LSI with scikit-learn
Step 3: Use the output matrices to do semantically-based matching

- Suppose: Query = “economic budget” and document = “government spending on the economy”
- These are semantically very related but have zero terms in common.
- An ideal search engine to be able to match these anyway.
- Use the LSI matrices to transform q, d vectors to their reduced latent space vectors q_k, d_k
- This is done with the formulas:

$$q_k = \Sigma_k^{-1} U_k q \qquad d_k = \Sigma_k^{-1} U_k d$$

- Then we compare (e.g. using cosine similarity) q_k, d_k instead of q, d.

Did LSI find a semantic match?

```
from sklearn.metrics.pairwise import cosine_similarity

# The terms in this query and document were deliberately selected
# to NOT overlap and thus should have a cosine similarity of ZERO
# in regular term space.
similarity_original = cosine_similarity(q, d)
print("similarity score in original term space:", similarity_original)

similarity_lsi = cosine_similarity(qk, dk)
print("similarity score in LSI term space:", similarity_lsi)
```

```
similarity score in original term space: [[0.]]
similarity score in LSI term space: [[0.41]]
```

But what semantically related terms was it matching on? Let's find out by computing UU^T to get the term-term relationship matrix

```
# the term-term matrix UU^T tells us the term expansion behavior of
# this LSI model. Think of it like mapping a term to the latent space L_k,
# then back again from L_k to term space. May take a minute or two to compute.
term_term_matrix = np.dot(reduced_term_matrix, np.transpose(reduced_term_matrix))
```

```
def print_lsi_related_terms(t, top):
    term_index = tfidf_vectorizer.vocabulary_[t]
    print ("Top LSI-related terms for", t,":")
    top_related_term_indexes = term_term_matrix[term_index, :].argsort()[::-1]
    for i in range(0, top):
        this_term = top_related_term_indexes[i]
        print('\t{} ({:.2f})'.format(tfidf_feature_names[this_term],
                                     term_term_matrix[term_index, this_term]))

print_lsi_related_terms("economy", 5)
print_lsi_related_terms("spending", 5)
print_lsi_related_terms("budget", 5)
print_lsi_related_terms("gasoline", 5)
print_lsi_related_terms("reboot", 5)
```

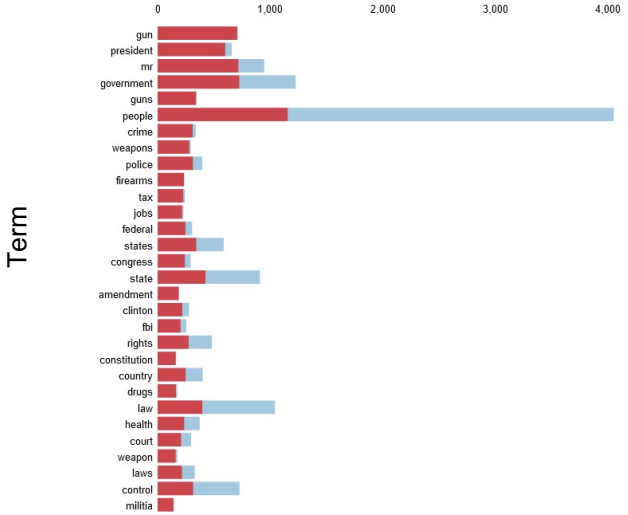
```
Top LSI-related terms for economy :
        government (0.36)
        people (0.21)
        clinton (0.16)
        president (0.15)
        money (0.14)
Top LSI-related terms for spending :
        time (0.21)
        clinton (0.19)
        government (0.19)
        people (0.17)
        just (0.17)
Top LSI-related terms for budget :
        space (0.25)
        people (0.21)
        president (0.20)
        clinton (0.18)
        government (0.18)
Top LSI-related terms for gasoline :
        car (0.05)
        engine (0.05)
        cars (0.05)
        got (0.03)
        gas (0.03)
Top LSI-related terms for reboot :
        windows (0.26)
        disk (0.24)
        drive (0.20)
        dos (0.19)
        computer (0.17)
```

- This gives insight into WHY Latent Semantic Indexing might help information retrieval by finding semantically related terms.

Applying LSI in scikit-learn: the 20 newsgroups dataset

Topics (K=10). Highlighting the 'white house' topic #4

```
topic 0: ['use', 'like', 'know', 'used', 'power', 'drive', 'new', 'don']
topic 1: ['just', 'like', 'don', 'good', 'think', 'time', 'car', 've']
topic 2: ['god', 'people', 'don', 'think', 'does', 'just', 'say', 'jesus']
topic 3: ['ax', 'max', 'g9v', 'b8f', 'a86', 'pl', '145', '1d9']
topic 4: ['people', 'government', 'mr', 'gun', 'president', 'don', 'think', 'know']
topic 5: ['edu', 'windows', 'use', 'file', 'com', 'available', 'software', 'program']
topic 6: ['space', 'file', 'university', 'program', 'nasa', '1993', 'israel', 'research']
topic 7: ['key', 'scsi', 'drive', 'chip', 'encryption', 'use', 'clipper', 'keys']
topic 8: ['game', 'team', 'year', 'db', 'armenian', 'said', 'people', 'armenians']
topic 9: ['00', '10', '25', '17', '11', '15', '16', '14']
```



Term frequency (within topic /within collection)

Top documents per topic

```
Highest topic 0 weight is document 2339 : House wiring and grounding
Highest topic 1 weight is document 1461 : Super slo-mo and frame advance
Highest topic 2 weight is document 5200 : An Introduction to Atheism
Highest topic 3 weight is document 4772 : ------------ TGZ Part 12 of 14
Highest topic 4 weight is document 6635 : White House Press Release
```

LSI Pros and Cons

- Pros:
  - Definitely addresses the vocabulary mismatch problem, by mapping documents and words to a shared concept space for matching/retrieval.
  - The concept space has much lower dimension than the original term-document matrix.
  - Increases search engine recall (and sometimes also precision)
  - Works best when little overlap expected between queries and documents
- Cons:
  - SVD on authentically large collections computationally intensive: this has limited the adoption of LSI.
  - Assumes SVD goal of finding an optimal element-wise least-squares approximation to the input: but may not be the right goal for all problems.
  - Hard to update with large numbers of new documents.

Kevyn Collins-Thompson

kevynct@umich.edu

School of Information

Introduction to Topic Modeling: Latent Dirichlet Allocation

Kevyn Collins-Thompson

Associate Professor of Information and Computer Science
School of Information, University of Michigan