TEXT MINING

Outline

- Text Analytics and NLP
- Compare Text Analytics, NLP and Text Mining
- Text Classification

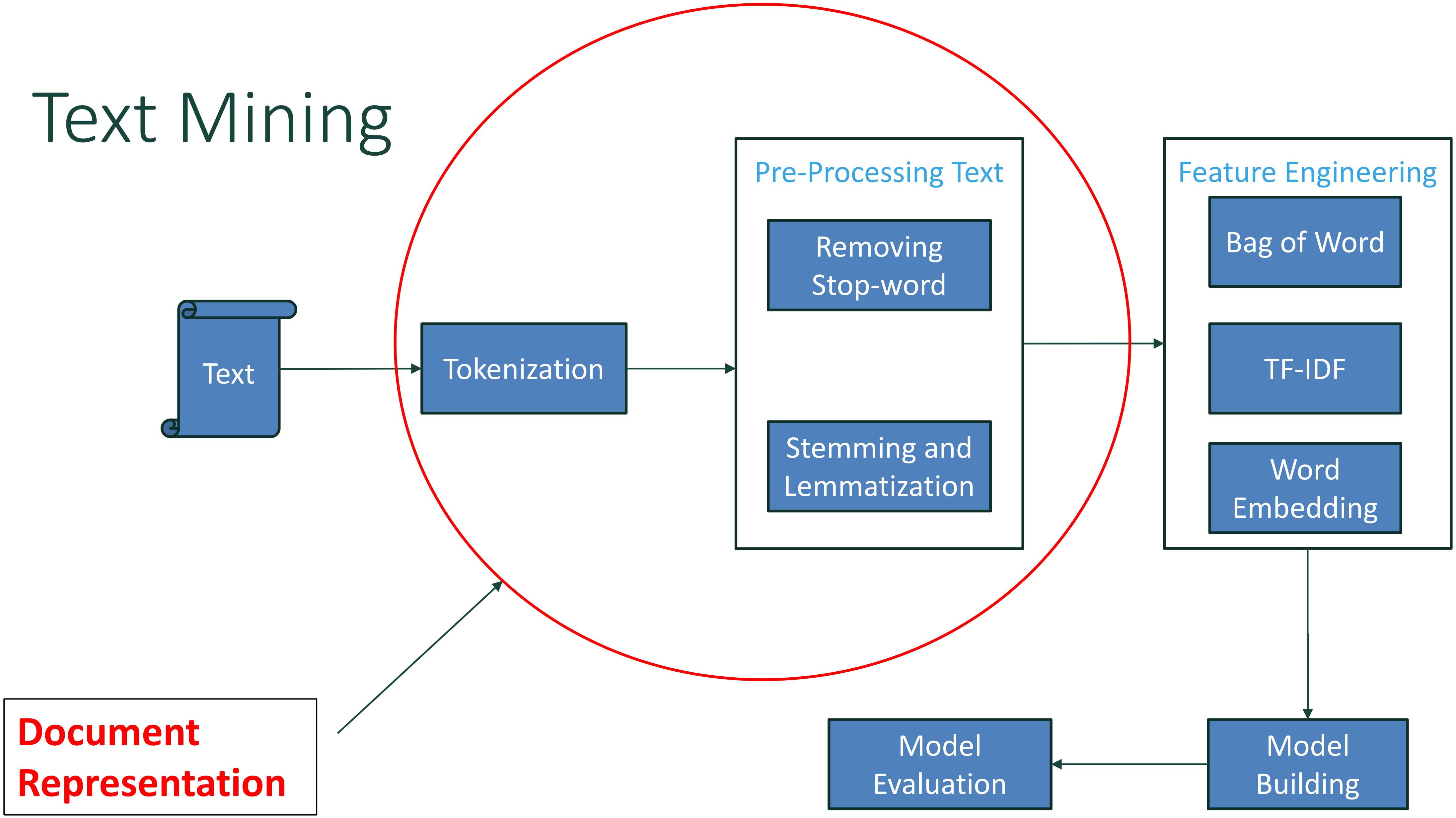
Applications

- Customer care services
  - Rapid, automated response to the customer
- Search query
  - Identify most relevant webpages to a search query
- Elections/Financial Decisions
  - Predict stock market or election outcome from social media posts
- Early Warnings
  - Alert civilians an earthquake is inbound from social media
- False information detection
  - Detect spam in email, fake news on social media

Text Analytics Vs NLP Vs Text Mining

- Text mining is a process of exploring sizeable textual data and find patterns.
  - Finding frequency counts of words,
  - length of the sentence,
  - presence/absence of specific words.
- Natural language processing is one of the components of text mining.
  - Identify sentiment,
  - Finding entities in the sentence, and
  - Finding category of blog/article.
- Text mining is preprocessing data for text analytics.
- In Text Analytics, statistical and machine learning algorithm used to classify information.

Text Mining


Text
Tokenization
Pre-Processing Text
Removing Stop-word
Stemming and Lemmatization
Feature Engineering
Bag of Word
TF-IDF
Word Embedding
Model Building
Model Evaluation
Document Representation

Text Analysis Operations using NLTK

- Document Representation:
- Applications:
  - Information Retrieval
  - Topic Modeling
  - Semantics
  - Sentiment Analysis
- NLTK helps the computer to analysis, preprocess, and understand the written text.

Getting started with nltk

- Open jupyter notebook
- Run this code:

```
import nltk
nltk.download()
```

Document Representation

For every NLP task:

- What are the features?
  - Segment/tokenize words in running text

- How to avoid curse of dimensionality?
  - Normalize word formats

- What is our unit of analysis?
  - Segment sentences in running text

Document Representation

- Tokenization: process of breaking down a text paragraph into smaller chunks such as words or sentence
- Preprocessing text:
  - Case folding, special characters, and unwanted spaces.
  - Stopwords Removal
  - Lexicon Normalization such as Stemming and Lemmatization

How many words?

- I do uh main- mainly business data processing
  - Fragments, filled pauses
- Seuss's cat in the hat is different from other cats!
  - **Lemma**: same stem, part of speech, rough word sense
    - cat and cats = same lemma
  - **Wordform**: the full inflected surface form
    - cat and cats = different wordforms

How many words?

they lay back on the San Francisco grass and looked at the stars and their

- **Type**: an element of the vocabulary.
- **Token**: an instance of that type in running text.
- How many?
  - 15 tokens
  - 13 types

at
and
back
Francisco
grass
lay
looked
on
San
stars
the
their
they

Tokenization

- Given a text file, output the word tokens and their frequencies
- Language dependent

- Simple case:
  - Replace each non-alphanumeric character with a newline character
  - Alphanumeric = letters and numbers

How many words?

N = number of tokens

V = vocabulary = set of types

|*V*| is the size of the vocabulary

	Tokens = N	**Types = |V|**
Switchboard phone conversations	2.4 million	20 thousand
Shakespeare	884,000	31 thousand
Google N-grams	1 trillion	13 million

Case folding

- Applications like IR: reduce all letters to lower case
  - Since users tend to use lower case
  - Possible exception: upper case in mid-sentence?
    - e.g., ***General Motors***
    - ***Fed*** vs. ***fed***
    - ***SAIL*** vs. ***sail***
- For sentiment analysis, Information extraction
  - Case is helpful (***US*** versus ***us*** is important)

Issues in Tokenization

- `Finland's capital` → `Finland Finlands Finland's ?`
- `what're, I'm, isn't` → `What are, I am, is not`
- `Hewlett-Packard` → `Hewlett Packard ?`
- `state-of-the-art` → `state of the art ?`
- `Lowercase` → `lower-case lowercase lower case ?`
- `San Francisco` → one token or two?
- m.p.h., PhD. → ??

Normalization

- Need to "normalize" terms
  - Information Retrieval: indexed text & query terms must have same form.
    - We want to match ***U.S.A.*** and ***USA***
- We implicitly define equivalence classes of terms
  - e.g., deleting periods in a term
- Alternative: asymmetric expansion:
  - Enter: ***window*** Search: ***window, windows***
  - Enter: ***windows*** Search: ***Windows, windows, window***
  - Enter: ***Windows*** Search: ***Windows***
- Potentially more powerful, but less efficient

Lemmatization

- Reduce inflections or variant forms to base form
  - *am, are, is* $\rightarrow$ *be*
  - *car, cars, car's, cars'* $\rightarrow$ *car*
- *the boy's cars are different colors* $\rightarrow$ *the boy car be different color*
- Lemmatization: have to find correct dictionary headword form
- Machine translation
  - Spanish quiero (‘I want’), quieres (‘you want’) same lemma as querer ‘want’

Morphology

- **Morphemes**:
  - The small meaningful units that make up words

  - **Stems**: The core meaning-bearing units

  - **Affixes**: Bits and pieces that adhere to stems
    - Often with grammatical functions

Stemming

- Reduce terms to their stems in information retrieval
- *Stemming* is crude chopping of affixes
  - language dependent
  - e.g., ***automate(s), automatic, automation*** all reduced to ***automat***.

for example compressed and compression are both accepted as equivalent to compress.

→

for exampl compress and compress ar both accept as equival to compress

Porter's algorithm
The most common English stemmer

Step 1a

```
sses → ss     caresses → caress
ies  → i      ponies   → poni
ss   → ss     caress   → caress
s    → ø      cats     → cat
```

Step 1b

```
(*v*)ing → ø  walking   → walk
              sing      → sing
(*v*)ed  → ø  plastered → plaster
…
```

Step 2 (for long stems)

```
ational→ ate  relational→ relate
izer→ ize     digitizer → digitize
ator→ ate     operator  → operate
…
```

Step 3 (for longer stems)

```
al    → ø  revival    → reviv
able  → ø  adjustable → adjust
ate   → ø  activate   → activ
…
```

Stemming Algorithm

- Porter’s algorithm:
  - Oldest stemming,
  - Most commonly used stemmer,
  - One of the most gentle stemmers,
  - Most computationally intensive.
- Snowball(Porter2):
  - An improvement over Porter,
  - Slightly faster computation time than porter.
- Lancaster:
  - Very aggressive stemming algorithm, sometimes to a fault,
  - The stemmed representations are not usually fairly intuitive to a reader,
  - The fastest algorithm of all 3,
  - Reduces set of words hugely, but if you want more distinction, not the tool you would want.

Exercise

```
from nltk.stem.porter import *
stem = PorterStemmer()
s1 = "they lay back on the grass and looked at the stars"
s2 = "the stars glowed brightly"
t1 = nltk.word_tokenize(s1)      #create list of terms in sentence 1
t2 = nltk.word_tokenize(s2)      #create list of terms in sentence 2
set1 = set()
set2 = set()
for i in t1:
      set1.add(stem.stem(i))    #stem each term in sentence 1
for i in t2:
      set2.add(stem.stem(i))    #stem each term in sentence 2
```

Exercise (cont.)

- *Sentence 1 = they lay back on the grass and looked at the stars*
- *Sentence 2 = the stars glowed brightly*
- *Set 1 = {they, lay, back, on, the, grass, and, look, at, star}*
- *Set 2 = {the, star, glow, brightli}*

	and	at	back	grass	lay	look	on	star	the	they	brightli	glow
0	1	1	1	1	1	1	1	1	2	1	0	0
1	0	0	0	0	0	0	0	1	1	0	1	1

How many tokens?

they lay back on the San Francisco grass and looked at the stars and their

- **Token**: an instance of that type in running text.
- Tokens may contain more than 1 word

- N-gram: a phrase of N words

Unigram (1-Gram)	Bigrams (2-Gram)	Trigrams (3-Gram)
At	They lay	They lay back
And	Lay back	Lay back on
Back	Back on	Back on the
Francisco	On the	On the San
Grass	The San	The San Francisco
Lay	San Francisco	San Francisco grass
Looked	Francisco grass	Francisco grass and
On	Grass and	Grass and looked
San	And looked	And looked at
Stars	Looked at	Looked at the
The	At the	At the stars
Their	The stars	The stars and
They	Stars and	Stars and their
	And their	

Stopwords

- Commonly used words that are eliminated from representation of both documents and queries
- Motivations for removal:
  - High frequency – carry little semantic weight
  - Can save considerable space

<u>Exercise</u>: what are the English stopwords?

```
import nltk
from nltk.corpus import stopwords
stop = set(stopwords.words('english’))
print(stop)
```

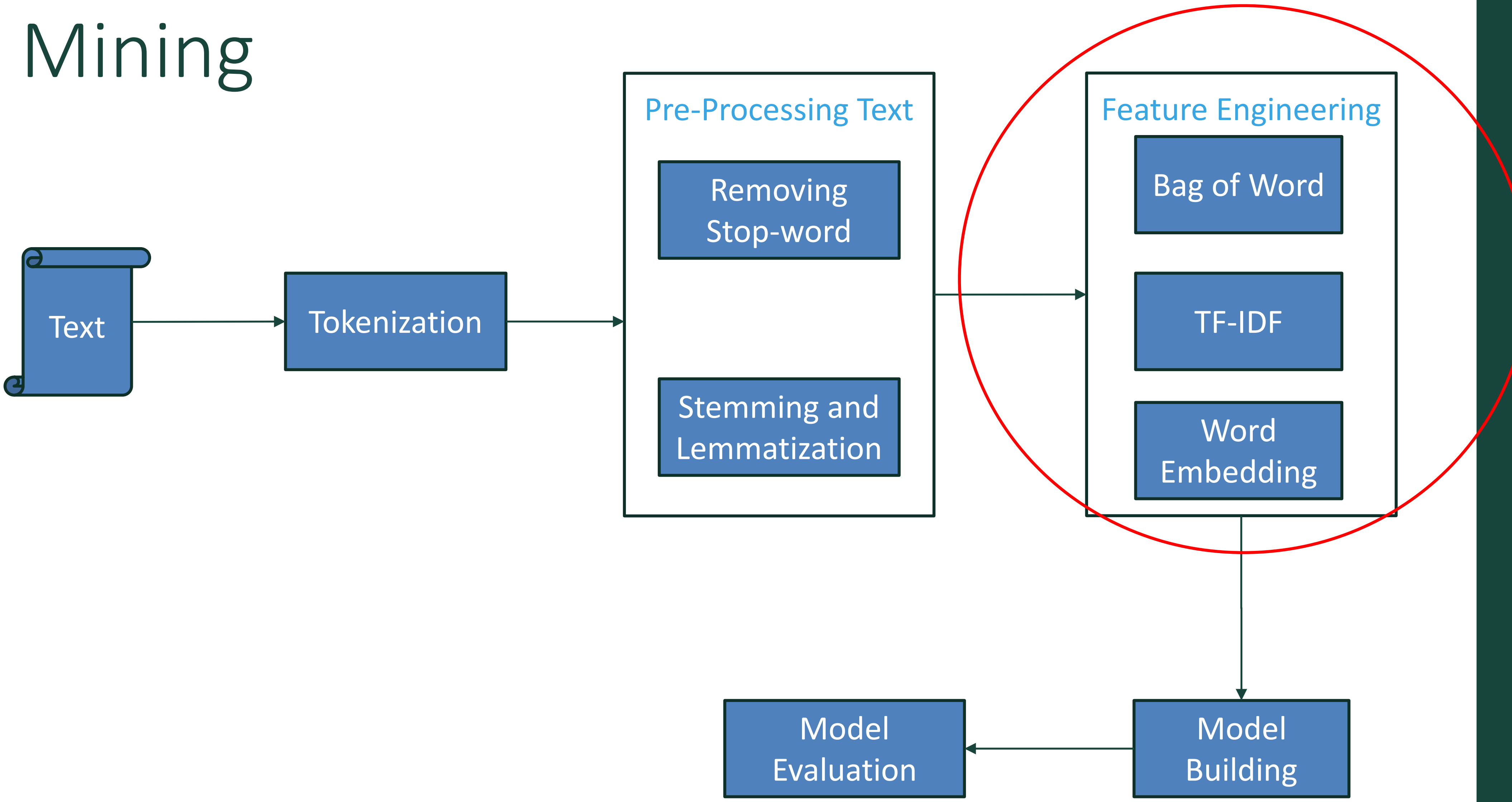
Exercise

- Remove stop words from each sentence
- Sentence1: *they lay back on the grass and looked at the stars*
- Sentence2: *the stars glowed brightly*

```
print(set1.difference(stop))
print(set2.difference(stop))
```

Lay	Back	Grass	Look	Star	Glow	Brightli
1	1	1	1	1	0	0
0	0	0	0	1	1	1

Text Mining



Text
Tokenization
Pre-Processing Text
Removing Stop-word
Stemming and Lemmatization
Feature Engineering
Bag of Word
TF-IDF
Word Embedding
Model Building
Model Evaluation

Information Retrieval

- Return a set of documents that are relevant to a query
- Simplest form of document representation is bag-of-words
  - Words are typically unigrams, but can be any length N-gram
  - Each document is represented by the count of each word

- *they lay back on the grass and looked at the stars*
- *the stars glowed brightly*

They	Lay	Back	On	The	Grass	And	Look	at	Star	Glow	Brightli
1	1	1	1	2	1	1	1	1	1	0	0
0	0	0	0	1	0	0	0	0	1	1	1

But raw frequency is a bad representation

- Frequency is clearly useful; if *sugar* appears a lot near *apricot*, that's useful information.

But overly frequent words like *the*, *it,* or *they* are not very informative about the context

Need a function that resolves this frequency paradox!

Pros and Cons of the Bag-Of-Words Approach

- Works fine for converting text to numbers.
- It assigns a score to a word based on its occurrence in a particular document.
- It doesn't take into account the fact that the word might also be having a high frequency of occurrence in other documents as well.

- TF-IDF resolves this issue by multiplying the term frequency of a word by the inverse document frequency.

tf-idf: combine two factors

- **tf: term frequency**. frequency count (usually log-transformed):

$$\mathrm{tf}_{t,d} = \begin{cases} 1+\log_{10}\mathrm{count}(t,d) & \text{if } \mathrm{count}(t,d)>0 \\ 0 & \text{otherwise} \end{cases}$$

- **Idf: inverse document frequency: tf-**

Total # of docs in collection

$$\mathrm{idf}_i = \log\left(\frac{N}{\mathrm{df}_i}\right)$$

Words like "the" or "good" have very low idf

# of docs that have word i

tf-idf value for word t in document d:

$$w_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t$$

Summary: tf-idf

- Compare two words using tf-idf cosine to see if they are similar

- Compare two documents
  - Take the centroid of vectors of all the words in the document
  - Centroid document vector is:

$$d = \frac{w_1 + w_2 + \ldots + w_k}{k}$$

Cosine for computing similarity

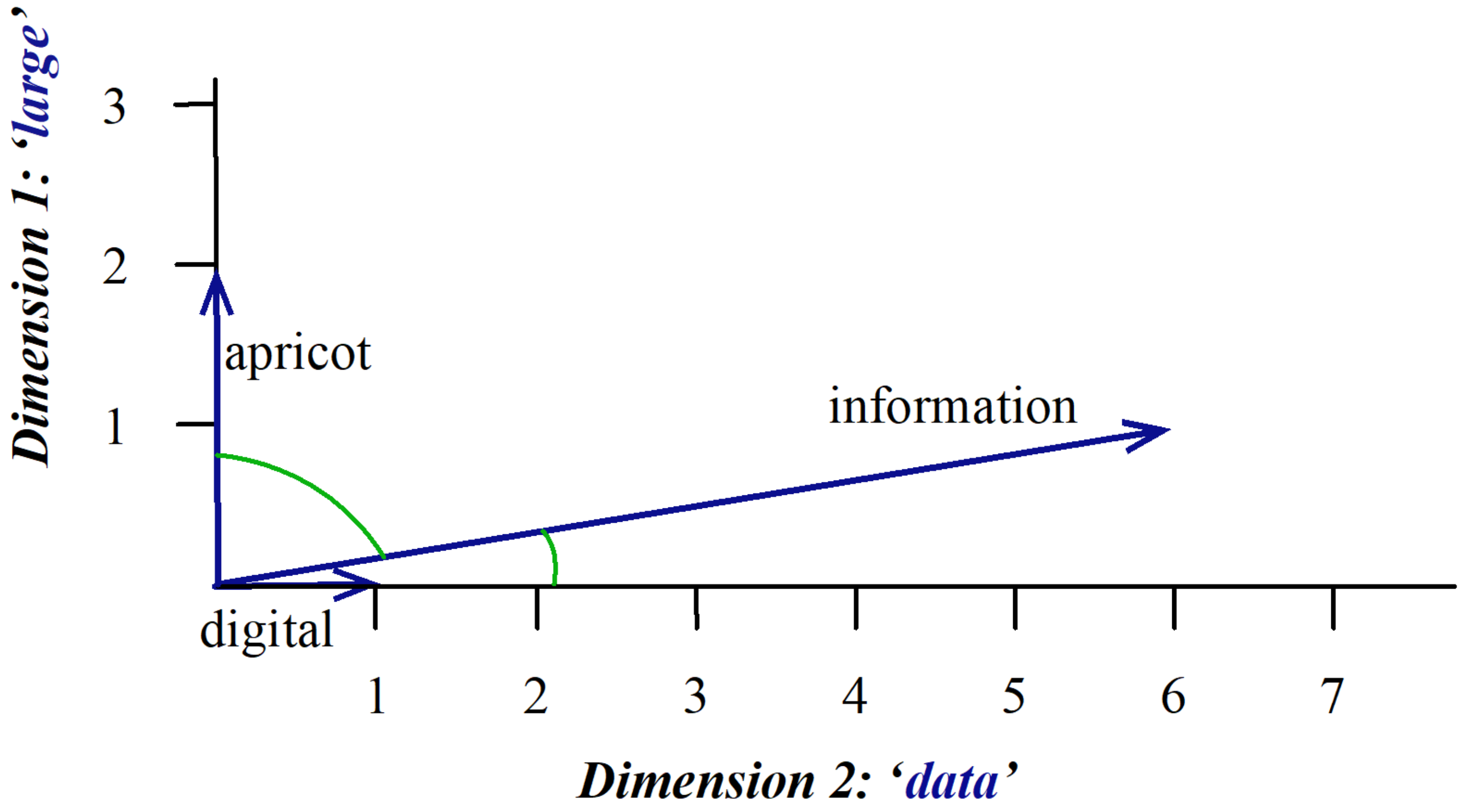
$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

$$\vec{a}\cdot\vec{b} = |\vec{a}||\vec{b}|\cos\theta$$
$$\frac{\vec{a}\cdot\vec{b}}{|\vec{a}||\vec{b}|} = \cos\theta$$

v_i is the tf-idf for word *v* in context *i*
w_i is the tf-idf for word *w* in context *i.*

Cos($\vec{v}$,$\vec{w}$) is the cosine similarity of $\vec{v}$ and $\vec{w}$

Visualizing cosines (well, angles)



Vectors

- TF-IDF vectors are
  - **long** (length |V|= 20,000 to 50,000)
  - **sparse** (most elements are zero)

  - Challenge: Curse of Dimensionality

- Alternative: dense vectors
  - **short** (length 50-1000)
  - **dense** (most elements are non-zero)

Sparse versus dense vectors

- Why dense vectors?
  - Short vectors may be easier to use as **features** in machine learning (less weights to tune)
  - Dense vectors may **generalize** better than storing explicit counts
  - **In practice, they work better**

Dense embeddings you can download!

- **Word2vec** (Mikolov et al.) (google)
- https://code.google.com/archive/p/word2vec/

- **Fasttext (Facebook)** http://www.fasttext.cc/

- **Glove (Standford)** (Pennington, Socher, Manning)
- http://nlp.stanford.edu/projects/glove/

Word2vec

- Popular embedding method
- Very fast to train
- Code available on the web
- Idea: **predict** rather than **count**

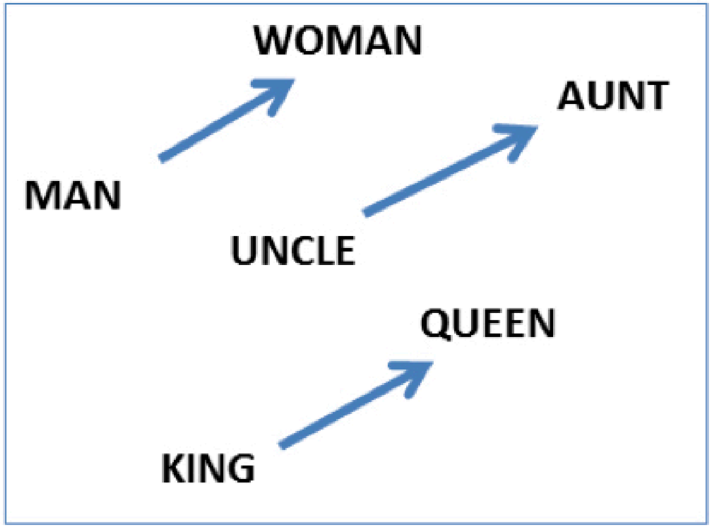
Word2vec

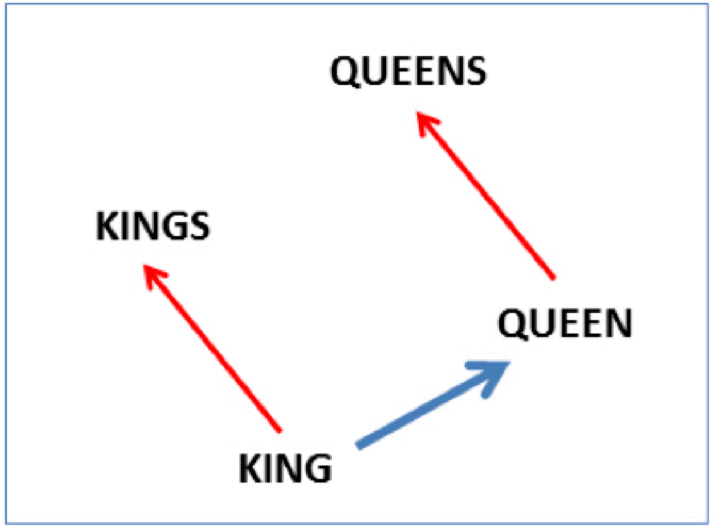
- Instead of **counting** how often each word *w* occurs near "*apricot*"
- Train a classifier on a binary **prediction** task:
  - Is *w* likely to show up near "*apricot*"?
- We don’t actually care about this task
  - But we'll take the learned classifier weights as the word embeddings

Analogy: Embeddings capture relational meaning!

vector(*‘king’*) - vector(*‘man’*) + vector(*‘woman’*) ≈ vector(‘queen’)

vector(*‘Paris’*) - vector(*‘France’*) + vector(*‘Italy’*) ≈ vector(‘Rome’)





Implementation

- Pre-trained model can be downloaded (note 1.5 GB)
- Model: GoogleNews-vectors-negative300.bin.gz
- https://code.google.com/archive/p/word2vec/

```
import gensim
model = gensim.models.KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin.gz', binary=True)
dog = model['dog’]
print(dog[:10])
print(model.similarity('woman', 'man'))
```

Exercise

- Download abcnews-date-text.csv from course website
  - Original data from Kaggle (https://www.kaggle.com/therohk/million-headlines/version/6)

```
import pandas as pd
data = pd.read_csv('abcnews-date-text.csv', error_bad_lines=False)
data_text = data[['headline_text']]
data_text['index'] = data_text.index
documents = data_text
```

conda install -c anaconda genism

Or

pip install --upgrade gensim

Exercise (cont.)

```
import gensim
from gensim.utils import simple_preprocess
from gensim.parsing.preprocessing import STOPWORDS
from nltk.stem import WordNetLemmatizer, SnowballStemmer
from nltk.stem.porter import *
import numpy as np
np.random.seed(891)
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
```

Stem and Lemmatize

```
#Create a stemmer
stemmer = SnowballStemmer(language = 'english')
#Create a lemmatizer
lemma = WordNetLemmatizer()
#Stem and lemmatize a term
def lemmatize_stemming(term):
    return stemmer.stem(lemma.lemmatize(term, pos='v'))
```

Preprocess a sentence with stopword removal

```
def preprocess(text):
    result = []
    for token in gensim.utils.simple_preprocess(text):
        if token not in stop and len(token) > 3:
            result.append(lemmatize_stemming(token))
    return result
```

Preprocess 1 Document

- Pick a number between 0 and 1,103,664. Set k to that number

```
k = 43
doc_sample = documents[documents['index'] == k].values[0][0]

print('original document: ')
words = []
for word in doc_sample.split(' '):
    words.append(word)
print(words)
print('\n\n tokenized and lemmatized document: ')
print(preprocess(doc_sample))
```

Preprocess all documents

```
processed_docs = documents['headline_text'].map(preprocess)
```

- Create a dictionary – word and its frequency in all documents

```
dictionary = gensim.corpora.Dictionary(processed_docs)
```

- Filter out infrequent terms appearing less than N times (no_below=N), terms appearing in more than 50% of documents (no_above=0.5), and keep only the top 100,000 terms (keep_n=100000)

```
dictionary.filter_extremes(no_below=15, no_above=0.5, keep_n=100000)
```

Exercise (cont.)

- Convert dictionary to document – bag of words matrix

```
bow_corpus = [dictionary.doc2bow(doc) for doc in processed_docs]
```

ADDITIONAL MATERIAL

Topic Modeling

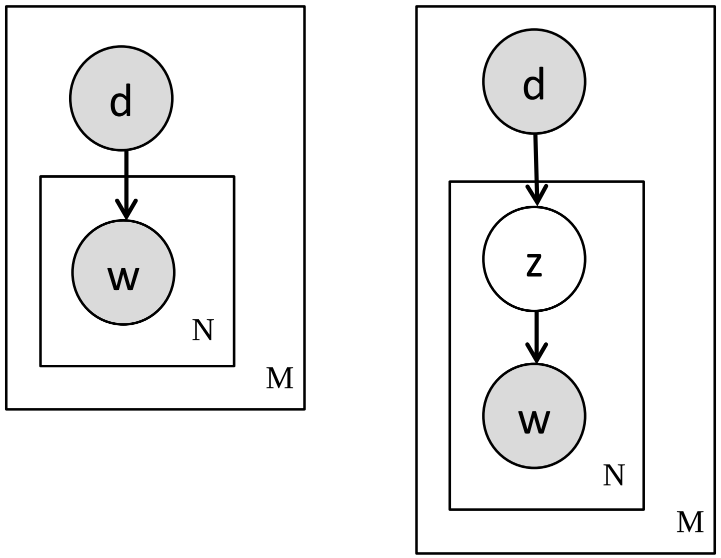
- Latent Dirichlet Allocation (LDA) is an unsupervised generative model to model a corpus of documents given the observed words are generated from hidden underlying topics
- Applications
  - Clustering
  - Queries
  - Dimension reduction
  - Classification (extension of LDA)

Probabilistic Modeling

- Modeling a document as a probability
  - Naïve Bayes
    - P(Y | X) = $\dfrac{P(X|Y)P(Y)}{P(X)}$ *Bayes Rule*
    - P(X|Y) = $\prod_{m=1}^{M} P(x_m|Y)$ *Conditional Independence*
  - Unsupervised Case: probability of a document
    - $P(D) = P(W|D)P(W)$
    - $P(W|D) = \prod_{i=1} P(w_i|D)$

Bayesian Network

- Graphical representation of probabilistic model
- Each Node is a variable
  - Shaded = observed
  - Unshaded = hidden
- Each Plate is a set of variables of the same type
  - Small plate is all words
  - Large plate is all documents



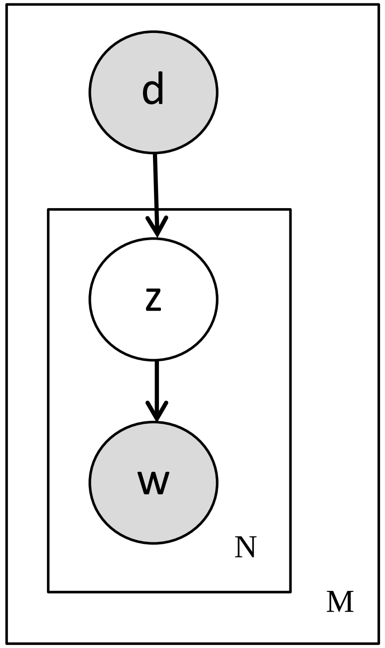
P(D,W) = $\prod p(d_i) * p(W|d_i)$
= $\prod_{i=1}^{M} p(d_i) \prod_j^N p(w_j|d_i)$

P(D,W,Z) =
$\prod_{i=1}^{M} p(d_i) \prod_j^N p(z_j|d_i) p(w_j|z_j)$

Probabilistic Latent Semantic Indexing (pLSI)

- In reality, words are not independent of each other within the same document
  - Often words relate to similar topics/themes
  - Other words present regardless of topic
- Extend model to include topic information
  - This model is a simple

    Topic Model



P(D,W,Z) =

$$\prod_{i=1}^{M} p(d_i) \prod_{j}^{N} p\left(z_j \middle| d_i\right) p(w_j|z_j)$$

Latent Dirichlet Allocation (LDA)

Extension of pLSI

Each document has a distribution of topics

Each topic has a distribution of words