

Text Analysis

Text Analysis

- Text analysis, also known as text mining, is the process of automatically classifying and extracting meaningful information from unstructured text. It involves detecting and interpreting trends and patterns to obtain relevant insights from data in just seconds.
- Let's say your team needs to analyze hundreds of online reviews to learn about the things that your clients like or dislike about your product. **Text analysis**, however, can help you automatically tag reviews by topic and classify each opinion as positive, negative, or neutral, saving your team valuable hours.
- Another term you may have heard is **text analytics**. While it's closely related to text analysis, text analytics uses data visualization tools to transform insights into quantitative data while text analysis obtains qualitative insights from unstructured text.

Text Analysis

- Text mining or text analytics is a discipline that combines language science and computer science with statistical and machine learning techniques.
- Text mining is used for analyzing texts and turning them into a more structured form. Then it takes this structured form and tries to derive insights from it.
- Example: When analyzing crime from police reports, for example, text mining helps you recognize persons, places, and types of crimes from the reports. Then this new structure is used to gain insight into the evolution of crimes

Text Analysis Example

Police reports of 10 June 2015

Danny W stole a watch in Chelsea Market

A person punched me, Xi Li, in the face in Orlando

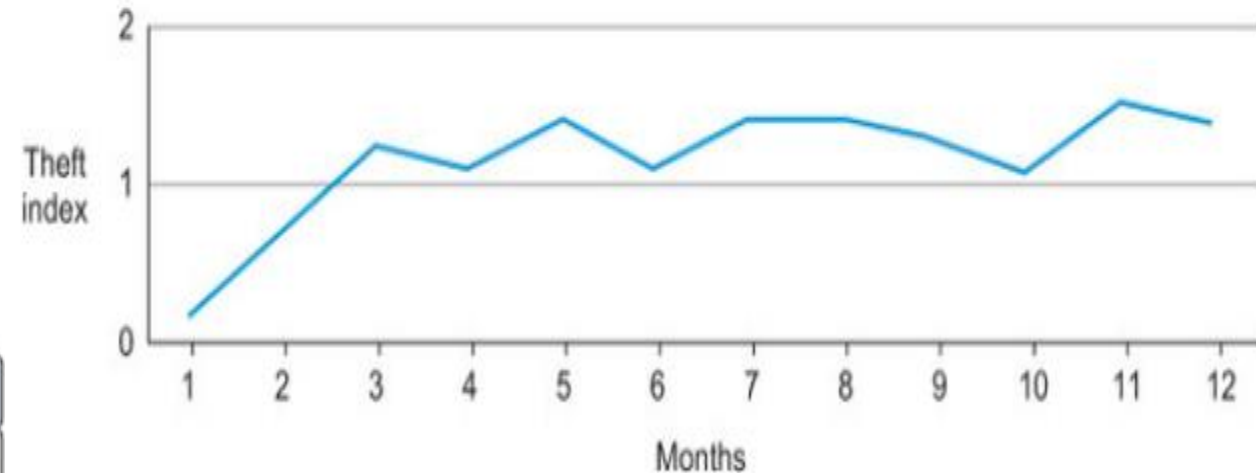
During the Chelsea soccer game, my car was stolen

Add structure

Person	Place	Crime	Victim	Date
Danny W	Chelsea Market, New York	Theft	Unknown	10th June 2015
Unknown	Orlando	Violence	Xi Li	10th June 2015
Unknown	Chelsea, London	Theft	Bart Smith	10th June 2015

Analyze and visualize

Evolution of the theft in Chelsea Market



Text Pre-processing using NLP

Text Mining in the real world

- In your day-to-day life you've already come across text mining and natural language applications. Autocomplete and spelling correctors are constantly analyzing the text you type before sending an email or text message. When Facebook autocompletes your status with the name of a friend, it does this with the help of a technique called **named entity recognition**, although this would be only one component of their repertoire. The goal isn't only to detect that you're typing a noun, but also to guess you're referring to a person and recognize who it might be.
- Another example of named entity recognition is shown in figure 8.2. Google knows Chelsea is a football club but responds differently when asked for a person.

Text Mining in the real world

Figure 8.2. The different answers to the queries “Who is Chelsea?” and “What is Chelsea?” imply that Google uses text mining techniques to answer these queries.

🔍Arno⋮

Web Images Maps News Videos More ▾ Search tools👤🔄

About 202,000,000 results (0.48 seconds)

Chelsea F.C. - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Chelsea_F.C. ▾
Chelsea Football Club / tʃɛlsɪ / are a professional football club based in Fulham, London, who play in the Premier League, the highest level of English football. Founded in 1905, the club have spent most of their history in the top tier of English football.
[2014–15 Chelsea FC season](#) - [Roman Abramovich](#) - [Stamford Bridge](#) - [Eden Hazard](#)


Chelsea, London - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Chelsea_London ▾
Chelsea is an affluent area in central London, bounded to the south by the River Thames. Its frontage runs from Chelsea Bridge along the Chelsea Embankment.

Chelsea F.C.

Football club

Chelsea Football Club are a professional football club based in Fulham, London, who play in the Premier League, the highest level of English football. Founded in 1905, the club have spent most of their history in the top tier of English football. [Wikipedia](#)

Manager: [José Mourinho](#)



Text Mining in the real world

- Google uses many types of text mining when presenting you with the results of a query. What pops up in your own mind when someone says “Chelsea”? Chelsea could be many things: a person; a soccer club; a neighborhood in Manhattan, New York or London; a food market; a flower show; and so on. Google knows this and returns different answers to the question “Who is Chelsea?” versus “What is Chelsea?” To provide the most relevant answer, Google must do (among other things) all of the following:
 - Preprocess all the documents it collects for named entities
 - Perform language identification
 - Detect what type of entity you’re referring to
 - Match a query to a result
 - Detect the type of content to return (PDF, adult-sensitive)

Text Mining in the real world

- Google uses text mining for much more than answering queries like to shielding its Gmail users from spam, it also divides the emails into different categories such as social, updates, and forums.
- Text mining has many applications, including, but not limited to, the following:
 - Entity identification
 - Plagiarism detection
 - Topic identification
 - Text clustering
 - Translation
 - Automatic text summarization
 - Fraud detection
 - Spam filtering
 - Sentiment analysis

Text Mining in the real world

- When looking at the examples, we might have gotten the impression that text mining is easy. Sadly, no. In reality text mining is a complicated task and even many seemingly simple things can't be done satisfactorily.
- **Ambiguity** is one of the most important problem that we face in a text mining application.
- Another problem is **spelling mistakes** and different (correct) spelling forms of a word. Take the following three references to New York: "NY," "Neww York," and "New York." For a human, it's easy to see they all refer to the city of New York. Because of the way our brain interprets text, understanding text with spelling mistakes comes naturally to us; people may not even notice them. But for a computer these are unrelated strings unless we use algorithms to tell it that they're referring to the same entity.
- Related problems are synonyms and the use of pronouns. **Try assigning the right person to the pronoun "she" in the next sentences:** "John gave flowers to Marleen's parents when he met her parents for the first time. She was so happy with this gesture." Easy enough, right? Not for a computer.

Example of a text data

- Emma knocked on the door. No answer. She knocked again and waited. There was a large maple tree next to the house. Emma looked up the tree and saw a giant raven perched at the tree top. Under the afternoon sun, the raven gleamed magnificently. Its beak was hard and pointed, its claws sharp and strong. It looked regal and imposing. It reigned the tree it stood on. The raven was looking straight at Emma with its beady black eyes. Emma felt slightly intimidated. She took a step back from the door and tentatively said, “Hello?”
- The paragraph contains a **lot of information**. Which parts of this paragraph of information are **salient (important) features**?
- So, we can apply **feature processing** on this piece of text

Feature

- **Feature** is numeric representation of **data**.
- Raw data can be changed to numeric measurements by many ways.
- Feature engineering is process of formulating the **most appropriate features given the data**.
- If there are **not enough informative features**, then the model will be **unable to perform the ultimate task**.
- **If there are too many features**, or if most of them are **irrelevant**, then the model will be **more expensive** and tricky to **train**.
- Features and models sit between raw data and the desired insights.

Feature

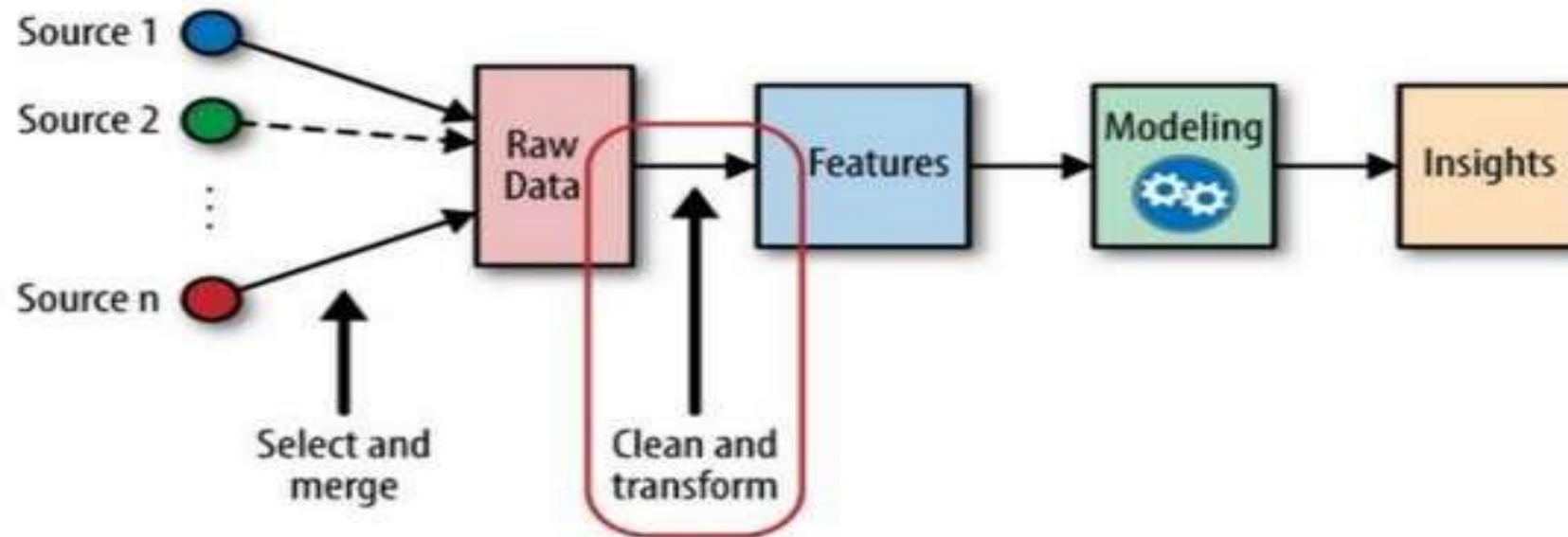


Fig. The place of feature engineering in the machine learning workflow

Feature

- In a machine learning workflow, both **model and features** are to be picked.
- It can be considered as a **double-jointed lever**, and the choice of one affects the other.
- **Good features** make the subsequent modeling step **easy and the resulting model more capable of completing the desired task**.
- Good features should not only represent salient aspects of the data, but also conform to the assumptions of the model.
- **Bad features** may require a **much more complicated** model to achieve the same level of performance.
- For selecting good features, feature engineering is required.

Feature Selection

- Feature selection techniques prune away non-useful features in order to reduce the complexity of the resulting model.
- The goal is to make a model that is quicker to compute, with little or no degradation in predictive accuracy.
- **Objectives of Feature Selection**
 - It eliminates irrelevant and noisy features by keeping the ones with minimum redundancy and maximum relevance to the target variable.
 - It reduces the computational time and complexity of training and testing a classifier, so it results in more cost-effective models.
 - It improves learning algorithms' performance, avoids overfitting, and helps to create better general models.

Feature Selection Methods

- There are three categories of feature selection methods, depending on how they interact with the classifier, namely, filter, wrapper, and embedded methods.
- **Filter methods**
 - Filtering techniques pre-process features to remove the features that are unlikely to be useful for the model.
 - For example, The correlation or mutual information between each feature and the response variable can be computed, and filter out the features that fall below a threshold.
 - Filtering techniques are much cheaper than the wrapper
 - They may not be able to select the right features for the model.

Feature Selection Methods

- **Wrapper methods** evaluate multiple models using procedures that add and/or remove predictors to find the optimal combination that maximizes model performance.
 - These procedures are normally built after the concept of Greedy Search technique (or algorithm).
 - A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.
 - Generally, three directions of procedures are possible:
 - **Forward selection** — starts with one predictor and adds more iteratively. At each subsequent iteration, the best of the remaining original predictors are added based on performance criteria.

Feature Selection Methods

- **Backward elimination** — starts with all predictors and eliminates one-by-one iteratively. One of the most popular algorithms is Recursive Feature Elimination (RFE) which eliminates less important predictors based on feature importance ranking.
- **Step-wise selection** — bi-directional, based on a combination of forward selection and backward elimination. It is considered less greedy than the previous two procedures since it does reconsider adding predictors back into the model that has been removed (and vice versa). Nonetheless, the considerations are still made based on local optimisation at any given iteration.

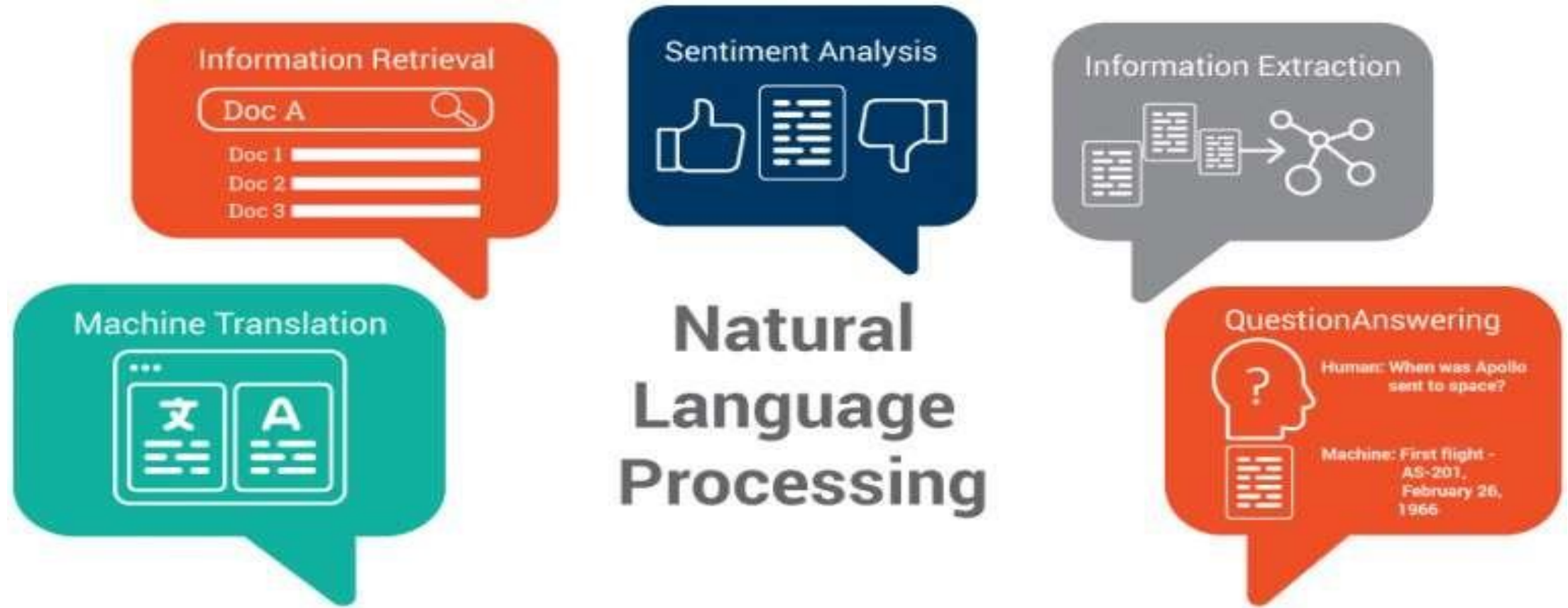
Feature Selection Methods

- **Embedded methods** bridge the gap between filters and wrappers.
 - To begin with, they fuse measurable and statistical criteria like a filter to choose some features, and then using a machine learning algorithm, they pick the subset with the best classification performance.
 - They reduce the computational complexity of wrappers without re-classifying the subsets in each iteration and can model feature dependencies. They do not perform iterations.
 - Feature selection is performed in the learning phase, meaning that these methods achieve both model fitting and feature selection at the same time.
 - One disadvantage is their dependency on the classifier.

Natural Language Processing (NLP)

- It is an area of computer science and artificial intelligence that is concerned with the **interaction between computers and humans in natural language**.
- NLP is all about enabling **computers to understand and to generate human language**.
- Applications of NLP techniques are
 - **Voice Assistants like Siri**
 - **Speech recognition,**
 - **Sentiment analysis,**
 - **Automatic text summarization,**
 - **Machine translation**

Natural Language Processing (NLP)



Text mining techniques

- Tackle the problem of text classification: automatically classifying uncategorized texts into specific categories. To get from raw textual data to our final destination we'll need a few data mining techniques that require background information for us to use them effectively. The first important concept in text mining is the “bag of words.”
- **Bag of words**
 - Bag of words is the simplest way of structuring textual data: every document is turned into a word vector. If a certain word is present in the vector it's labelled “True”; the others are labelled “False”.

Bag of words

- Figure shows a simplified example of this, in case there are only two documents. The two word vectors together form the *document-term matrix*. The document-term matrix holds a column for every term and a row for every document. The values are yours to decide upon. In this chapter we'll use binary: term is present? True or False.

Figure 8.7. A text is transformed into a bag of words by labeling each word (term) with “True” if it is present in the document and “False” if not.

Game of Thrones is a great television series but the books are better.

Doing data science is more fun than watching television.

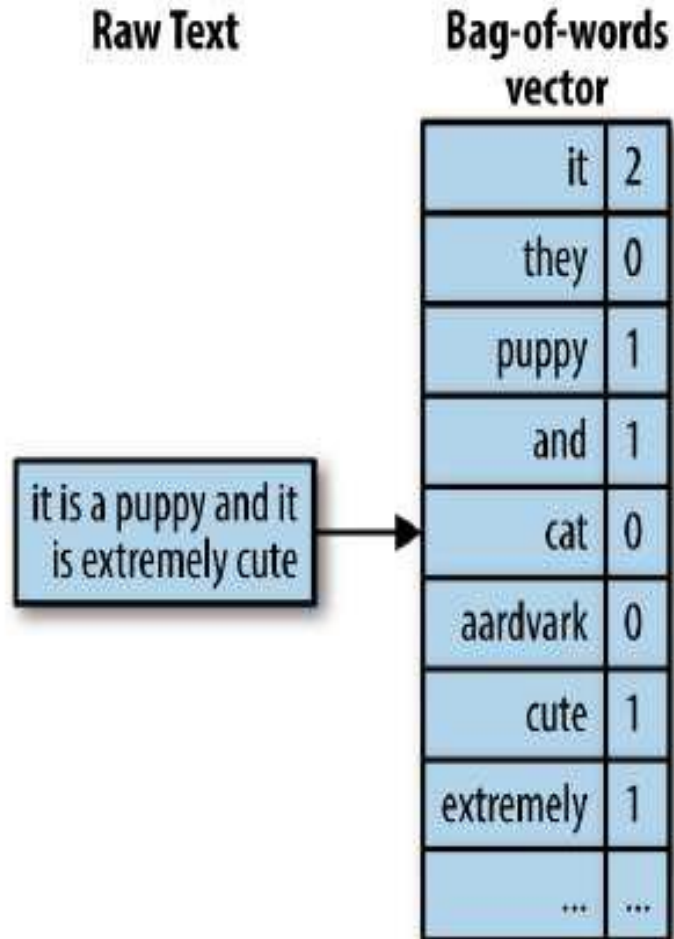
```
[({'game':True,'of':True,'thrones':True,'is':True,'a':True,
'great':True,'television':True,'series':True,'but':True,
'the':True,'books':True,'are':True,'better':True,'doing':
False,'data':False,'science':False,'more':False,'fun':False,
'than':False,'watching':False},
'gameofthrones'),
({'doing':True,'data':True,'science':True,'is':True,'more':
True,'fun':True,'than':True,'watching':True,'television':True,
'game':False,'of':False,'thrones':False,'a':False,'great':
False,'series':False,'but':False,'the':False,'books':False,
'are':False,'better':False},
'datascience')]
```

Bag-of-words

- A bag-of-words is a representation of text that describes the **occurrence of words within a document**.
- It involves two things:
 - A **vocabulary** of known **words**.
 - A **measure of the presence of known words**.
- In **bag-of-words (BoW) featurization**, a text document is converted into a **vector of counts**. (A vector is just a collection of *n numbers*.)
- *The **vector contains an entry** for every possible **word in the vocabulary**.*
- If the word—say, “is”—appears **three** times in the document, then the feature **vector has a count of 3** in the position corresponding to that word. If a word in the **vocabulary doesn't appear in the document**, then it gets a count of 0.

Bag of words

- Example



- Fig. Turning raw text into a bag-of-words representation

Bag of words

- Example: **Data:** “It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness”,

Design the Vocabulary

- The unique words here (ignoring case and punctuation) are:

- “it”
- “was”
- “the”
- “best”
- “of”
- “times”
- “worst”
- “age”
- “wisdom”
- “foolishness”

Bag-of-Words

- **Create Document Vectors**

- The objective is to turn each document of free text into a vector that we can use as input or output for a machine learning model.
- **Vector for first line:** *“It was the best of times”*
 - “it” = 1
 - “was” = 1
 - “the” = 1
 - “best” = 1
 - “of” = 1
 - “times” = 1
 - “worst” = 0
 - “age” = 0
 - “wisdom” = 0
 - “foolishness” = 0

Bag-of-Words

- EG:
 - Similarly for other lines:
 - "it was the worst of times" = [1, 1, 1, 0, 1, 1, 1, 0, 0, 0]
 - "it was the age of wisdom" = [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
 - "it was the age of foolishness" = [1, 1, 1, 0, 1, 0, 0, 1, 0, 1]
 - These docs can also be represented as **document-term** matrix:

	it	was	the	best	of	times	worst	age	wisdom	foolishness
Doc 1	1	1	1	0	1	1	1	0	0	0
Doc 2	1	1	1	0	1	0	0	1	1	0
Doc 3	1	1	1	0	1	0	0	1	0	1

Table 1 : An example document-term matrix

Bag-of-words

- Bag-of-words converts a text document into a flat vector.
- It is “flat” because it doesn’t contain any of the original textual structures.
- The original text is a sequence of words.
- But a bag-of-words has no sequence; it just remembers how many times each word appears in the text.

Bag-of-words

- **Drawbacks:**
 - Bag-of-words **is not perfect.**
 - Breaking down a sentence into single words can **destroy the semantic meaning.**
 - For instance, “not bad” semantically means “decent” or “good”
 - But “not” and “bad” constitute a **negation** plus a **negative sentiment.**
 - **“toy dog” and “dog toy”** could be very different things and the meaning is lost with the singleton words “toy” and “dog.”

Bag-of-n-Grams

- **Bag-of-n-Grams** is an extension of bag-of-words.
- Each word or token is called a “gram”.
- An **n-gram** is a **sequence of n tokens**.
- A word is a **1-gram**, also known as a **unigram**.
- An N-gram is an **N-token sequence of words**:
 - A 2-gram (bigram) is a two-word sequence of words like “please turn”, “turn your”, or “your homework”,
 - A 3-gram (trigram) is a three-word sequence of words like “please turn your”, or “turn your homework”
- After tokenization, the counting mechanism can **collate individual tokens into word counts**, or **count overlapping sequences** as n-grams.
- For example, the sentence “Emma knocked on the door” generates the n-grams “Emma knocked,” “knocked on,” “on the,” and “the door.”

Bag-of-n-Grams

- **n-grams retain more of the original sequence structure of the text,**
- **Hence, the bag-of-n-grams representation can be more informative.**
- **However, n-grams are more expensive to compute, store, and model.**
- **The larger n is, the richer the information, and the greater the cost.**

Frequency based Filtering

- the **occurrence of words** in example documents needs to be **scored** after vocabulary has been made
- **Counts:** Count the number of times each word appears in a document.
- **Frequencies:** Calculate the frequency that each word appears in a document out of all the words in the document.
- **Rare words: need to filter out rare words.** To a statistical model, a word that appears in only one or two documents is more like noise than useful information.
 - Rare words can be easily identified and trimmed based on word count statistics.
 - **Their counts can be aggregated into a special garbage bin, which can serve as an additional feature**

Frequency based Filtering

- **Frequency based Filtering**
 - Rare words:

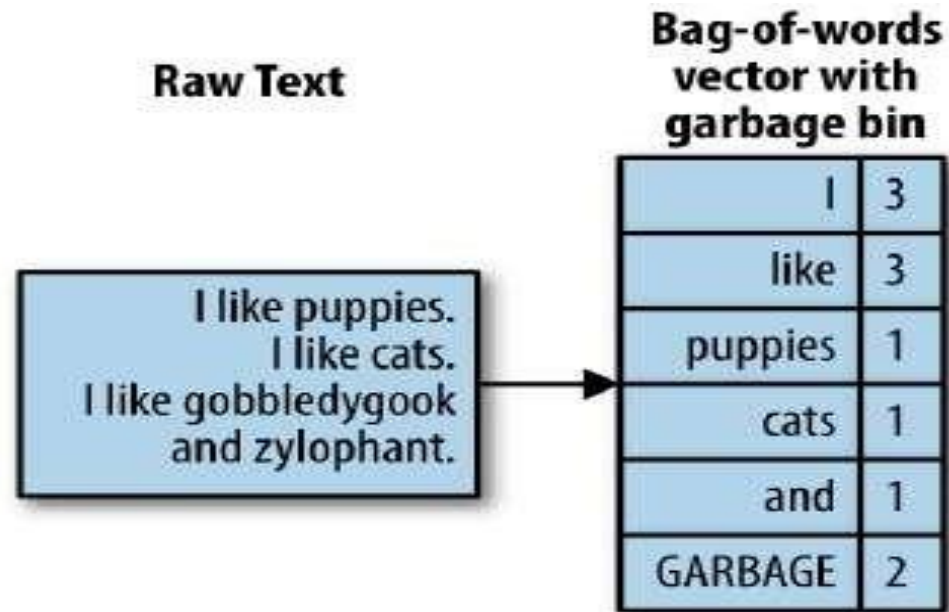


Fig. Bag-of-words feature vector with a garbage bin

Tf-Idf

- Tf-idf is a simple twist on the bag-of-words approach. It stands for **term frequency– inverse document frequency**.
- Bag of Words just creates a set of vectors containing the **count of word occurrences** in the document , while the TF-IDF model contains information on the **more important words and the less important** ones as well.
- Bag of Words vectors are easy to interpret. However, TF-IDF usually performs better in machine learning models. Hence TF-IDF is preferred.
- TF-IDF is a statistical measure **used to evaluate how important a word is to a document** in a collection or corpus.
- The tf-idf weight is composed by two terms:
 - **TF: Term Frequency**
 - **IDF: Inverse Document Frequency,**

Tf-Idf

– TF: Term Frequency

- **How frequently a term** occurs in a document.
- $TF(t) = (\text{Number of times term } t \text{ appears in a document}) / (\text{Total number of terms in the document})$.

– IDF: Inverse Document Frequency

- **measures how important a term** is
- While computing TF, all terms are considered equally important.
- However it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance.
- Thus we need to weigh down the frequent terms while scale up the rare ones, by computing the following:
- $IDF(t) = \log(\text{Total number of documents} / \text{Number of documents with term } t \text{ in it})$.

Tf-Idf

– Hence, **Tf-Idf = tf*idf**

– Eg:

- Consider a document containing 100 words where in the word *cat* appears 3 times.
- **tf for *cat* = (3 / 100) = 0.03.**
- Now, assume we have **10 million documents** and the word *cat* appears in one thousand of these.
- **idf = $\log(10,000,000 / 1,000) = 4$.**
- **Tf-idf : 0.03 * 4 = 0.12.**

Tf-Idf

- Common words like ‘is’, ‘the’, ‘a’ etc. tend to **appear quite frequently in comparison** to the words which are important to a document.
- For example, a document **on Lionel Messi is going to contain more occurrences of the word “Messi”** in comparison to other documents.
- But common words like “the” etc. **are also going to be present in higher frequency** in almost every document
- TF-IDF works by penalising these common words by assigning them lower weights **while giving importance to words like Messi in a particular document.**

Tf-Idf

Document 1

Term	Count
This	1
is	1
about	2
Messi	4

Document 2

Term	Count
This	1
is	2
about	1
Tf-idf	1

Compute TF-IDF for **THIS**:

TF = (Number of times term t appears in a document)/(Number of terms in the document)

So, $TF(\text{This}, \text{Document1}) = 1/8$

$TF(\text{This}, \text{Document2}) = 1/5$

TF denotes the contribution of the word to the document i.e words relevant to the document should be frequent.

eg: A document about Messi should contain the word 'Messi' in large number.

Tf-Idf

- $IDF = \log(N/n)$,
 - where, **N is the number of documents**
 - **n is the number of documents a term t has appeared in.**
- So, $IDF(\text{This}) = \log(2/2) = 0$.
- Why IDF?
- Ideally, **if a word has appeared in all the documents**, then probably that word is **not relevant to a particular document**.
- But if **word has appeared in a subset of documents** then probably **the word is of some relevance to the documents it is present in.**

Tf-Idf

Document 1

Term	Count
This	1
is	1
about	2
Messi	4

Document 2

Term	Count
This	1
is	2
about	1
Tf-idf	1

- Compute IDF for the word 'Messi':

$$\text{IDF} = \log(N/n),$$

where, **N** is the number of documents

n is the number of documents a term **t** has appeared in.

$$\text{IDF}(\text{Messi}) = \log(2/1) = 0.301.$$

Tf-Idf

Document 1

Term	Count
This	1
is	1
about	2
Messi	4

Document 2

Term	Count
This	1
is	2
about	1
Tf-idf	1

- Compare the TF-IDF for a common word **'This'** and a word **'Messi'** which seems to be of relevance to **Document 1**.
- $\text{TF-IDF}(\text{This}, \text{Document1}) = \text{TF} * \text{IDF} = (1/8) * (0) = 0$
- $\text{TF-IDF}(\text{Messi}, \text{Document1}) = (4/8) * 0.301 = 0.15$
- TF-IDF method heavily penalizes the word 'This' but assigns greater weight to 'Messi'.
- So, this may be understood as 'Messi' is an important word for Document1 from the context of the entire corpus.

Tokenization

- Tokenization is the process breaking complex data like paragraphs into simple units called tokens.
- **Sentence tokenization** : split a paragraph into list of sentences using `sent_tokenize()` method
- **Word tokenization** : split a sentence into list of words using `word_tokenize()` method
- Some other important terms related to word Tokenization are:
- **Bigrams**: Tokens consist of two consecutive words known as bigrams.
- **Trigrams**: Tokens consist of three consecutive words known as trigrams.
- **Ngrams**: Tokens consist of 'N' number of consecutive words known as ngrams

Tokenization

- These tokens are the most basic unit of information you'll use for your model. The terms are often words but this isn't a necessity. Entire sentences can be used for analysis.
- We'll use unigrams: terms consisting of one word. Often, however, it's useful to include bigrams (two words per token) or trigrams (three words per token) to capture extra meaning and increase the performance of your models. This does come at a cost, though, because you're building bigger term-vectors by including bigrams and/or trigrams in the equation.

Processing Text Features

- **Text cleaning techniques :**
 - Ignoring **case**
 - Ignoring **punctuation**
 - Ignoring frequent words that don't contain much information, called **stop words**, like “a,” “of,” etc.
 - Reducing words **to their stem** (e.g. “play” from “playing”).

Processing Text Features

- **Stopwords:**

- **Stopwords** refers to the most common words in a language (such as “the”, “a”, “an”, “in”) which helps in formation of sentence to make sense, but these words does not provide any significance in language processing so remove it .
- In the sentence “**Emma knocked on the door,**” the words “**on**” and “**the**” don’t change the fact that this sentence is about a person and a door
- For **coarse-grained tasks** such as classification, the pronouns, articles, and prepositions may not add much value.

Stemming

- Stemming is a normalization technique where list of tokenized words are converted into shorten root words to remove redundancy. Stemming is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form.
- A computer program that stems word may be called a stemmer.
- A stemmer reduce the words like *fishing*, *fished*, and *fisher* to the stem *fish*. The stem need not be a word, for example the Porter algorithm reduces, *argue*, *argued*, *argues*, *arguing*, and *argus* to the stem *argu* .

Stemming

- Over-stemming: Over-stemming is when two words with different stems are stemmed to the same root. This is also known as a false positive.
- **Example: universal, university, universe**
- All the above 3 words are stemmed to **univers** which is wrong behavior. Though these three words are etymologically related, their modern meanings are in widely different domains, so treating them as synonyms in NLP/NLU will likely reduce the relevance of the search results
- Under-stemming is when two words that should be stemmed to the same root are not. This is also known as a false negative. Below is the example for the same.
- **Example: alumnus, alumni, alumnae**

Lemmatization

- Major drawback of stemming is it produces **Intermediate representation** of word. **Stemmer may or may not return meaningful word.**
- To overcome this problem Lemmatization comes into picture.
- Stemming algorithm works by cutting suffix or prefix from the word. On the contrary Lemmatization consider morphological analysis of the words and **returns meaningful word** in proper form.
- Hence, **lemmatization is preferred.**

POS (Parts of Speech)

- POS (Parts of Speech) tell us about grammatical information of words of the sentence by assigning specific token (Determiner, noun, adjective , adverb ,verb,Personal Pronoun etc.) as tag (DT,NN ,JJ,VB,PRP etc) to each words.
- Word can have more than one POS depending upon context where it is used. we can use POS tags as statistical NLP tasks it distinguishes sense of word which is very helpful in text realization and infer semantic information from gives text for sentiment analysis.

POS (Parts of Speech)

```
import nltk
from nltk.tokenize import word_tokenize
data="The pink sweater fit her perfectly"
words=word_tokenize(data)
for word in words:
    print(nltk.pos_tag([word]))
```

Output:

```
[('The', 'DT')]
[('pink', 'NN')]
[('sweater', 'NN')]
[('fit', 'NN')]
[('her', 'PRP$')]
[('perfectly', 'RB')]
```

Named Entity Recognition

- The process of recognizing named entity such as person name, location name , organization name , designation of person ,quantities or values.

```
import nltk
from nltk.tokenize import word_tokenize
from nltk import ne_chunk
sentence="The US president stays in WHITE HOUSE"
sent_tokens=word_tokenize(sentence)
tags=nltk.pos_tag(sent_tokens)
NER=ne_chunk(tags)
print(NER)
```

output:

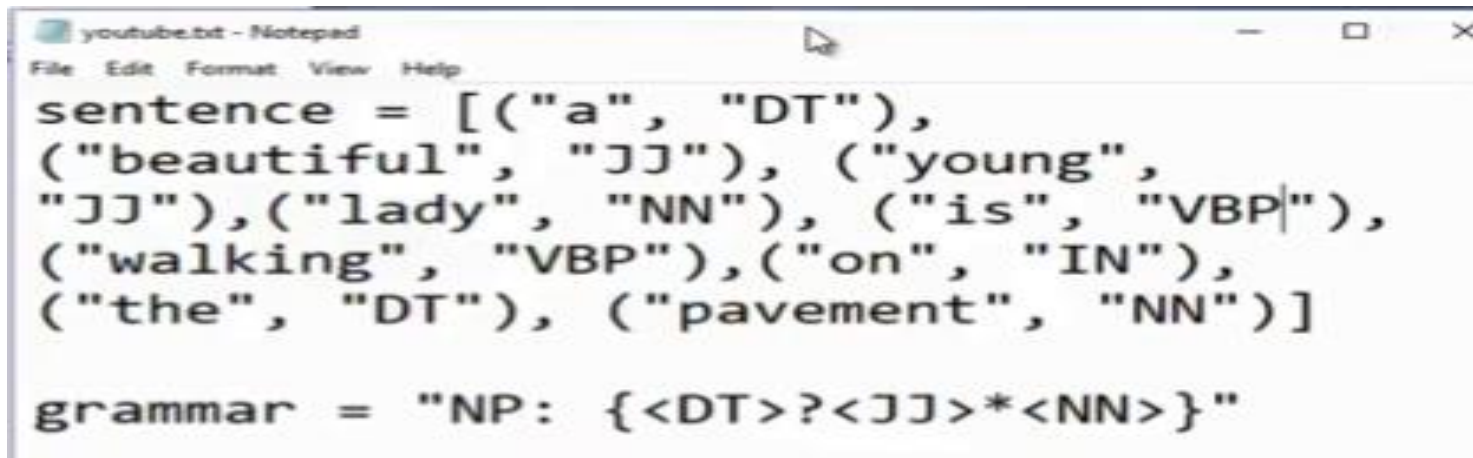
```
(S
  The/DT
  (GSP US/NNP)
  president/NN
  stays/NNS
  in/IN
  (FACILITY WHITE/NNP HOUSE/NNP))
```

Chunking

- The identification of parts of speech and short phrases (like noun phrases).
- Part of speech tagging tells you whether words are noun, verbs, adjective etc. but it does not give you any clue about the structure of the sentence or phrases in the sentence.
- Also like tokenization, the pieces produced by chunker do not overlap in source text.
- Used in Named Entity Recognition.
- Labeling of tokens present in raw text.

Chunking

- Steps to produce noun phrase chunker through NLTK module.
 - Defining a **chunker grammar**
 - Creating a **chunk parser**
 - Output will be a **tree** which shows all chunks present in the sentence.



A screenshot of a Notepad window titled 'youtube.txt - Notepad'. The window contains two lines of Python code. The first line defines a list named 'sentence' containing tuples of words and their part-of-speech tags. The second line defines a string named 'grammar' representing a noun phrase (NP) grammar rule.

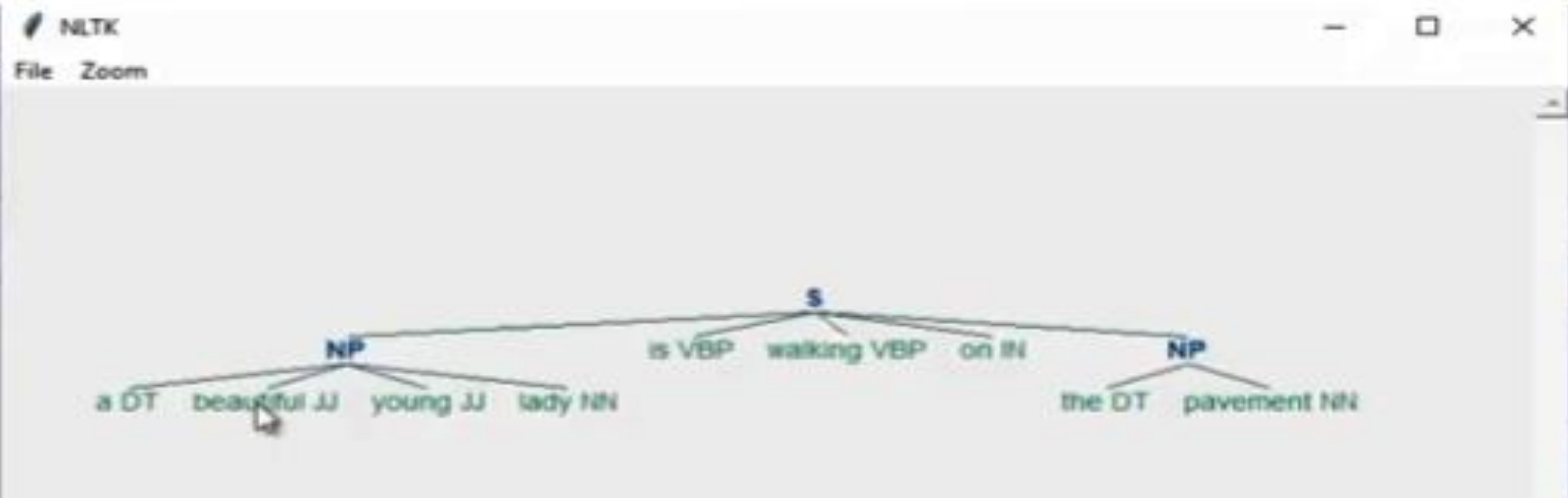
```
youtube.txt - Notepad
File Edit Format View Help
sentence = [("a", "DT"),
("beautiful", "JJ"), ("young",
"JJ"), ("lady", "NN"), ("is", "VBP"),
("walking", "VBP"), ("on", "IN"),
("the", "DT"), ("pavement", "NN")]

grammar = "NP: {<DT>?<JJ>*<NN>}"
```

Chunking

```
Python 3.4.4 Shell
File Edit Shell Debug Options Window Help

Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import nltk
>>> sentence = [("a", "DT"), ("beautiful", "JJ"), ("young", "JJ"), ("lady", "NN"),
, ("is", "VBP"), ("walking", "VBP"), ("on", "IN"), ("the", "DT"), ("pavement", "
NN")]
>>> grammar = "NP: [<DT>?<JJ>*<NN>]"
>>> parser=nltk.RegexpParser(grammar)
>>> output=parser.parse(sentence)
>>> output.draw()
```

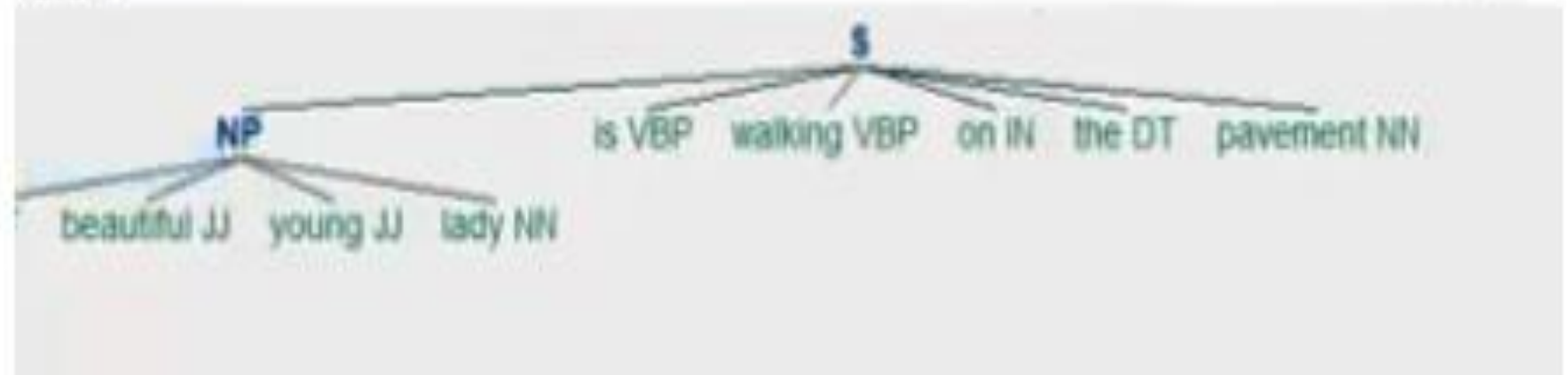


Chunking

```
Python 3.4.4 Shell
File Edit Shell Debug Options Window Help
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import nltk
>>> sentence = [("a", "DT"), ("beautiful", "JJ"), ("young", "JJ"), ("lady", "NN"),
, ("is", "VBP"), ("walking", "VBP"), ("on", "IN"), ("the", "DT"), ("pavement", "
NN")]
>>> grammar = "NP: (<DT>?<JJ>*<NN>)"
>>> parser=nltk.RegexpParser(grammar)
>>> output=parser.parse(sentence)
>>> output.draw()
>>> grammar = "NP: (<DT>?<JJ>+<NN>)"
>>> parser=nltk.RegexpParser(grammar)
>>> output=parser.parse(sentence)
>>> output.draw()
```

NLTK

Zoom



Chunking

- Chunking is the process of making a group of tokens into chunks. In simple words chunking is used as selecting the subsets of tokens. The parts of speech are combined with ***regular expressions***.
- Chunking is used for entity detection. An entity is that part of the sentence by which machine get the value for any intention
- Chunking is used to categorize different tokens into the same chunk. The result will depend on grammar which has been selected. Further chunking is used to tag patterns and to explore text corpora.

Feature Scaling

- In many machine learning algorithms, **to bring all features in the same standing**, we need to do scaling so that one significant number doesn't impact the model just because of their large magnitude.
- The machine learning algorithm works on numbers and does not know what that number represents. A weight of 10 grams and a price of 10 dollars represents completely two different things — which is a no brainer for humans, but for a model as a feature, it treats both as same.
- The “Weight” cannot have a meaningful comparison with the “Price.” So the assumption algorithm makes that since “Weight” > “Price,” thus “Weight,” is more important than “Price.”

Feature Scaling

- So these more significant number starts playing a more decisive role while training the model. Thus feature scaling is needed **to bring every feature in the same footing without any upfront importance.**
- One more reason is **saturation**, like in the case of sigmoid activation in Neural Network, scaling would help not to saturate too fast.
- Feature scaling in machine learning is one of the most critical steps during the pre-processing of data before creating a machine learning model. Scaling can make a difference between a weak machine learning model and a better one

Feature Scaling

- The most common techniques of feature scaling are Normalization and Standardization.
- **Normalization** is used when we want to bound our values between two numbers, typically, between $[0,1]$ or $[-1,1]$. While **Standardization** transforms the data to have zero mean and a variance of 1, they make our data **unitless**.
- Algorithms that do not require normalization/scaling are the ones that **rely on rules**. They would not be affected by any monotonic transformations of the variables. Scaling is a monotonic transformation. Examples of algorithms in this category are all the tree-based algorithms — **CART, Random Forests, Gradient Boosted Decision Trees**. These algorithms utilize rules and **do not require normalization**.

Feature Scaling

- We must have always faced this question of whether to Normalize or to Standardize. While there is no obvious answer to this question, it really depends on the application
- Normalization is good to use when,
 - In Neural Networks algorithm that require data on a 0–1 scale, normalization is an essential pre-processing step. Another popular example of data normalization is image processing, where pixel intensities have to be normalized to fit within a certain range (i.e., 0 to 255 for the RGB color range).
- Standardization can be helpful in cases where the data follows a Gaussian distribution. Though this does not have to be necessarily true. Since standardization does not have a bounding range, so, even if there are outliers in the data, they will not be affected by standardization.

Feature Scaling

- There are some points which can be considered while deciding whether we need Standardization or Normalization
 - Standardization may be used when data represent Gaussian Distribution, while Normalization is great with Non-Gaussian Distribution
 - Impact of Outliers is very high in Normalization

Dimensionality Reduction

- As we can imagine, when we have real-world documents the number of the dimensions would increase dramatically like a few thousand because the **number of the dimensions equals to the number of unique terms** in the entire set of documents. We call this type of data 'high-dimensionality' data.
- Another thing you would notice by looking at the table is that there are many cells that are left empty because all of the terms don't necessarily exist in all the documents. We call this type of data '**sparse**' data or '**sparse matrix**'.
- Calculating the distance among all the documents from this type of high-dimensional and sparse data with algorithms like 'K-means' would have a problem called '[Curse of dimensionality](#)'.

Dimensionality Reduction

- Problems with Bag-of-words or N-gram
 - Curse of Dimensionality (too many features)
 - Example: Find sentiment of tweets (tweet has 280 characters or 50+ words). So when we are working with 10,000 tweets, then we have huge matrix of features.

Illinois is also known as Land of Lincoln. Abraham Lincoln, Statesman and Lawyer, served as the 16th President of the United States.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Illinois	is	also	known	as	land	of	Lincoln	Abraham	statesman	and	lawyer	served	the	16th	president	United	States
1	1	1	1	2	1	2	2	1	1	1	1	1	2	1	1	1	1

Dimensionality Reduction

- Example: When we have only 4 sentences but when we plot them with bag-of-words then it transforms into matrix of 12 features though we have max. 4 words in each sentence.

[illegible]

Dimensionality Reduction

- **Need:** As the number of features increase, the number of samples also increases proportionally. The more features we have, the more number of samples we will need to have all combinations of feature values well represented in our sample.
- As the number of features increases, the model becomes more complex. The more the number of features, the more the chances of overfitting. resulting in poor performance on real data, beating the purpose.

Dimensionality Reduction

- Avoiding overfitting is a major motivation for performing dimensionality reduction. The fewer features our training data has, the lesser assumptions our model makes and the simpler it will be. But that is not all and dimensionality reduction has a lot more advantages to offer, like
 - Less misleading data means model accuracy improves.
 - Less dimensions mean less computing. Less data means that algorithms train faster.
 - Less data means less storage space required.
 - Less dimensions allow usage of algorithms unfit for a large number of dimensions
 - Removes redundant features and noise.

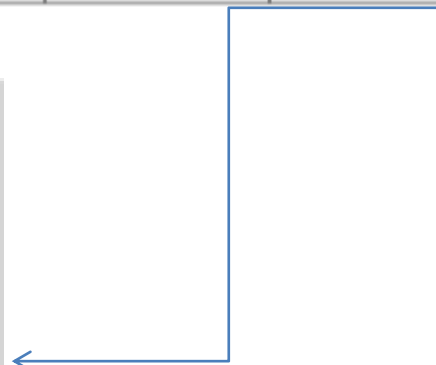
Dimensionality Reduction

- Can we compact this high dimensional and sparse data into a data that is represented by only a **few dimensions without losing** much of the original **information**? If we can, then maybe we can feed such reduced data into the clustering or any algorithm.
- Dimensionality reduction is, the process of reducing the dimension of your feature set. Your feature set could be a dataset with a hundred columns (i.e features) or it could be an array of points that make up a large sphere in the three-dimensional space.
- Dimensionality reduction is bringing the number of columns down to say, twenty or converting the sphere to a circle in the two-dimensional space.

Dimensionality Reduction

	a	an	apple	ate	banana	eat	i	today	will	yesterday
Doc 1		0.0811	0.0811	0.0811			0			0.0811
Doc 2		0.0676	0.0676			0.1831	0	0.1831	0.1831	
Doc 3	0.2197			0.0811	0.2197		0			0.0811

	1	2	3
Doc 1	0.0144	0.8164	0.5774
Doc 2	-0.7142	-0.3957	0.5774
Doc 3	0.6998	-0.4207	0.5774

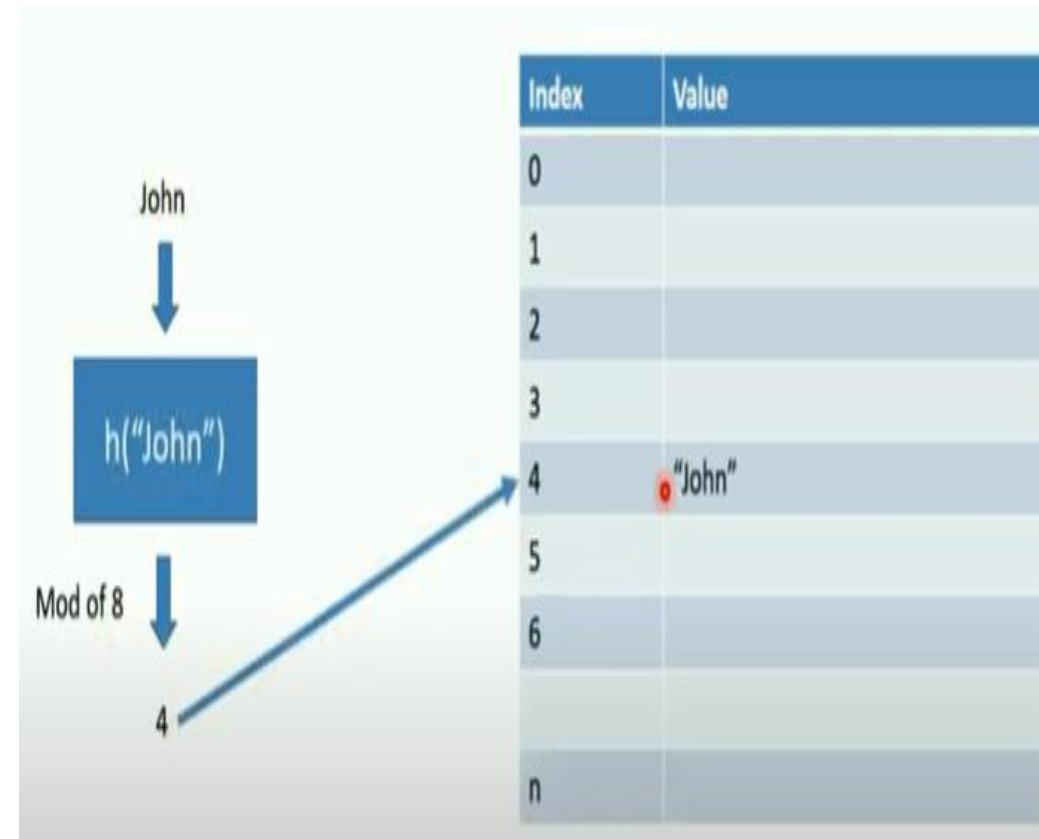
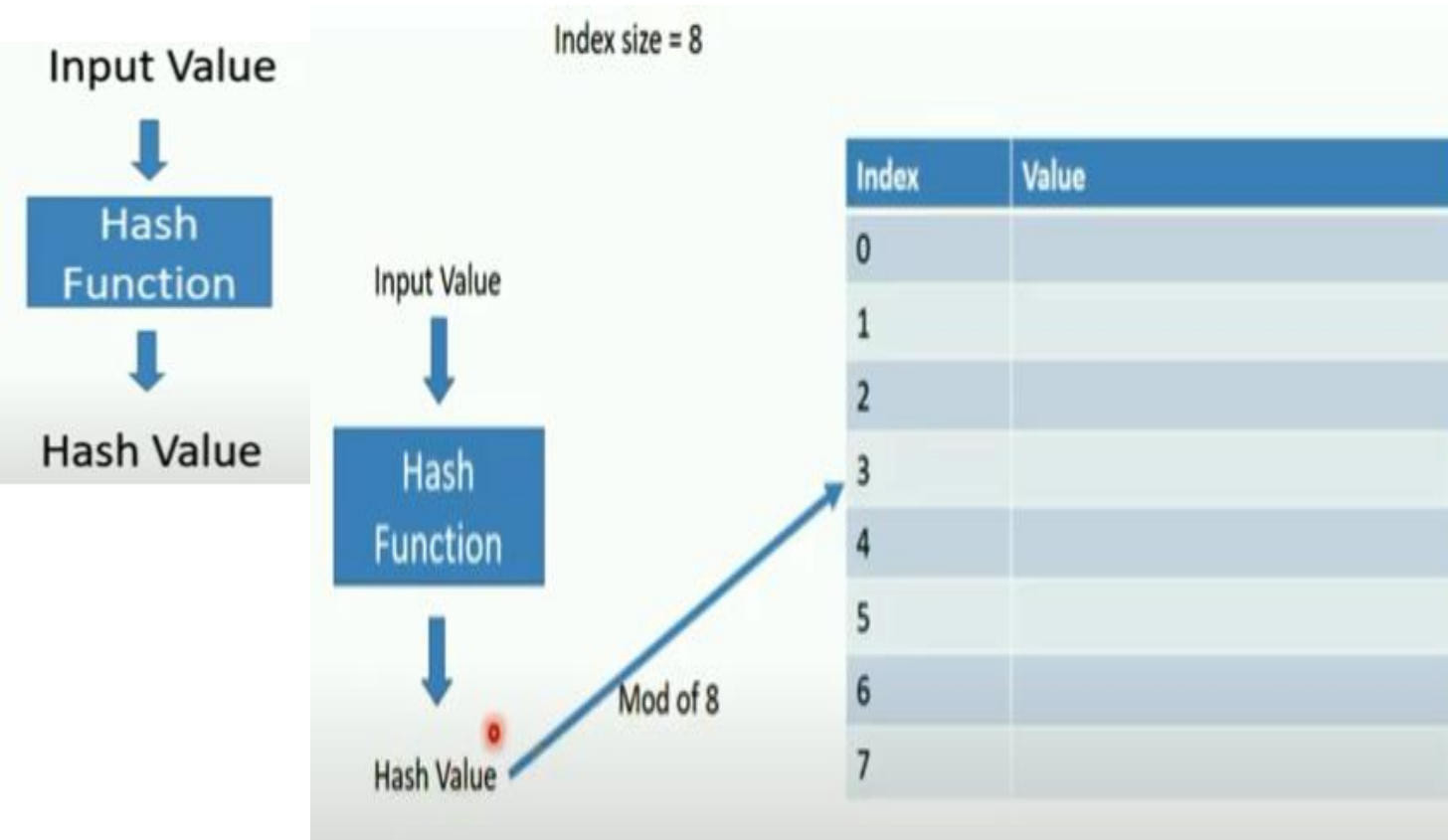


Dimensionality Reduction

- Dimensionality reduction could be done by both feature selection methods as well as feature engineering methods.
- Feature selection is the process of identifying and selecting relevant features for your sample. Feature engineering is manually generating new features from existing features, by applying some transformation or performing some operation on them.

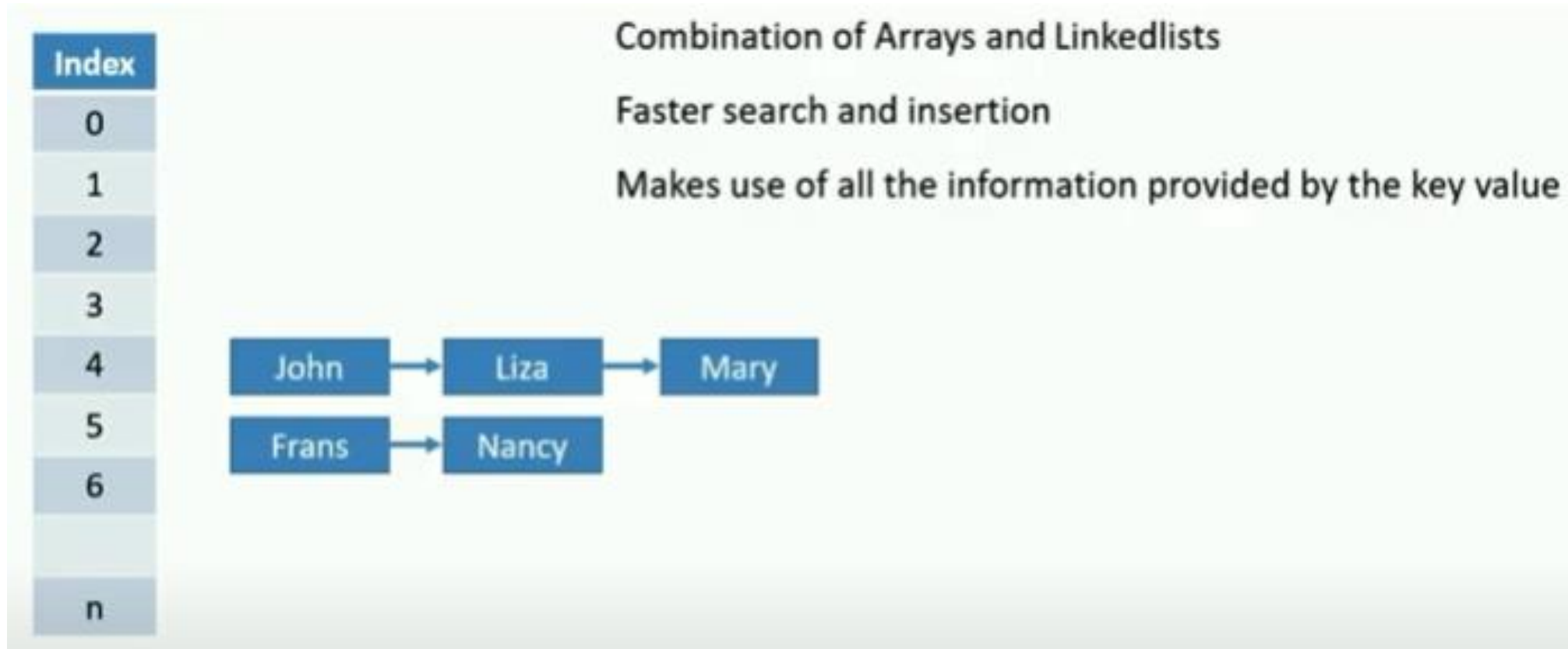
Feature Hashing

- Hashing: Hash is simply fixed length number or string that is generated by the hashing algorithm because we are trying to generate indices of table or array; we use hashing algorithms.



Feature Hashing

- Hash table with separate chaining



Feature Hashing

- No. of extracted features = mod of number

Sentence	Murmurhash3	Divide by	Reminder
john	3487894951	8	7
likes	1103617568	8	0
movies	3188341541	8	5

Index	Value
0	likes
1	
2	
3	
4	
5	movies
6	
7	john

Sentence	Murmurhash3	Divide by	Reminder
john	3487894951	8	7
likes	1103617568	8	0
movies	3188341541	8	5

Index	Value
0	likes
1	
2	
3	
4	
5	movies
6	
7	john

Value	likes					movies		john
Index	0	1	2	3	4	5	6	7
Feature	1	2	3	4	5	6	7	8

Comparing bag-of-word with feature Hashing Model

- No. of extracted features = mod of number

Bag Of Words

	John	likes	movies	Liza	watching	Frans	loves	playing	football	merry	enjoys	action
John likes movies	1	1	1	0	0	0	0	0	0	0	0	0
Liza likes watching movies	0	1	1	1	1	0	0	0	0	0	0	0
Frans loves playing football	0	0	0	0	0	1	1	1	1	0	0	0
Merry enjoys action movies	0	0	1	0	0	0	0	0	0	1	1	1

Feature Hashing

	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Feature 7	Feature 8
John likes movies	1	0	0	0	0	2	0	0
Liza likes watching movies	2	0	1	0	0	1	0	0
Frans loves playing football	0	1	1	0	1	0	0	1
Merry enjoys action movies	0	0	0	0	0	3	0	1

Feature Hashing

- Also known as **hashing trick** and Used for **vectorizing features**
- It works by applying a **hash function** to the features and using their **hash values as indices directly**.
- A **hash function** is a **deterministic** function that **maps a potentially unbounded integer to a finite integer range $[1, m]$** .

Feature Hashing

- Need
 - Many domains have **very high feature dimension**.
 - Some feature domains are **naturally dense** (for eg, **images and video**).
 - This creates a challenge at **scale** because even simple models can become very large, **Consume more memory and resources** during training and when making predictions in production systems.
 - Instead of maintaining a **one-to-one mapping of categorical feature values to locations in the feature vector**, we use a **hash function** to determine **the feature's location in a vector dimension**.

Feature Hashing

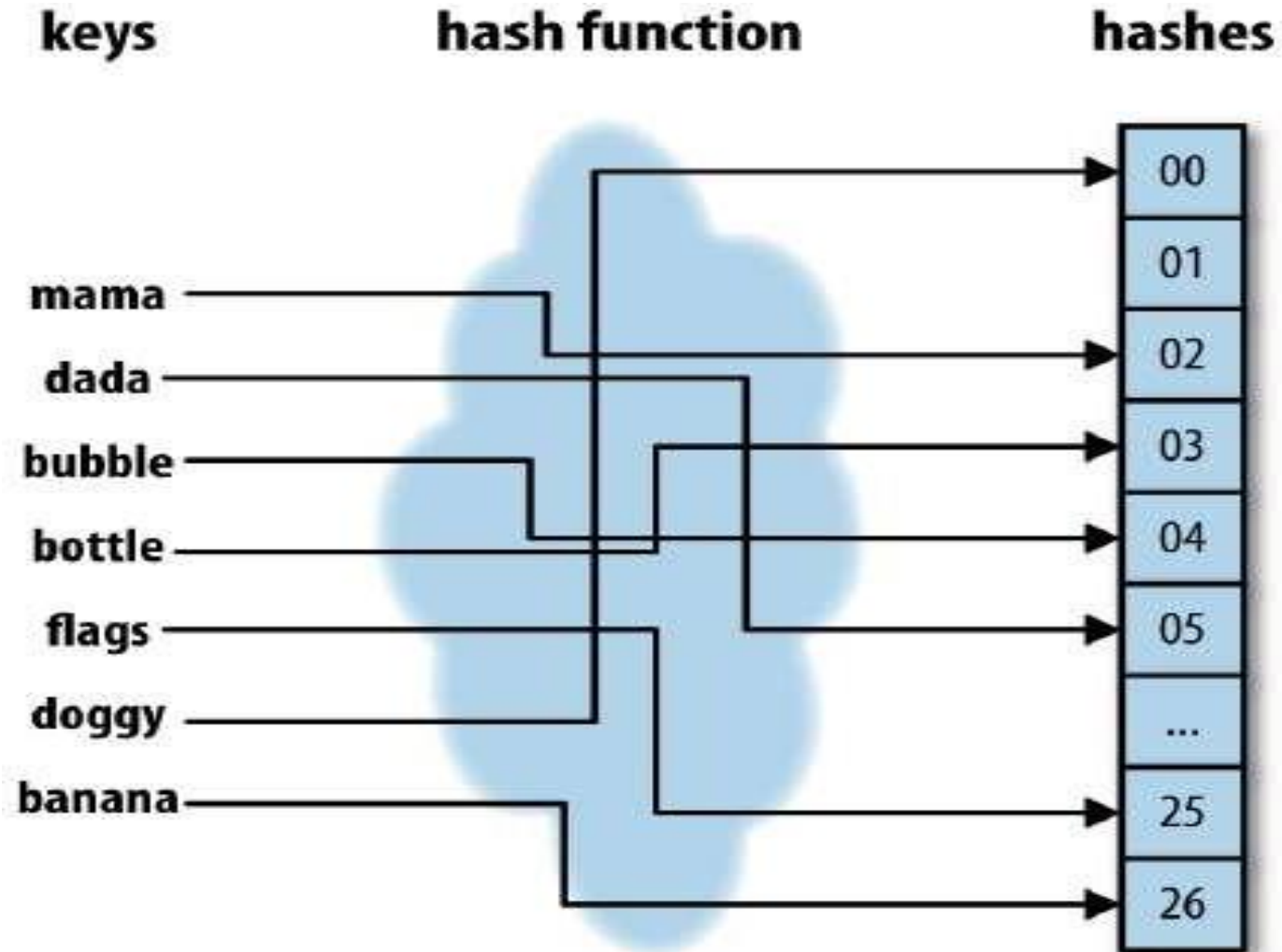


Fig. Hash functions mapping

Feature Hashing

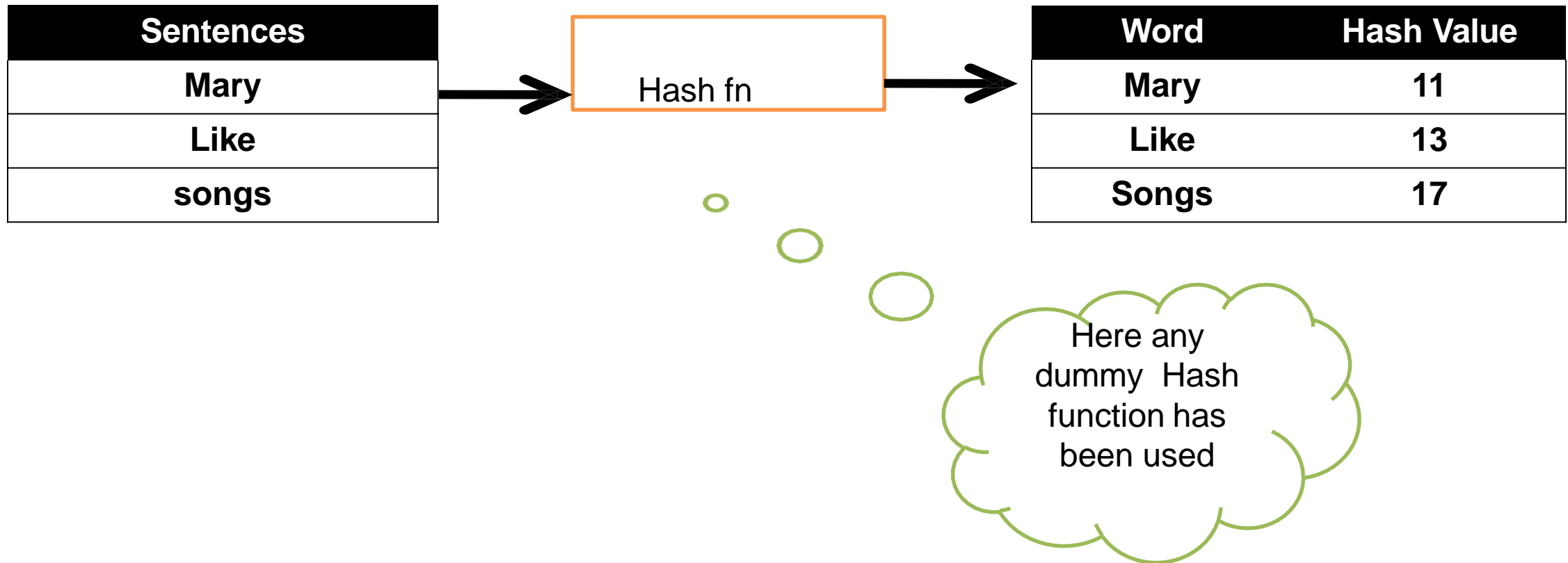
```
def hash_features(word_list, m):  
    for word in word_list: index =  
        hash_fn(word) % m output[index]  
        += 1
```

```
return output
```

- Here **m = m-dimensional vector**
- **hash_fn** is any hashing function

Feature Hashing

- EG:



Feature Hashing

- EG: let $m = 4$ i.e. four dimensional vector

Word	Hash Value(h)	Index (h %m)
Mary	11	3
Like	13	1
Songs	10	2

- Feature Vector

Sentences	Feature 1 (Remainder -0)	Feature 2 (Remainder -1)	Feature 3 (Remainder -2)	Feature 4 (Remainder -3)
Mary	0	0	0	1
Like	0	1	0	0
songs	0	0	1	0

Feature Hashing

- Advantages:
 - Significant advantage can be gained in **memory usage**
 - Feature hashing is well-suited to online learning scenarios, systems with **limited resources, or when speed and simplicity are important.**
- Disadvantages:
 - The ability to perform **the inverse mapping from feature indices back to feature values is lost**

Distance Measurement

- In NLP, we also want to find the similarity among sentence or document. Text is not like number and coordination that we cannot compare the different between “Apple” and “Orange” but similarity score can be calculated.
- **Why?**
 - Since we cannot simply subtract between “Apple is fruit” and “Orange is fruit” so that we have to find a way to convert text to numeric in order to calculate it. Having the score, we can understand how similar among two objects.
- **When?**
 - Compare whether 2 article are describing same news
 - Identifying similar documents
 - Classifying the category by giving product description

Distance Measurement

- How?
 - Euclidean Distance
 - Cosine Distance
 - Jaccard Similarity
- Before any distance measurement, text have to be tokenized.
- **Euclidean Distance**
- Comparing the shortest distance among two objects. It uses Pythagorean Theorem.
- Score means the distance between two objects. If it is 0, it means that both objects are identical. The following example shows score when comparing the first sentence.

Distance Measurement

- **Euclidean Distance**

Euclidean

$$\sqrt{\sum_{i=1}^k (x_i - y_i)^2}$$

```
Master Sentence: Elon Musk's Boring Co to build high-speed airport  
link in Chicago
```

```
-----
```

```
Score: 0.00, Comparing Sentence: Elon Musk's Boring Co to build high-  
speed airport link in Chicago
```

```
-----
```

```
Score: 1.73, Comparing Sentence: Elon Musk's Boring Company to build  
high-speed Chicago airport link
```

```
-----
```

```
Score: 4.36, Comparing Sentence: Elon Musk's Boring Company approved  
to build high-speed transit between downtown Chicago and O'Hare  
Airport
```

```
-----
```

```
Score: 4.24, Comparing Sentence: Both apple and orange are fruit
```

Distance Measurement

- **Cosine Similarity**

- Determine the angle between two objects is the calculation method to find similarity. The range of score is 0 to 1. If score is 1, it means that they are same in orientation (not magnitude). The following example shows score when comparing the first sentence.
- The measure computes the cosine of the angle between vectors ***x and y.***
 - **A cosine value of 0 means that the two vectors are at 90 degrees to each other (orthogonal) and have no match.**
 - **The closer the cosine value to 1, the smaller the angle and the greater the match between vectors.**

Distance Measures

- **Cosine Similarity**

- Let *x and y be two vectors for comparison*

$$\text{sim}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}|| ||\mathbf{y}||},$$

- **$||\mathbf{x}||$** is Euclidean norm of vector $\mathbf{x}=(x_1, x_2, \dots, x_p)$

- **Can be Calculated using:**

$$\sqrt{x_1^2 + x_2^2 + \dots + x_p^2}.$$

Distance Measures

- **Cosine Similarity**

Document Vector or Term-Frequency Vector

Document	team	coach	hockey	baseball	soccer	penalty	score	win	loss	season
<i>Document1</i>	5	0	3	0	2	0	0	2	0	0
<i>Document2</i>	3	0	2	0	1	1	0	1	0	1
<i>Document3</i>	0	7	0	2	1	0	0	3	0	0
<i>Document4</i>	0	1	0	0	1	2	2	0	3	0

Distance Measures

- **Cosine similarity between two term-frequency vectors.**
- first two term-frequency vectors in Table. That is, $\mathbf{x} = (5, 0, 3, 0, 2, 0, 0, 2, 0, 0)$ and $\mathbf{y} = (3, 0, 2, 0, 1, 1, 0, 1, 0, 1)$.

$$\mathbf{x}^t \cdot \mathbf{y} = 5 \times 3 + 0 \times 0 + 3 \times 2 + 0 \times 0 + 2 \times 1 + 0 \times 1 + 0 \times 0 + 2 \times 1 + 0 \times 0 + 0 \times 1 = 25$$

$$||\mathbf{x}|| = \sqrt{5^2 + 0^2 + 3^2 + 0^2 + 2^2 + 0^2 + 0^2 + 2^2 + 0^2 + 0^2} = 6.48$$

$$||\mathbf{y}|| = \sqrt{3^2 + 0^2 + 2^2 + 0^2 + 1^2 + 1^2 + 0^2 + 1^2 + 0^2 + 1^2} = 4.12$$

$$\text{sim}(\mathbf{x}, \mathbf{y}) = 0.94$$

- if we were using the cosine similarity measure to compare these documents, they would be considered quite similar

Distance Measures

- Cosine Similarity Applications:

Similarity between 2 Docs:

Sentence 1: AI is our friend and it has been friendly

Sentence 2: AI and humans have always been friendly

Soln:

- **lemmatization** to reduce words to the same root word.
- “friend” and “friendly” will both become “friend”, “has” and “have” will both become “has”.

Term Frequencies:										
Sentence	AI	IS	FRIEND	HUMAN	ALWAYS	AND	BEEN	OUR	IT	HAS
1	1	1	2	0	0	1	1	1	1	1
2	1	0	1	1	1	1	1	0	0	1

Distance Measures

- Cosine Similarity Applications:

Term Frequencies:											
Sentence	AI	IS	FRIEND	HUMAN	ALWAYS	AND	BEEN	OUR	IT	HAS	
1	1	1	2	0	0	1	1	1	1	1	
2	1	0	1	1	1	1	1	0	0	1	

– Cosine: $(1+2+1+1+1) / \sqrt{11} * \sqrt{7} = 0.684$

Distance Measures

- Cosine Similarity Applications:

Term Frequencies:											
Sentence	AI	IS	FRIEND	HUMAN	ALWAYS	AND	BEEN	OUR	IT	HAS	
1	2	1	2	0	0	1	1	1	1	1	
2	1	0	1	1	1	1	1	0	0	1	

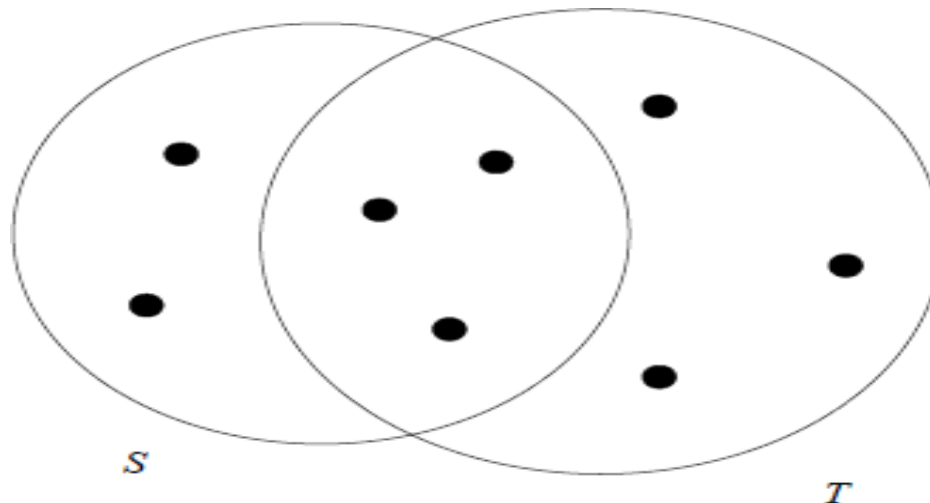
- If AI appears 2 times in sen:1
- $(2+2+1+1+1)/\sqrt{14}*\sqrt{7} = 0.7071$

Distance Measurement

- **Jaccard Similarity**
- The measurement is refer to number of common words over all words. More commons mean both objects should be similarity.
- Jaccard Similarity = $(\text{Intersection of A and B}) / (\text{Union of A and B})$
- The range is 0 to 1. If score is 1, it means that they are identical. There is no any common word between the first sentence and the last sentence so the score is 0. The following example shows score when comparing the first sentence.

Distance Measures

- **Jaccard similarity :**
 - The Jaccard similarity of sets S and T
 - is $|S \cap T| / |S \cup T|$,
 - that is, the ratio of the size of the intersection of S and T to the size of their union.



$$\text{sim}(s,t) = 3/8$$

Distance Measures

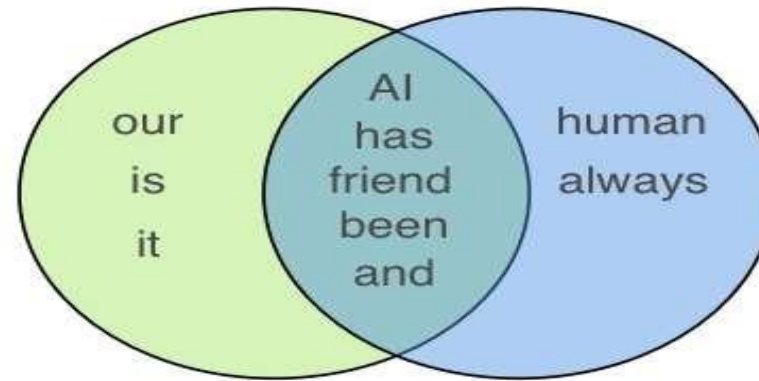
- Jaccard Similarity Applications:

Similarity between 2 Docs:

Sentence 1: AI is our friend and it has been friendly

Sentence 2: AI and humans have always been friendly

Soln:



- **lemmatization** to reduce words to the same root word.
- “friend” and “friendly” will both become “friend”, “has” and “have” will both become “has”.
- Jaccard similarity of $5/(5+3+2) = 0.5$

Distance Measures

- Jaccard Similarity Applications:
 - **Recommend Movies /Recommend Products**
 - If a customer rated a movie highly, put “liked” for that movie in the customer’s set.
 - If they gave a low rating to a movie, put “hated” for that movie in their set. Then, we can look for high Jaccard similarity among these sets.
 - Eg: If ratings are **1-to-5-stars**, ***put a movie in a customer’s set n times if they rated the movie n -stars.***
- Then, use Jaccard similarity for bags when measuring the similarity of customers.
- The Jaccard **similarity for bags B and C is defined by counting an element n times in the intersection if n is the minimum of the number of times the element appears in B and C.**
 - In the union, we count the element the sum of the number of times it appears in B and in C.
 - The bag-similarity of bags **$\{a, a, a, b\}$ and $\{a, a, b, b, c\}$ is $1/3$.**
 - The intersection counts a twice and b once, so its size is 3. The size of the union of two bags is always the sum of the sizes of the two bags, or 9 in this case.

Distance Measures

- Jaccard vs. Cosine Similarity Applications:
 - Jaccard similarity takes only **unique set of words** for each sentence / document while cosine similarity takes **total length of the vectors**.

Distance Measures

- **Hamming Distance**

- Hamming distance calculates the distance between two binary vectors, also referred to as binary strings or bit strings for short.
- You are most likely going to encounter bit strings when you one-hot encode categorical columns of data.
- For example, if a column had the categories 'red,' 'green,' and 'blue,' you might one hot encode each example as a bit string with one bit for each column.
- red = [1, 0, 0]
- green = [0, 1, 0]
- blue = [0, 0, 1]

Distance Measures

- **Hamming Distance**

- The distance between red and green could be calculated as the sum or the average number of bit differences between the two bitstrings. This is the Hamming distance.
- For a one-hot encoded string, it might make more sense to summarize to the sum of the bit differences between the strings, which will always be a 0 or 1.
- Hamming Distance = sum for i to N $\text{abs}(v1[i] - v2[i])$
- For bitstrings that may have many 1 bits, it is more common to calculate the average number of bit differences to give a hamming distance score between 0 (identical) and 1 (all different).
- Hamming Distance = $(\text{sum for } i \text{ to } N \text{ } \text{abs}(v1[i] - v2[i])) / N$

Distance Measures

- **Hamming Distance**

- Given two vectors to be the number of components in **which they differ**
- The Hamming distance allows only substitution, hence, it only applies to strings of the same length.
- Most commonly, Hamming distance is used when the vectors are Boolean; they consist of 0's and 1's only.
- However, in principle, the vectors can have components from any set.
- Eg:
 - The Hamming distance between the vectors 10101 and 11110 is 3.
 - That is, these vectors differ in the second, fourth, and fifth components, while they agree in the first and third components.

Distance Measures

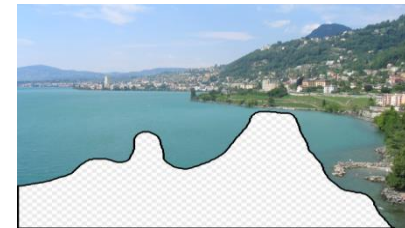
- **Hamming Distance**

- Applications:

- Information theory, coding theory, and cryptography.
 - *Error detecting and error correcting codes*
 - telecommunication
 - to count the number of flipped bits in a fixed-length binary word as an estimate of error, and therefore is sometimes called the **signal distance**

A Common Metaphor

- Many problems can be expressed as finding “similar” sets:
 - Find near-neighbors in high-dimensional space
- **Examples:**
 - Pages with similar words
 - For duplicate detection, classification by topic
 - Customers who purchased similar products
 - Products with similar customer sets
 - Images with similar features
 - Users who visited similar websites



Locality Sensitive Hashing (LSH)

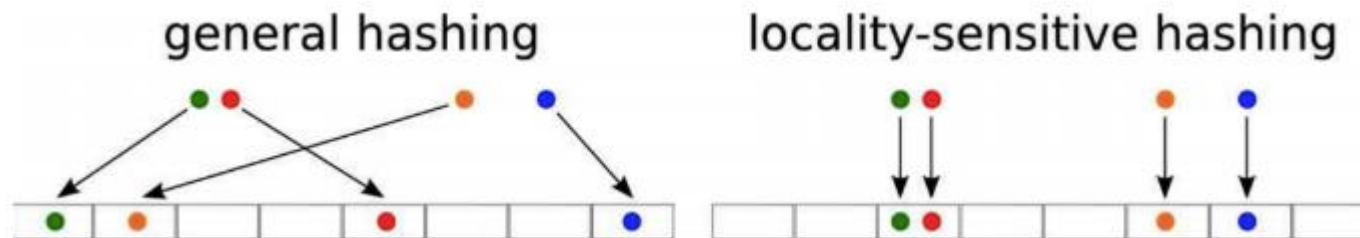
- The task of finding nearest neighbours is very common.
- You can think of applications **like finding duplicate or similar documents, audio/video search**. Although using brute force to check for all possible combinations will give you the exact nearest neighbour but it's not scalable at all.
- Approximate algorithms to accomplish this task has been an area of active research. Although these algorithms don't guarantee to give you the exact answer, more often than not they'll provide a good approximation. These algorithms are faster and scalable.
- Locality sensitive hashing (LSH) is a procedure for finding similar pairs in a large dataset. For a dataset of size N , the brute force method of comparing every possible pair would take $N!/(2!(N-2)!) \sim N^2/2 = O(N^2)$ time. The LSH method aims to cut this down to $O(N)$ time.

Locality Sensitive Hashing (LSH)

- Locality sensitive hashing (LSH) is one such algorithm. LSH has many applications, including:
 - Near-duplicate detection: LSH is commonly used to deduplicate large quantities of documents, webpages, and other files.
 - Genome-wide association study: Biologists often use LSH to identify similar gene expressions in genome databases.
 - Large-scale image search: Google used LSH along with PageRank to build their image search technology VisualRank.
 - Audio/video fingerprinting: In multimedia technologies, LSH is widely used as a fingerprinting technique A/V data.

Locality Sensitive Hashing (LSH)

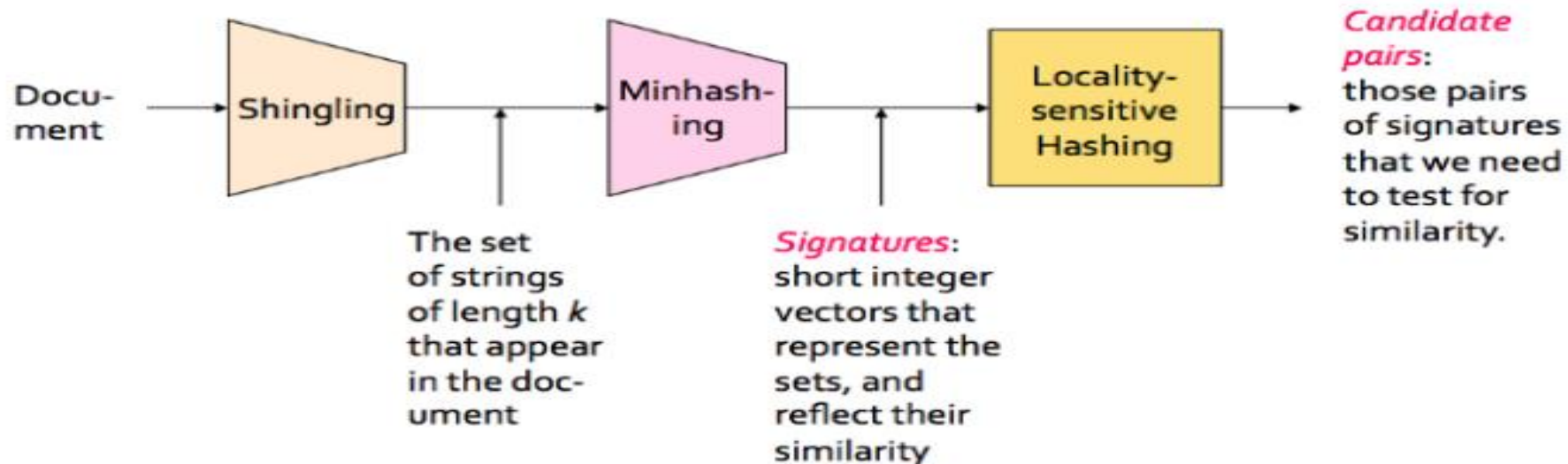
- LSH refers to a family of functions (known as LSH families) to hash data points into buckets so that **data points near each other are located in the same buckets with high probability**, while data points far from each other are likely to be in different buckets. This makes it easier to identify observations with various degrees of similarity.

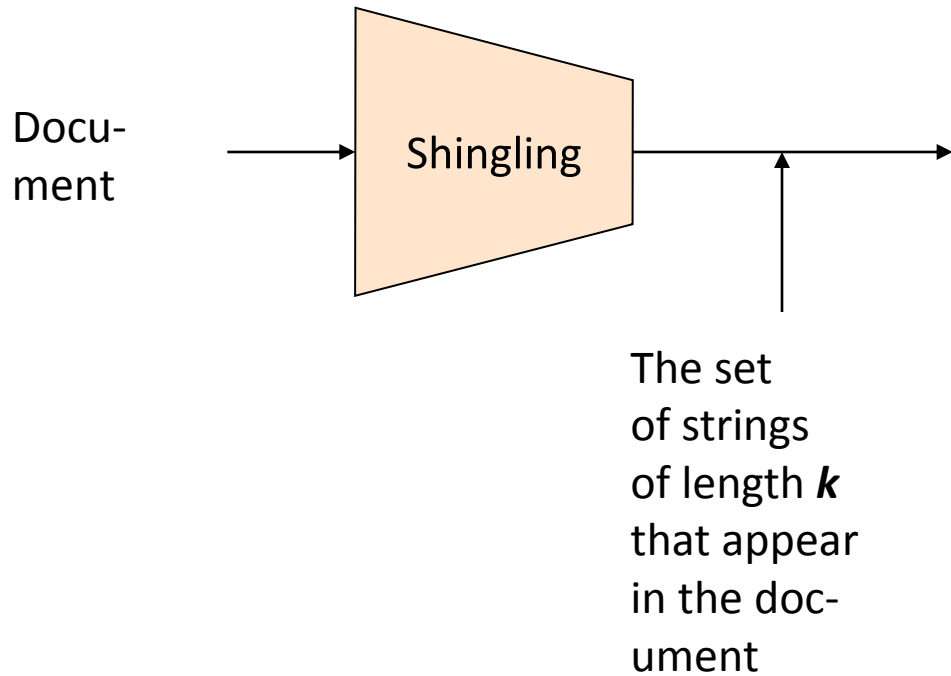


- Finding similar documents
 - how we can leverage LSH in solving an actual problem. The problem that we're trying to solve: **Goal:** You have been given a large collections of documents. You want to find “near duplicate” pairs.

Locality Sensitive Hashing (LSH)

- In the context of this problem, we can break down the LSH algorithm into 3 broad steps:
- **Shingling** - Convert documents to sets
- **Min hashing** – Convert large sets to short signatures while preserving similarity. Signature estimate the Jaccard similarity of sets.
- **Locality-sensitive hashing** - Focus on pairs of signatures likely to be from similar documents





Shingling

Step 1: *Shingling*: Convert documents to sets

In order to quantify a document, we need to vectorize it. One method for doing this, is to enumerate all of its **k-shingles**. A **k-shingle** is just any k -consecutive characters occurring in the document.

Locality Sensitive Hashing (LSH)

- **Shingling**

- In this step, we convert each document into a set of characters of length k (also known as k -shingles or k -grams). The key idea is to represent each document in our collection as a set of k -shingles.
- For ex: One of your document (D): “Nadal”. Now if we’re interested in 2-shingles, then our set: {Na, ad, da, al}. Similarly set of 3-shingles: {Nad, ada, dal}.
- Similar documents are more likely to share more shingles
- k value of 8–10 is generally used in practice. A small value will result in many shingles which are present in most of the documents (bad for differentiating documents)

Finding Similar Items

- **Simple approaches:**
 - Document = set of words appearing in document
 - Document = set of “important” words
- A k-shingle (or k-gram) for a document is a sequence of k tokens that appears in the doc
 - Tokens can be characters, words or something else, depending on the application
 - Assume tokens = characters for examples
 - **Example: $k=2$; document $D_1 = \text{abcab}$**
Set of 2-shingles: $S(D_1) = \{\text{ab}, \text{bc}, \text{ca}\}$

Finding Similar Items

- **Finding Similar Documents (3 – Steps)**
- **Step 1: Shingling: Convert documents to sets**
- "a rose is a rose is a rose"
 - The set of all contiguous sequences of 4 tokens (Thus $4=n$, thus 4-grams) is
 - { (a,rose,is,a), (rose,is,a,rose), (is,a,rose,is),
(a,rose,is,a), (rose,is,a,rose) }
 - reduced to { (a,rose,is,a), (rose,is,a,rose),
(is,a,rose,is) }.

Finding Similar Items

- Choosing the Shingle Size
 - k should be picked large enough that the probability of any given shingle appearing in any given document is low.
 - Effect of k : $k = 1$, most Web pages will have most of the common characters and few other characters, so almost all Web pages will have high similarity.
 - Short (E-mail): $k = 5$
 - large documents, such as research articles: choice $k = 9$

Choosing k

- Let $k = 5$
- Suppose that only letters and general white space characters appear in emails.
- If so, there would be $27^5 = 14,348,907$ possible shingles.
- Since the typical email is much smaller than 14 million characters long, we would expect $k = 5$ to work well and it does.
- A rule of thumb is to imagine there are 20 characters and estimate the number of shingles as 20^k . For large documents (research articles), a choice of $k = 9$ is considered to be safe

Finding Similar Items

- Matrix Representation of Sets
 - To construct small signatures from large sets,
 - First : visualize a collection of sets as their characteristic matrix
 - The **columns of the matrix** correspond to the sets,
 - **The rows correspond to elements** of the universal set from which elements of the sets are drawn.

Finding Similar Items

- Matrix Representation of Sets

– Here, $S_1 = \{a, d\}$, $S_2 = \{c\}$, $S_3 = \{b, d, e\}$, and $S_4 = \{a, c, d\}$.

rows correspond to
elements of the
universal set

columns of the matrix
correspond to the sets
..like $S_1 = \{a, b\}$

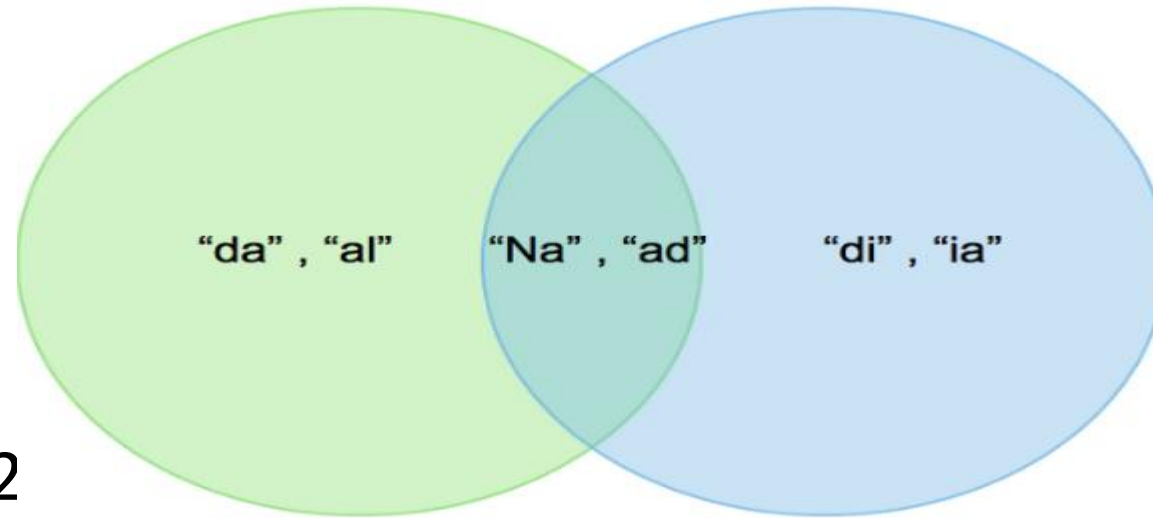
<i>Element</i>	S_1	S_2	S_3	S_4
<i>a</i>	1	0	0	1
<i>b</i>	0	0	1	0
<i>c</i>	0	1	0	1
<i>d</i>	1	0	1	1
<i>e</i>	0	0	1	0

Fig 1: Matrix Representation of sets

Locality Sensitive Hashing (LSH)

- Jaccard Index
 - The **Jaccard similarity** of two **sets** is the size of their intersection divided by the size of their union:
$$\text{sim}(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$
 - **Jaccard distance:** $d(C_1, C_2) = 1 - |C_1 \cap C_2| / |C_1 \cup C_2|$
 - Suppose A: “Nadal” and B: “Nadia”, then 2-shingles representation will be: A: {Na, ad, da, al} and B: {Na, ad, di, ia}.

Locality Sensitive Hashing (LSH)



- Jaccard Index = 2
- More number of common shingles will result in bigger Jaccard Index and hence more likely that the documents are similar.
- Let's discuss 2 big issues that we need to tackle:
 - Time complexity
 - Space complexity

Locality Sensitive Hashing (LSH)

- **Time complexity**

- Now you may be thinking that we can stop here. But if you think about the scalability, doing just this won't work. For a collection of n documents, you need to do $n*(n-1)/2$ comparison, basically **$O(n^2)$** . Imagine you have 1 million documents, then the number of comparison will be $5*10^{11}$.

- **Space complexity**

- The document matrix is a sparse matrix and storing it as it is will be a big memory overhead.
- One way to solve this is hashing.

Locality Sensitive Hashing (LSH)

- **Hashing**
- The idea of hashing is to convert each document to a small signature using a hashing function H . Suppose a document in our corpus is denoted by d . Then:
- $H(d)$ is the signature and it's small enough to fit in memory
- If $\text{similarity}(d_1, d_2)$ is high then $\text{Probability}(H(d_1) == H(d_2))$ is high
- If $\text{similarity}(d_1, d_2)$ is low then $\text{Probability}(H(d_1) == H(d_2))$ is low
- Choice of hashing function is tightly linked to the similarity metric we're using. For Jaccard similarity the appropriate hashing function is min-hashing.

Locality Sensitive Hashing (LSH)

- Computing Minhash values
 - Rows are permuted randomly
 - Minhasfunc $h(C)$ = the number of the first (n the permuted order) row in which column C has 1.
 - Independent hash functions to create signature of each column.

Minhashing

- Let the order of rows **beadc** be the random permutation for the matrix (fig 1)
- It defines a minhash function h that maps sets to rows.

<i>Element</i>	S_1	S_2	S_3	S_4
<i>b</i>	0	0	1	0
<i>e</i>	0	0	1	0
<i>a</i>	1	0	0	1
<i>d</i>	1	0	1	1
<i>c</i>	0	1	0	1

Minhashing

- To compute *the minhash value of set S_1 according to h* .
- Check for first '1' in the column:
 - The first column, which is the column for set S_1 , has 0 in row b,
 - So we proceed to row e, the second in the permuted order. There is again a 0 in the column for S_1 ,
 - So we proceed to row a, where we find a 1.
 - **$h(S_1) = a$.**
 - **(Obtained similarly) $h(S_2) = c$, $h(S_3) = b$, and $h(S_4) = a$**

<i>Element</i>	S_1	S_2	S_3	S_4
<i>b</i>	0	0	1	0
<i>e</i>	0	0	1	0
<i>a</i>	1	0	0	1
<i>d</i>	1	0	1	1
<i>c</i>	0	1	0	1

Minhashing

- Let **Random number n** (say, 100s or 1000s) be the
- Let minhash functions determined by these permutations h_1, h_2, \dots, h_n .
- From the column representing set S ,
 - – construct the minhash signature for S , the vector $[h_1(S), h_2(S), \dots, h_n(S)]$. List hash-values as columns

<i>Element</i>	S_1	S_2	S_3	S_4
<i>b</i>	0	0	1	0
<i>e</i>	0	0	1	0
<i>a</i>	1	0	0	1
<i>d</i>	1	0	1	1
<i>c</i>	0	1	0	1

a	c	b	a
---	---	---	---

Minhashing

- **Minhashing and Jaccard Similarity**
- The probability that the minhash function for a random permutation of rows produces the same value for two sets equals the jaccard similarity of those sets.
- X rows (1's in both cols)
- Y rows (1 in one of the columns, 0 in the other)
- Z rows (0 in both cols)

C_1	C_2		
0	1	*	
1	0	*	
1	1	*	*
0	0		
1	1	*	*
0	1	*	

$\text{Sim}(C_1, C_2) = \frac{2}{5} = 0.4$

Minhashing

- Min hashing (Example-2)
- **Step 1:** Random permutation (π) of *row index* of document shingle matrix.

Permutation π Input matrix (Shingles x Documents)

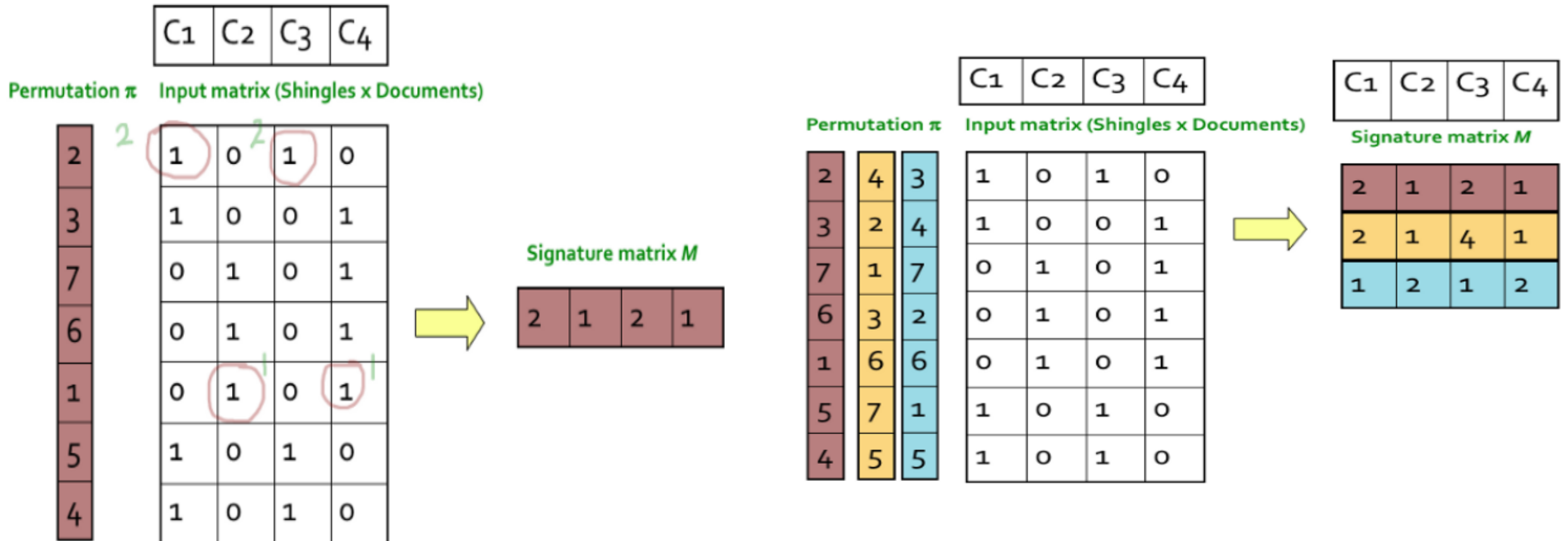
2	1	0	1	0
3	1	0	0	1
7	0	1	0	1
6	0	1	0	1
1	0	1	0	1
5	1	0	1	0
4	1	0	1	0

- **Step 2:** Hash function is the index of the first (in the permuted order) row in which column C has value 1. Do this several time (use different permutations) to create signature of a column.

$$h_{\pi}(C) = \min_{\pi} \pi(C)$$

Minhashing

- Min hashing



Minhashing

- Algorithm

- Let $SIG(i, c)$ be the element of the signature matrix for the i th hash function and column c . Initially, set $SIG(i, c)$ to ∞ for all i and c

For each row r do begin

 For each hash function h_i do:

 Compute $h_i(r)$;

 For each column c

 If c has 1 in row r

 For each hash function h_i do:

 If $h_i(r) < SIG(i, c)$ then

$SIG(i, c) = h_i(r)$

Minhashing

- Algorithm

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

$h_1(x)$ $h_2(x)$

Row	S_1	S_2	S_3	S_4	$x + 1 \bmod 5$	$3x + 1 \bmod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

$h_1(x)$ $h_2(x)$

Row	S_1	S_2	S_3	S_4	$x + 1 \bmod 5$	$3x + 1 \bmod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

$h_1(x)$ $h_2(x)$

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

$h_1(x)$ $h_2(x)$

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

Row	S_1	S_2	S_3	S_4	$x + 1 \bmod 5$	$3x + 1 \bmod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Row	S_1	S_2	S_3	S_4	$x + 1 \bmod 5$	$3x + 1 \bmod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Minhashing

- Algorithm $P[h_\pi(C_1) = h_\pi(C_2)] = \text{sim}(C_1, C_2)$

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

$$\text{sim}(S_1, S_2) = 1 \cdot 0$$

Row	S_1	S_2	S_3	S_4	$x+1 \bmod 5$	$3x+1 \bmod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

$$\text{Jaccard sim} = \frac{2}{3}$$

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

$$\text{sim}(S_1, S_2) = \frac{1}{2}$$

Row	S_1	S_2	S_3	S_4	$x+1 \bmod 5$	$3x+1 \bmod 5$
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

$$\text{sim}(S_1, S_3) = \frac{1}{4}$$

Minhashing

- Algorithm $P[h_\pi(C_1) = h_\pi(C_2)] = \text{sim}(C_1, C_2)$

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

$$\text{sim}(S_1, S_2) = 0$$

Row	S_1	S_2	S_3	S_4	$x + 1 \mod 5$	$3x + 1 \mod 5$
✓ 0	1	0	0	1	1	1
1	0	0	1	0	2	4
✗ 2	0	1	0	1	3	2
✗ 3	1	0	1	1	4	0
4	0	0	1	0	0	3

$$\text{sim}(S_1, S_2) = 0$$

Min-Hashing Example

Permutation π

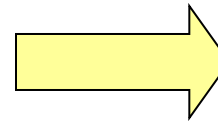
2	4	3
3	2	4
7	1	7
6	3	2
1	6	6
5	7	1
4	5	5

Input matrix (Shingles x Documents)

1	0	1	0
1	0	0	1
0	1	0	1
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0

Signature matrix M

2	1	2	1
2	1	4	1
1	2	1	2

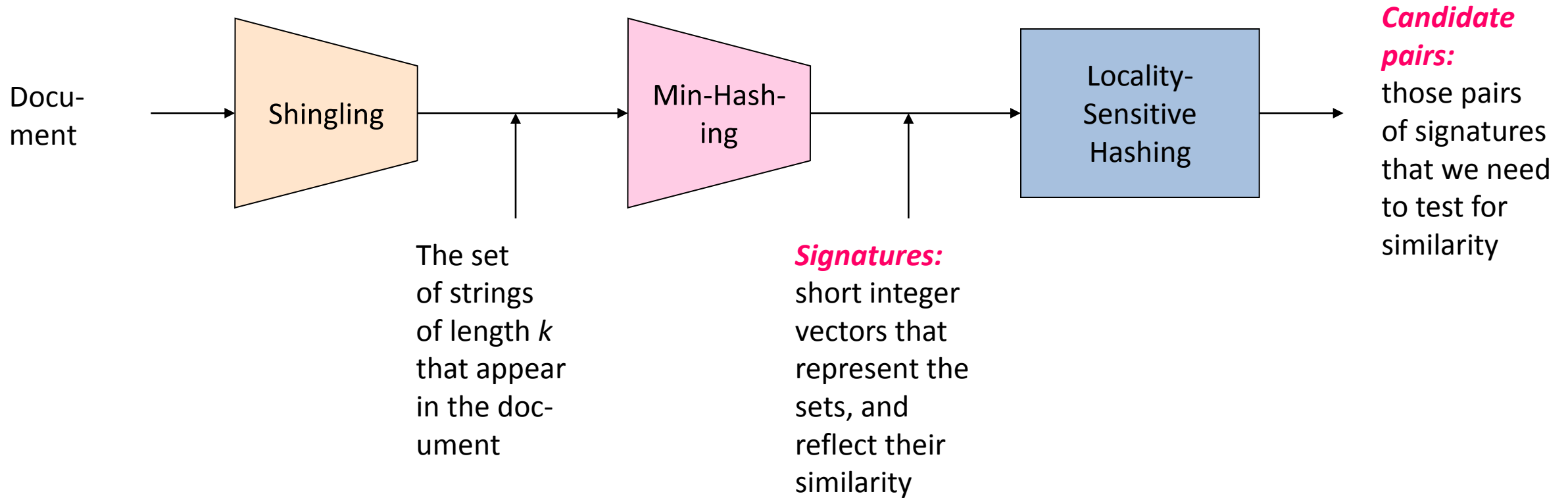


Similarities:

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

Locality Sensitive Hashing (LSH)

- **Locality-sensitive hashing- The band structure procedure.**
- Goal: Find documents with Jaccard similarity of at least t
- The general idea of LSH is to find an algorithm such that if we input signatures of 2 documents, it tells us that those 2 documents form a candidate pair or not i.e. their similarity is greater than a threshold t . Remember that we are taking similarity of signatures as a proxy for Jaccard similarity between the original documents.
- The intuition is this: instead of comparing every pair of elements, what if we could just hash them into buckets, and hopefully elements that map to the same buckets will be the right level of “close” to each other. The level of “close” is precisely the desired similarity threshold



Locality Sensitive Hashing

Step 3: *Locality-Sensitive Hashing:*

Focus on pairs of signatures likely to be from similar documents

LSH: First Cut

2	1	4	1
1	2	1	2
2	1	2	1

- Goal: Find documents with Jaccard similarity at least s (for some similarity threshold, e.g., $s=0.8$)
- LSH – General idea: Use a function $f(x,y)$ that tells whether x and y is a *candidate pair*: a pair of elements whose similarity must be evaluated
- For Min-Hash matrices:
 - Hash columns of signature matrix M to many buckets
 - Each pair of documents that hashes into the same bucket is a *candidate pair*

Candidates from Min-Hash

2	1	4	1
1	2	1	2
2	1	2	1

- Pick a similarity threshold s ($0 < s < 1$)
- Columns \mathbf{x} and \mathbf{y} of \mathbf{M} are a **candidate pair** if their signatures agree on at least fraction s of their rows:
 $\mathbf{M}(i, \mathbf{x}) = \mathbf{M}(i, \mathbf{y})$ for at least frac. s values of i
 - We expect documents \mathbf{x} and \mathbf{y} to have the same (Jaccard) similarity as their signatures

LSH for Min-Hash

2	1	4	1
1	2	1	2
2	1	2	1

- **Big idea:** Hash columns of signature matrix M several times
- Arrange that (only) **similar columns** are likely to **hash to the same bucket**, with high probability
- **Candidate pairs** are those that hash to the same bucket

LSH for Min-Hash

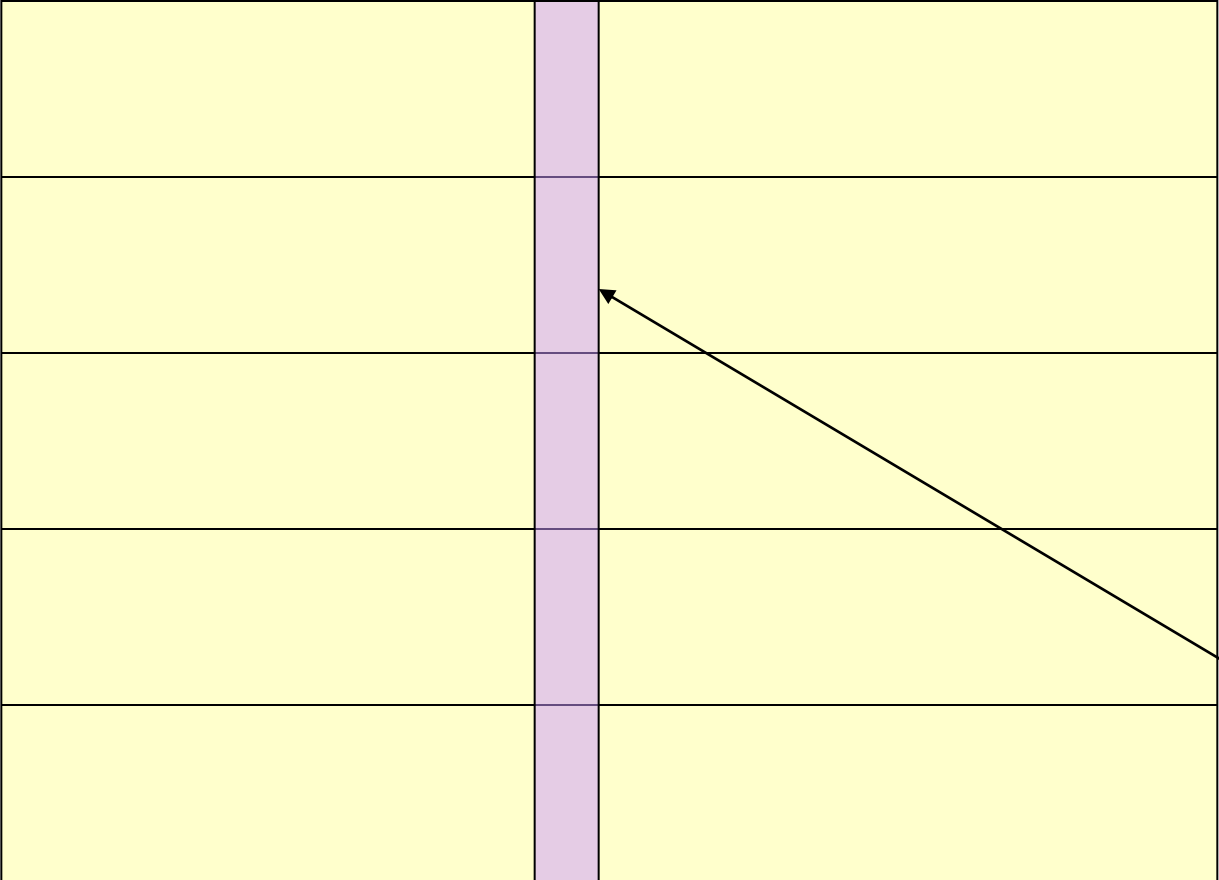
2	1	4	1
1	2	1	2
2	1	2	1

- **Band Method**
- n is the number of hash functions, i.e. the dimension K of the signature matrix. Also define:
 - b as the number of bands
 - r as the number of rows per band
 - and it's apparent that $n=b*r$.
- Two rows are considered **candidate pairs** (meaning they have the possibility to have a similarity score above the desired threshold), if the two rows have **at least one** band in which all the rows are identical
- Hopefully the intuition behind b and r should make sense here; **increasing b gives rows more chances to match**, so it let's in candidate pairs **with lower similarity scores**. **Increasing r makes the match criteria stricter**, restricting to **higher similarity scores**. r and b do opposite things

Partition M into b Bands

2	1	4	1
1	2	1	2
2	1	2	1

b bands



r rows
per band

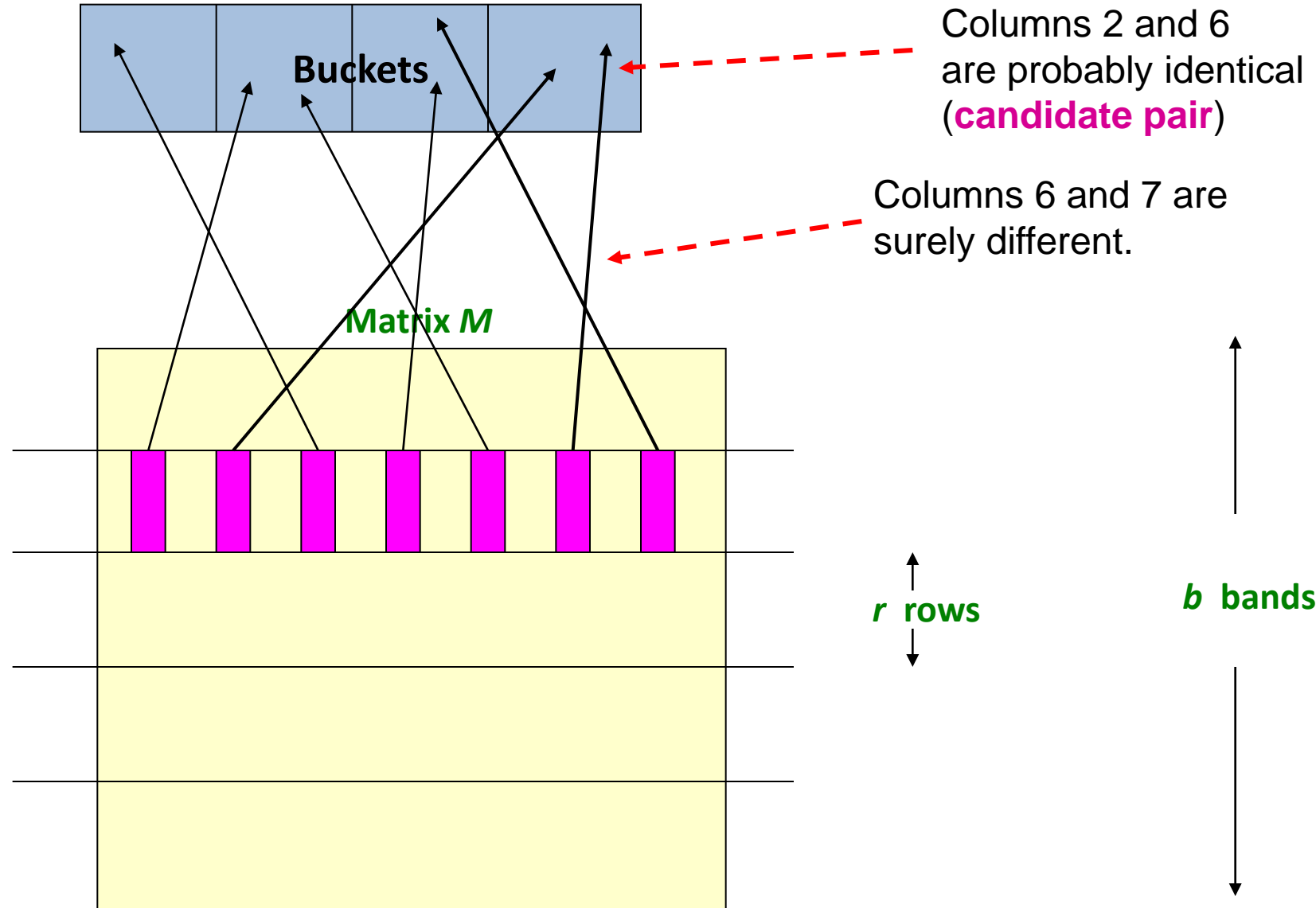
One
signature

Signature matrix M

Partition M into Bands

- Divide matrix M into b bands of r rows
- For each band, hash its portion of each column to a hash table with k buckets
 - Make k as large as possible
- **Candidate** column pairs are those that hash to the same bucket for ≥ 1 band
- Tune b and r to catch most similar pairs, but few non-similar pairs

Hashing Bands



Example of Bands

2	1	4	1
1	2	1	2
2	1	2	1

Assume the following case:

- Suppose 100,000 columns of \mathbf{M} (100k docs)
- Signatures of 100 integers (rows)
- Therefore, signatures take 40Mb
- Choose $b = 20$ bands of $r = 5$ integers/band
- **Goal:** Find pairs of documents that are at least $s = 0.8$ similar

C_1, C_2 are 80% Similar

2	1	4	1
1	2	1	2
2	1	2	1

- Find pairs of $\geq s=0.8$ similarity, set $b=20$, $r=5$
- Assume: $\text{sim}(C_1, C_2) = 0.8$
 - Since $\text{sim}(C_1, C_2) \geq s$, we want C_1, C_2 to be a **candidate pair**: We want them to hash to at **least 1 common bucket** (at least one band is identical)
- Probability C_1, C_2 identical in one particular band: $(0.8)^5 = 0.328$
- Probability C_1, C_2 are **not** similar in all of the 20 bands: $(1-0.328)^{20} = 0.00035$
 - i.e., about 1/3000th of the 80%-similar column pairs are **false negatives** (we miss them)
 - We would find **99.965% pairs of truly similar documents**

C_1, C_2 are 30% Similar

2	1	4	1
1	2	1	2
2	1	2	1

- **Find pairs of $\geq s=0.8$ similarity, set $b=20, r=5$**
- **Assume:** $\text{sim}(C_1, C_2) = 0.3$
 - Since $\text{sim}(C_1, C_2) < s$ we want C_1, C_2 to hash to **NO common buckets** (all bands should be different)
- **Probability C_1, C_2 identical in one particular band:** $(0.3)^5 = 0.00243$
- **Probability C_1, C_2 identical in at least 1 of 20 bands:** $1 - (1 - 0.00243)^{20} = 0.0474$
 - In other words, approximately 4.74% pairs of docs with similarity 0.3% end up becoming **candidate pairs**
 - They are **false positives** since we will have to examine them (they are candidate pairs) but then it will turn out their similarity is below threshold s

LSH Involves a Tradeoff

2	1	4	1
1	2	1	2
2	1	2	1

- **Pick:**
 - The number of Min-Hashes (rows of \mathbf{M})
 - The number of bands \mathbf{b} , and
 - The number of rows \mathbf{r} per bandto balance false positives/negatives
- **Example:** If we had only 15 bands of 5 rows, the number of false positives would go down, but the number of false negatives would go up

b bands, r rows/band

- Columns C_1 and C_2 have similarity t
- Pick any band (r rows)
 - Prob. that all rows in band equal = t^r
 - Prob. that some row in band unequal = $1 - t^r$
- Prob. that no band identical = $(1 - t^r)^b$
- Prob. that at least 1 band identical = $1 - (1 - t^r)^b$

Example: $b = 20$; $r = 5$

- Similarity threshold s
- Prob. that at least 1 band is identical:

s	$1-(1-s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

LSH Summary

- Tune M , b , r to get almost all pairs with similar signatures, but eliminate most pairs that do not have similar signatures
- Check in main memory that **candidate pairs** really do have **similar signatures**
- **Optional:** In another pass through data, check that the remaining candidate pairs really represent similar documents

Summary: 3 Steps

- **Shingling:** Convert documents to sets
 - We used hashing to assign each shingle an ID
- **Min-Hashing:** Convert large sets to short signatures, while preserving similarity
 - We used **similarity preserving hashing** to generate signatures with property $\Pr[h_{\pi}(C_1) = h_{\pi}(C_2)] = \text{sim}(C_1, C_2)$
 - We used hashing to get around generating random permutations
- **Locality-Sensitive Hashing:** Focus on pairs of signatures likely to be from similar documents
 - We used hashing to find **candidate pairs** of similarity $\geq s$

Thank you.