REVA UNIVERSITY
Bengaluru, India

nirf

Natural Language Processing

Unit 3

Text Summarization , **Text Similarity and Clustering** (Chapter 5and 6 from textbook-2)

UNIT-II

Text Summarization: Text Summarization and Information Extraction, Feature Matrix, Single Value Decomposition, Keyphrase Extraction, Topic Modelling, Automated Document Summarization**. Text Similarity and Clustering:** Information Retrieval, Feature Engineering, Text Similarity, Analyzing Term Similarity, Analyzing Document Similarity, Document Clustering, Clustering Greatest Movies of All Time

Textbook-2 Link :
https://revaedu-my.sharepoint.com/:b:/g/personal/shilpa_mathpati_reva_edu_in/Ed0Ohc19rdlKuyC-9pLm0d0BfIipYTU9xEW6NcfCngI05Q?e=aOob0T

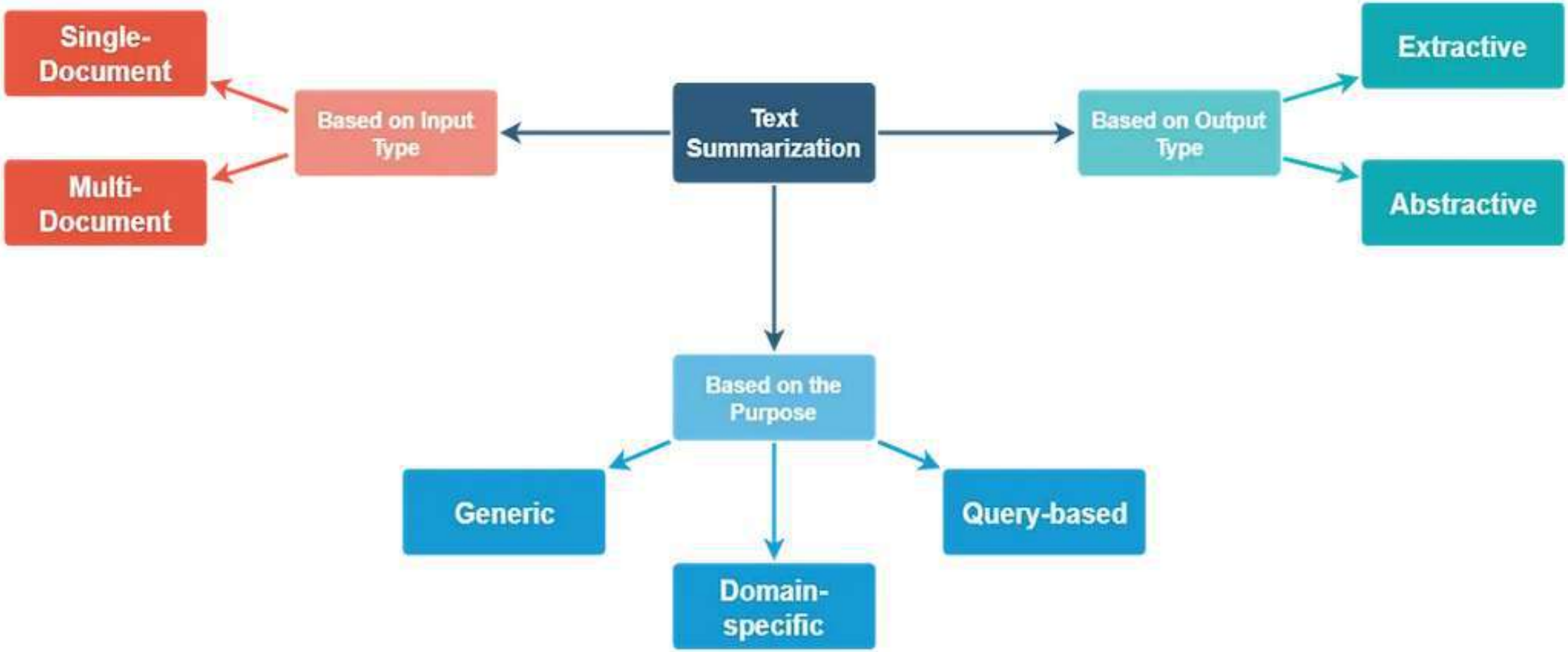
Text Summarization:Understanding the Need and Techniques of Summarizing Text

- Text analytics and NLP help process large amounts of textual data.
  Machine learning (ML) techniques improve classification and feature extraction.
- The need for text summarization arises due to **information overload**.
- Helps businesses and consumers by condensing lengthy documents while retaining core information.
- **The Evolution of Information Overload**
  - **Ancient times:** Scholars overwhelmed by excessive book production.
  - **1440 AD:** Gutenberg's printing press led to mass production of books.
  - **20th century:** Digital advancements increased data generation.
  - **Internet Era:** Social media, emails, and digital content led to exponential information growth.

Why is Text Summarization Important?

- Humans have **short attention spans** and limited cognitive processing.
- Large text documents can be difficult to consume and analyze.
- Summarization helps in:
  - Extracting **key phrases**.
  - Identifying **diverse topics**.
  - Generating concise **summaries** that retain core themes.
- **Types of Text Summarization**
  - **Key Phrase Extraction:** Identifies the most influential words/phrases.
  - **Topic Modeling:** Clusters different concepts in a document.
  - **Automated Text Summarization:** Uses AI/ML to create shorter versions of text while keeping essential information

Type of summarization:



Text Summarization and Information Extraction

What is Information Overload?

Definition: Excess data that exceeds human processing capabilities.

Historical context: From ancient books to the digital age.

Examples: Social media, emails, news articles.

Why Do We Need Text Summarization?

Challenges: Cognitive limits, time constraints, decision-making.

Business applications: Executives need quick insights.

Key Concepts

Text summarization: Reducing large documents to concise summaries.

Information extraction: Identifying key themes and concepts.

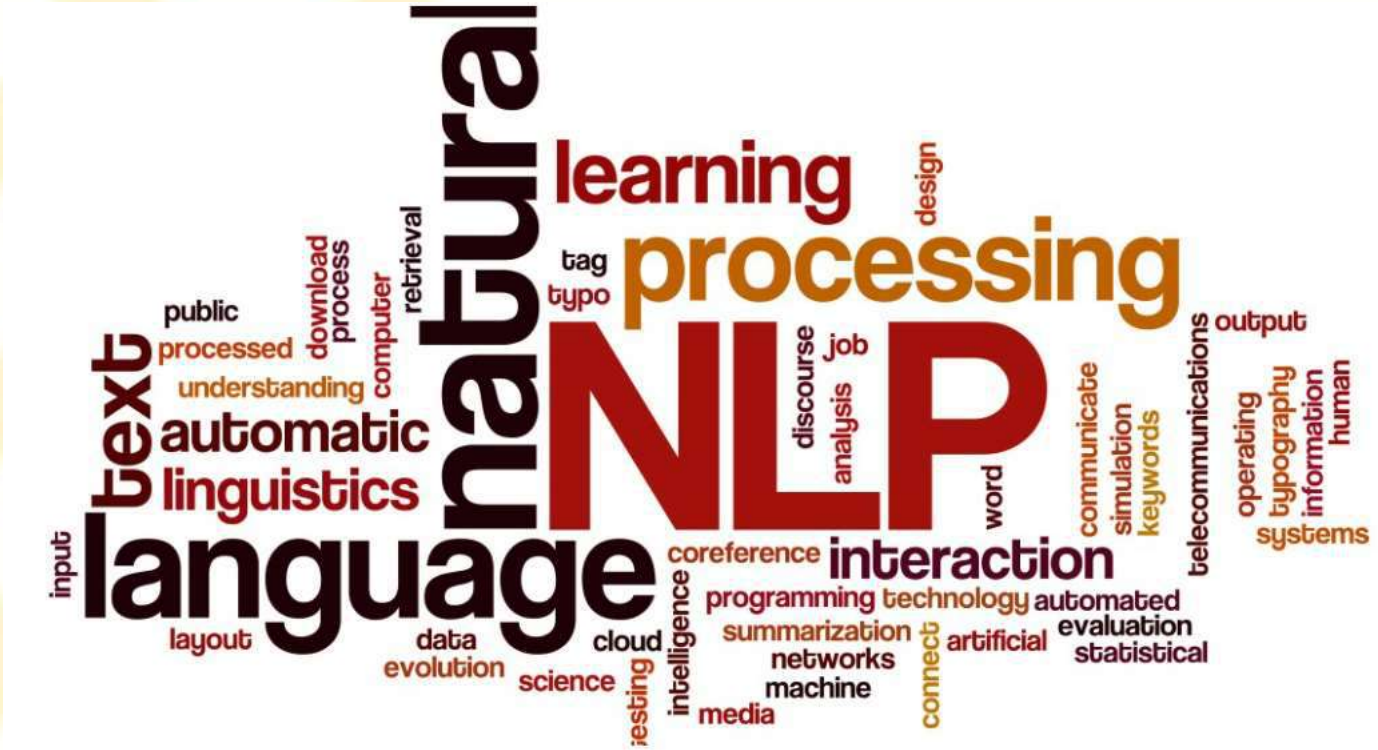
Techniques for Text Summarization

Keyphrase Extraction

Definition: Extracting keywords or phrases that capture main ideas.

Applications: Research papers, product descriptions, SEO.

Techniques: TF-IDF, RAKE, YAKE.

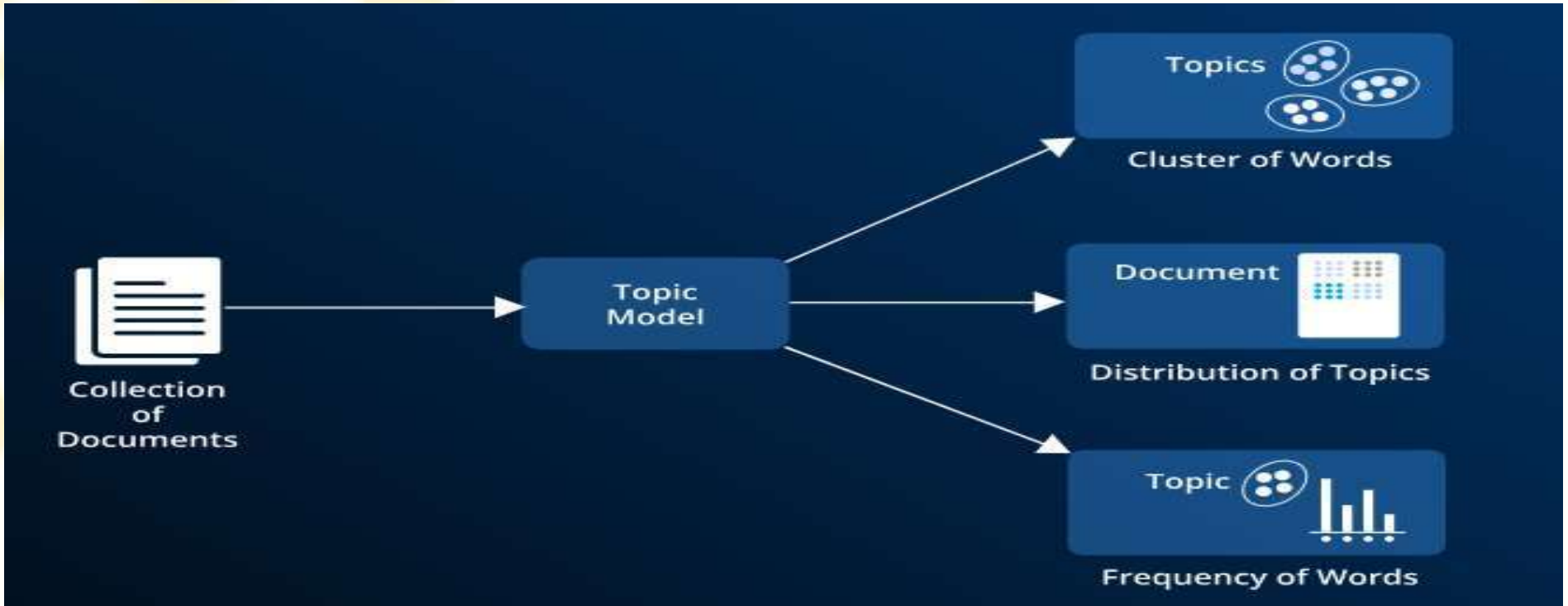


Topic Modeling

Definition0:It is unsupervised machine learning technique Identifying hidden topics (Thematic structures) in a corpus of documents.

Techniques: Latent Semantic Indexing(LSI) ,Latent Dirichlet Allocation (LDA), Non-Negative Matrix Factorization (NMF).

Applications: Text analytics, bioinformatics.



Automated Document Summarization

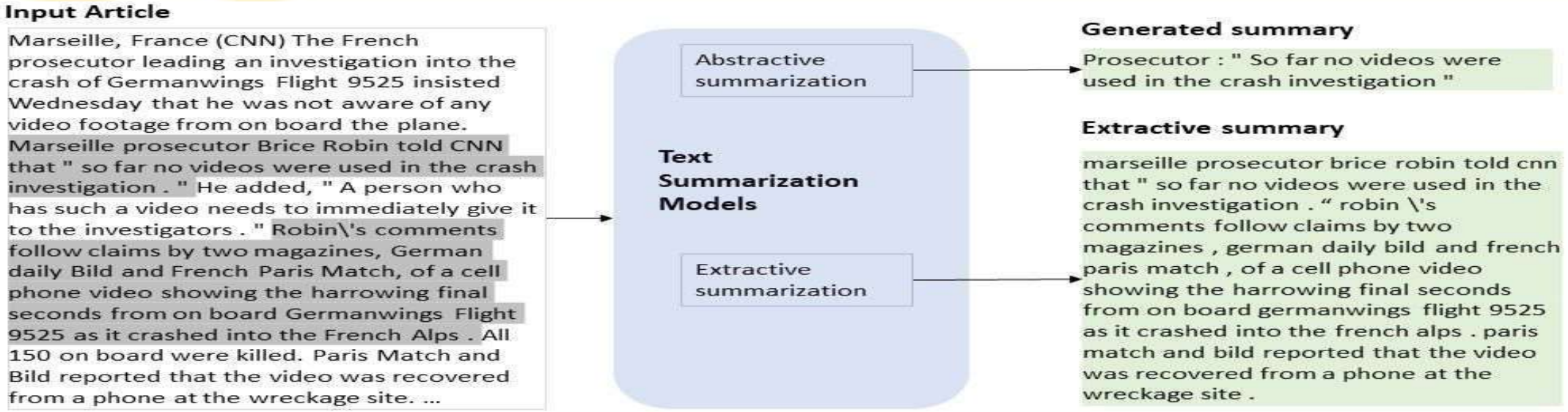
Definition: It refers to algorithms and models that produce brief summaries of longer documents by retaining original meaning and context.

Techniques : neural networks, transformer models, sequence-to-sequence are employed to achieve high quality summarization results.

Extraction-based (e.g., LexRank, TextRank).

Abstraction-based (e.g., GPT-based models).

Applications: Executive summaries, legal document analysis.



Extractive Vs Abstractive Summarization

Extractive Summarization Output:

While Huawei's sales fell 5 per cent from the same quarter a year earlier, South Korea's Samsung posted a bigger drop of 30 per cent, owing to disruption from the coronavirus in key markets such as Brazil, the United States and Europe, Canalys said. Huawei's overseas shipments fell 27 per cent in Q2 from a year earlier, but the company increased its dominance of the China market which has been faster to recover from COVID-19 and where it now sells over 70 per cent of its phones.

Abstractive Summarization Output:

Huawei overtakes Samsung as world's biggest seller of mobile phones in the second quarter of 2020. Sales of Huawei's 55.8 million devices compared to 53.7 million for south Korea's Samsung. Shipments overseas fell 27 per cent in Q2 from a year earlier, but company increased its dominance of the china market. Position as number one seller may prove short-lived once other markets recover, a senior Huawei employee says.

Important Concepts

Documents

- **A document** is usually an entity containing a whole body of text data with optional headers and other metadata information.
- **A corpus** usually consists of a collection of documents. These documents can be simple sentences or complete paragraphs of textual information.
- **Tokenized corpus** refers to a corpus where each document is tokenized or broken down into tokens, which are usually words

Text Normalization

Text normalization is the process of cleaning, normalizing, and standardizing textual data with techniques like removing special symbols and characters, removing extraneous HTML tags, removing stop-words, correcting spellings, stemming, and lemmatization

Important Concepts(Continued..)

Feature Extraction

- These techniques transform text into a format suitable for statistical or machine learning algorithms
- also known as vectorization, is the process of converting raw textual data into numerical vectors, as conventional algorithms require numerical input.
- This involves extracting meaningful features or attributes from text. Common methods include:

Bag of Words (Binary Features): Indicates the presence or absence of words or phrases in a document.
Bag of Words (Frequency Features): Measures how often words or phrases appear in a document.
TF-IDF (Term Frequency-Inverse Document Frequency): Weighs terms based on their frequency in a document and their rarity across documents.

Feature Matrix

- A **feature matrix** maps a collection of documents to features, where each row represents a document and each column represents a specific feature (e.g., a word or group of words). After feature extraction, documents or sentences are represented as feature matrices, which are then used for applying statistical and machine learning techniques in practical applications.

Feature matrix Example

Original Documents

"Classic tomato pasta with garlic and basil"

"Chocolate chip cookies with vanilla extract"

"Grilled salmon with lemon butter sauce"

2. Chosen Features (Columns)

We'll use 3 simple features:

IsIngredient (1/0): Does the word appear in our ingredient dictionary?

WordLength: How many letters the word has (longer words often more specific)

PositionScore: Where word appears (1.0 = first word, 0.1 = last word)

- **Relative Position Score:**

$$\text{Score} = \frac{\text{Word Index}}{\text{Total Words}}$$

- **Inverse Position Score:**

$$\text{Score} = 1 - \frac{\text{Word Index}}{\text{Total Words}}$$

Feature Matrix: example

Word	IsIngredient	WordLength	PositionScore
Classic	0	7	1.0
tomato	1	6	0.8
pasta	1	5	0.6
with	0	4	0.4
garlic	1	6	0.2
and	0	3	0.1
basil	1	5	0.1

1. **Filter**: Remove words where IsIngredient=0 ("Classic", "with", "and") =
2. **Score**: For remaining words, calculate:

```
Score = (IsIngredient) + (WordLength/10) + PositionScore
```

   - tomato: 1 + 0.6 + 0.8 = **2.4**
   - pasta: 1 + 0.5 + 0.6 = **2.1**
   - garlic: 1 + 0.6 + 0.2 = **1.8**
   - basil: 1 + 0.5 + 0.1 = **1.6**
3. **Result**: Top 2 scores → "tomato" (2.4), "pasta" (2.1)

Important Concepts(Continued..)

Singular Value Decomposition

Singular Value Decomposition (SVD) is a linear algebra technique widely used in summarization algorithms. It factorizes a real or complex matrix M of dimensions m×n into three components:

$$M_{m \times n} = U_{m \times m} S_{m \times n} V^T_{n \times n}$$

where we have the following decompositions:

- U is an $m \times m$ unitary matrix such that $U^T U = I_{m \times m}$ where I is the identity matrix. The columns of U indicate left singular vectors.
- S is a diagonal $m \, x \, n$ matrix with positive real numbers on the diagonal of the matrix. This is also often also represented as a vector of m values that indicate the singular values.
- V^T is a $n \times n$ unitary matrix such that $V^T V = I_{n \times n}$ where I is the identity matrix. The rows of V indicate right singular vectors.

Important Concepts(Continued..)

Implementation from scipy extract the top k singular values and also return the corresponding U, S and V matrices.

The following code snippet we will be using is in the utils.py file:

```
from scipy.sparse.linalg import svds
def low_rank_svd(matrix, singular_count=2):
   u, s, vt = svds(matrix, k=singular_count)
   return u, s, v^t
```

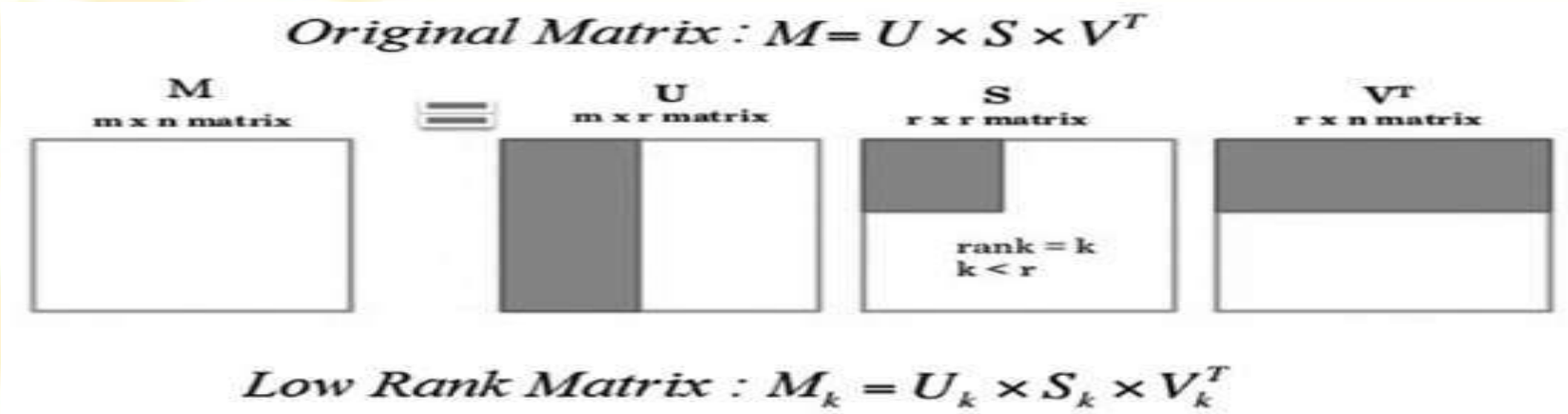


Figure 5-1. Singular Value Decomposition with low rank matrix approximation

Text Normalization

Text normalization–related code is available in the file “normalization.py “’

The main steps performed in text normalization include the following:

1. Sentence extraction(parse_document(document))
2. Unescape HTML escape sequences(unescape_html(parser, text))
3. Expand contractions(expand_contractions(text, CONTRACTION_MAP))
4. Lemmatize text(lemmatize_text(text))
5. Remove special characters(remove_special_characters(text))
6. Remove stop-words(remove_stopwords(text))

1.Sentence extraction (parse_document(document))

```
def parse_document(document):
    document = re.sub('\n', ' ', document)
    document = document.strip()
    sentences = nltk.sent_tokenize(document)
    sentences = [sentence.strip() for sentence in sentences]
    return sentences
```

2.Unescape HTML escape sequences (unescape_html(parser, text))

•Example:

& → &

& → &

```
from HTMLParser import HTMLParser
html_parser = HTMLParser()
def unescape_html(parser, text):
    return parser.unescape(text)
```

3.Expand contractions (expand_contractions(text, CONTRACTION_MAP))

It Converts shorten words to its full forms

Examples: can't → cannot

I'm → I am

4.Lemmatize text(lemmatize_text(text))

Example: running → run

better → good

if lemmatize:

text = lemmatize_text(text)

5.Remove special characters(remove_special_characters(text))

Input: "Hello! How are you? 😊 #NLP"

Output: "Hello How are you NLP"

6.Remove stop-words(remove_stopwords(text))

Input: "The quick brown fox jumps over the lazy dog."

Output: "quick brown fox jumps lazy dog."

The complete normalization function

```
def normalize_corpus(corpus, lemmatize=True, tokenize=False):
    normalized_corpus = []
    for text in corpus:
        text = html_parser.unescape(text)
        text = expand_contractions(text, CONTRACTION_MAP)
        if lemmatize:
            text = lemmatize_text(text)
        else:
            text = text.lower()
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        if tokenize:
            text = tokenize_text(text)
        normalized_corpus.append(text)
    return normalized_corpus
```

Output

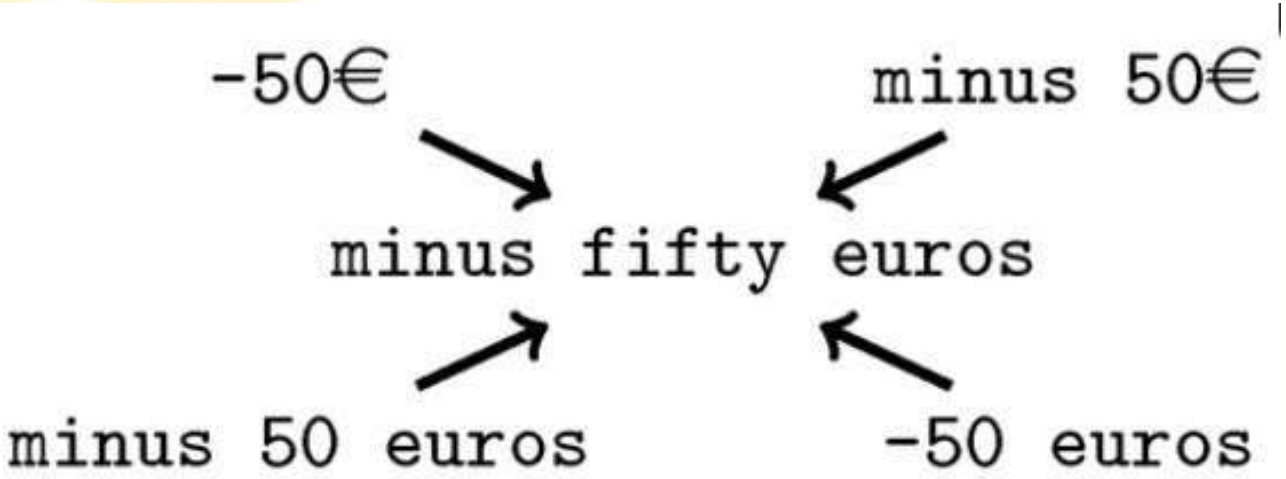
Input: ["I'm feeling gr8! 😊 Let's meet @ 5 PM."]

Output: ["I am feeling great Let us meet at 5 PM"]

Applications of Text Normalization

- Text Summarization
- Sentiment Analysis
- Machine Translation
- Search Engines

EXAMPLE of Text normalization(An example of equivalent phrases producing the same normalized output)



Feature Extraction

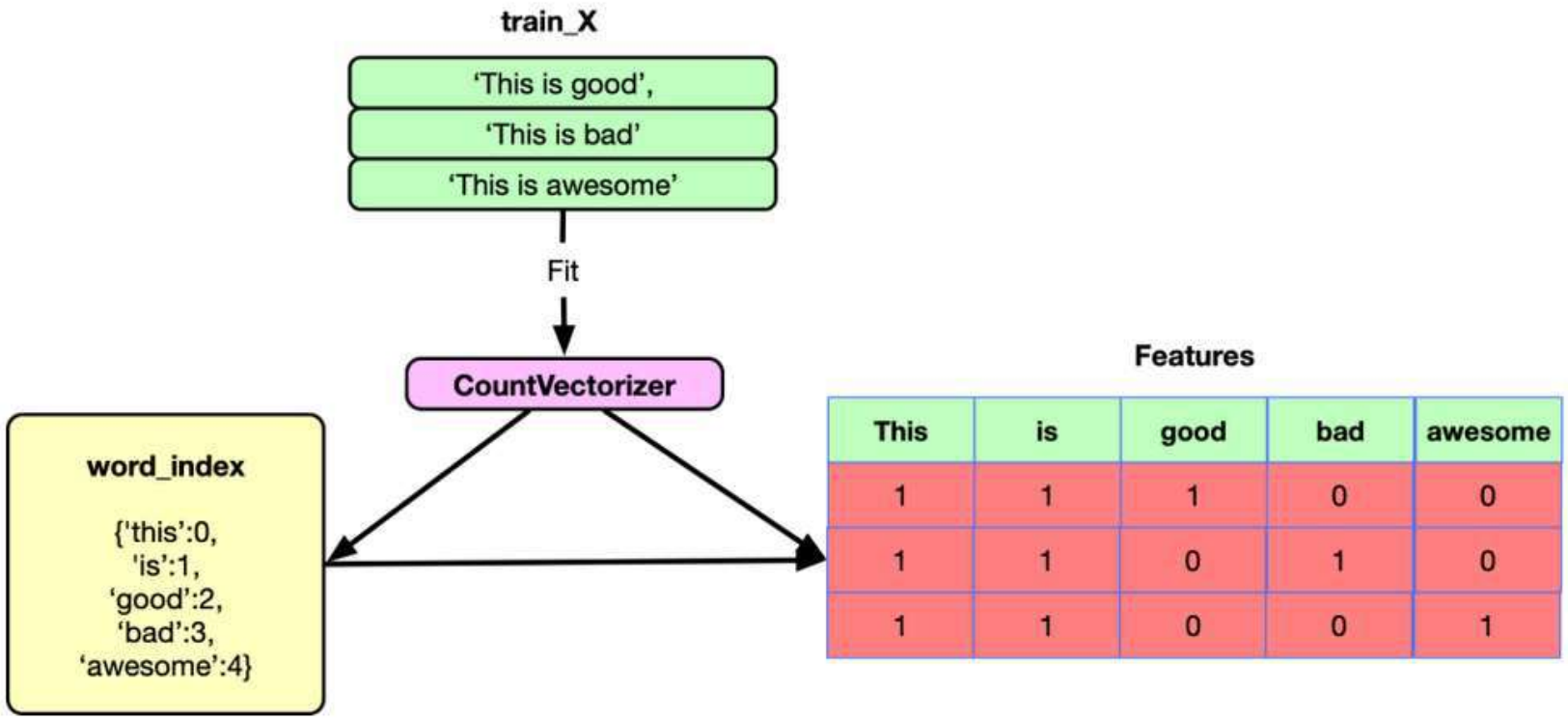
- It is transforming text data into numerical features for NLP tasks
- Essential for text classification, summarization, and sentiment analysis
- Various approaches like binary, frequency-based, and TF-IDF

Key Feature Extraction Methods

- **Binary term occurrence–based features**
- **Frequency bag of words–based features**
- **TF-IDF–weighted features**

Binary term occurrence–based features / Countvectorizer

It is a simple and flexible way of extracting features from Documents .A Countvectorizer model is a representation of text that describes the occurrence of words within a document.
The model is only concerned with whether known words occur in the document, not wherein the document.



basic snippet of using count vectorization

CODE:

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = ["We become what we think about", "Happiness is not something readymade."]

# initialize count vectorizer object
vect = CountVectorizer()

# get counts of each token (word) in text data
X = vect.fit_transform(corpus)

# convert sparse matrix to numpy array to view
X.toarray()

# view token vocabulary and counts
print("vocabulary", vect.vocabulary_)
print("shape", X.shape)
print('vectors: ', X.toarray())
```

Output:

Vocabulary *: {'we': 8, 'become': 1, 'what': 9, 'think': 7, 'about': 0, 'happiness': 2, 'is': 3, 'not': 4, 'something': 6, 'readymade': 5}*

Shape *: (2, 10)*

Vectors *: [[1 1 0 0 0 0 0 1 2 1]*
[0 0 1 1 1 1 1 0 0 0]]

Frequency bag of words–based features

What is Frequency BoW?

•Represents text as **word counts** (how often each word appears in a document)

•Ignores word order, only tracks term frequencies

•Example: *Document:* "apple orange apple" → {"apple":2, "orange":1}

•CODE:

```
from sklearn.feature_extraction.text import CountVectorizer
docs = [ "I like peanut butter and jelly sandwich with extra butter and jelly",  #doc1
         "I like cheese and pinapple pizza with extra cheese and pinaple ",  #doc2
         "I like hot chocolate with extra chocolate" ]  #doc3
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)
 # Get feature names (vocabulary)
print("Vocabulary:", vectorizer.get_feature_names_out())
# Show frequency matrix
print(X.toarray())
```

Output:

```
Vocabulary: ['and' 'butter' 'cheese' 'chocolate' 'extra' 'hot' 'jelly' 'like' 'peanut' 'pinaple' 'pinapple' 'pizza' 'sandwich' 'with']
[[2 2 0 0 1 0 2 1 1 0 0 0 1 1 ]
 [2 0 2 0 1 0 0 1 0 1 1 1 0 1 ]
 [0 0 0 2 1 1 0 1 0 0 0 0 0 1]]
```

TF-IDF–weighted features

- **TF (Term Frequency)**: How often a term appears in a document
  tf(t,d) = count of t in d / total terms in d
- **IDF (Inverse Document Frequency)**: How rare a term is across all documents
  idf(t) = log(total docs / docs containing t) + 1
- **TF-IDF = TF × IDF**: Highlights terms that are **frequent in a document but rare in the corpus**

When to Use TF-IDF

Best for: Technical documents, research papers, medium-length texts

Less ideal for: Very short texts (tweets), or when word order is critical

CODE:

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer()
X = tfidf.fit_transform(docs)
print("Vocabulary:", tfidf.get_feature_names_out())
print("TF-IDF Matrix:\n", X.toarray().round(2))
```

Output: Vocabulary: ['algorithms', 'deep', 'learning', 'machine', 'neural', 'networks', 'vision']

	algorithms	deep	learning	machine	neural	networks	vision
Doc 1	0.65	0.	0.39	0.65	0.	0.	0.
Doc 2	0.	0.45	0.27	0.	0.65	0.65	0.
Doc 3	0.	0.34	0.20	0.34	0.	0.	0.65

Keyphrase extraction feature matrix demo

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
documents = [    "machine learning algorithms",    "deep learning neural networks",    "machine vision and deep learning"]
def build_feature_matrix(documents, feature_type='frequency'):
    feature_type = feature_type.lower().strip()
    if feature_type == "binary":
        vectorizer = CountVectorizer(binary=True, min_df=1,ngram_range=(1, 1))
    elif feature_type == "frequency":
        vectorizer = CountVectorizer(binary=False, min_df=1,ngram_range=(1, 1))
    elif feature_type == "tfidf":
        vectorizer = TfidfVectorizer(min_df=1,ngram_range=(1, 1))
    else:
        raise Exception("Wrong feature type entered. Possible values:'binary', 'frequency', 'tfidf'")
    feature_matrix = vectorizer.fit_transform(documents).astype(float)
    return vectorizer, feature_matrix
# Compare all three methods
methods = ['binary', 'frequency', 'tfidf']
for method in methods:
    print(f"\n=== {method.upper()} Feature Matrix ===")
    vectorizer, matrix = build_feature_matrix(documents, method)

    # Print feature names (vocabulary)
    print("Vocabulary:", vectorizer.get_feature_names_out())

    # Convert matrix to dense array for readability
    print(matrix.toarray())
```

Output

```
=== BINARY Feature Matrix ===
Vocabulary: ['algorithms' 'and' 'deep' 'learning' 'machine' 'networks' 'neural'
 'vision']
[[1. 0. 0. 1. 1. 0. 0. 0.]
 [0. 0. 1. 1. 0. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0. 0. 1.]]

=== FREQUENCY Feature Matrix ===
Vocabulary: ['algorithms' 'and' 'deep' 'learning' 'machine' 'networks' 'neural'
 'vision']
[[1. 0. 0. 1. 1. 0. 0. 0.]
 [0. 0. 1. 1. 0. 1. 1. 0.]
 [0. 1. 1. 1. 1. 0. 0. 1.]]

=== TFIDF Feature Matrix ===
Vocabulary: ['algorithms' 'and' 'deep' 'learning' 'machine' 'networks' 'neural'
 'vision']
[[0.72033345 0.         0.         0.42544054 0.54783215 0.
  0.         0.        ]
 [0.         0.         0.44451431 0.34520502 0.         0.5844829
  0.5844829  0.        ]
 [0.         0.53409337 0.40619178 0.31544415 0.40619178 0.
  0.         0.53409337]]
```

Keyphrase Extraction

Definition: Extracting important and relevant terms/phrases from unstructured text.

Example:

Input Text: "Machine learning is a subset of artificial intelligence."
Extracted Keyphrases: "machine learning," "artificial intelligence.“

Applications of Keyphrase Extraction

- Semantic web
- Query-based search engines
- Recommendation systems
- Tagging systems
- Document similarity
- Translation

Example:

Search Engine: User query → Keyphrase extraction → Relevant results.
Recommendation System: Keyphrases from user history → Recommended products.

Techniques for Keyphrase Extraction

Collocations:

Phrases that frequently occur together (e.g., "machine learning," "artificial intelligence").

Weighted Tag–Based Phrase Extraction:

Assigning weights to phrases based on importance (e.g., TF-IDF).

Example:Weighted phrases with scores (e.g., "data science" = 0.85, "deep learning" = 0.92).

Collocations

Collocations are word groupings that appear together frequently beyond random chance

It introduces various methods to extract collocations, particularly using **n-grams**—sequences of words of length *n*.

Steps involved are :

1. **Extracting n-grams** from a text corpus.
2. **Counting their frequency** and ranking them.
3. **Using Python's NLTK library** to automate this process.

NLTK's built-in tools. It demonstrates how to:

Flatten a corpus into a single text string.

Compute n-grams using a sliding window.

Identify top collocations using frequency-based ranking.

Use NLTK's BigramCollocationFinder and TrigramCollocationFinder to find collocations based on raw frequency and Pointwise Mutual Information (PMI).

https://revaedu-my.sharepoint.com/:w:/g/personal/shilpa_mathpati_reva_edu_in/Ec9BC61oFYRFopF5Dm4dWf8Bwg5Fot_ijm2GNlLlE4UCeA?e=GccGrw

Weighted Tag–Based Phrase Extraction

Weighted Tag–Based Phrase Extraction is an approach to extract keyphrases by combining **shallow parsing** and **term weighting**. Inspired by research papers, this method follows a **two-step process**:

- **Extract noun phrase chunks** using **shallow parsing** with POS tagging.
- **Compute TF-IDF weights** for each phrase and return the highest-weighted ones.

This technique enhances keyphrase extraction by focusing on **important noun phrases** and ranking them based on their significance in the text. The process is demonstrated using a **sample corpus** from Wikipedia.

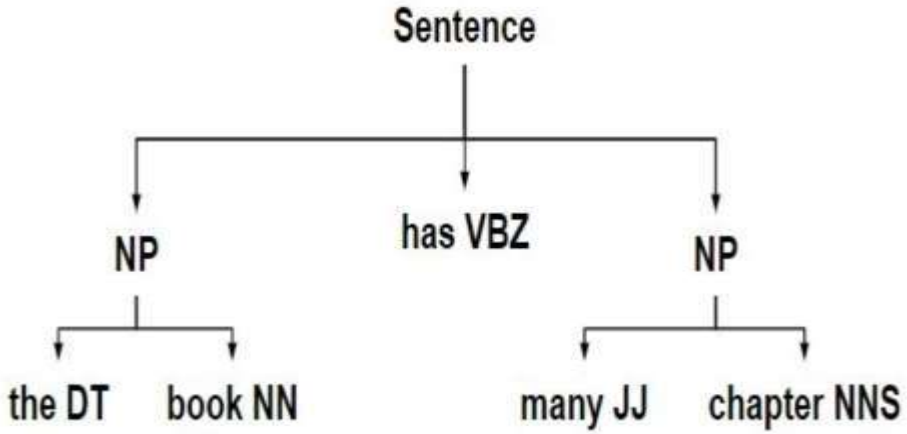
Extract noun phrase chunks using **shallow parsing** with POS tagging.

- Instead of performing deep parsing (full syntactic analysis),extracting **noun phrases (NPs)** using **chunking**.
- **Chunking** groups words together based on part-of-speech (POS) tagging.

- **Why Noun Phrases?**
  - Most **important concepts** in a text are noun phrases.
  - Example: *“machine learning”*, *“natural language processing”*.
- **Chunking Grammar Used:**
  - NP: {<DT>? <JJ>* <NN.*>+}
  - This matches **noun phrases (NPs)** that include:
    - DT (Determiner) → Optional (*e.g., "the"*)
    - JJ (Adjective) → Optional (*e.g., "modern"*)
    - NN.* (Noun) → One or more (*e.g., "processing"*)
- **Example Chunking Output:**
  - *"Natural language processing is a subfield of AI."*
  - Chunked Output: **["natural language processing", "subfield", "AI"]**

Extract noun phrase chunks using shallow parsing with POS tagging

- **Example Chunking Output:**
  - *"The book has many chapters."*
  - Chunked Output: **["The book", "has", "many chapters"]**




```
from nltk.chunk.regexp import ChunkString, ChunkRule, ChinkRule
from nltk.tree import Tree
from nltk.chunk import RegexpChunkParser
# ChunkString() starts with the flat tree
tree = Tree('S', [('the', 'DT'), ('book', 'NN'),  ('has', 'VBZ'), ('many', 'JJ'), ('chapters',
'NNS')])
# Initializing ChunkRule
chunk_rule = ChunkRule('<DT><NN.*><.*>*<NN.*>', 'chunk determiners and nouns')
# Another ChinkRule
chink_rule = ChinkRule('<VB.*>', 'chink verbs')
# Applying RegexpChunkParser
chunker = RegexpChunkParser([chunk_rule, chink_rule])
chunker.parse(tree)
```

Output:

```
Tree('S', [Tree('CHUNK', [('the', 'DT'), ('book', 'NN')]), ('has', 'VBZ'),
 Tree('CHUNK', [('many', 'JJ'), ('chapters', 'NNS')])])
```

Compute TF-IDF weights for each chunk and return the top weighted phrases

After extracting noun phrases, compute their **importance** using **TF-IDF (Term Frequency-Inverse Document Frequency)**.

What is TF-IDF?

TF-IDF is a statistical measure used to evaluate how important a word or phrase is in a document relative to a collection of documents.

Formula:

TF–IDF=TF×IDF

Term Frequency (TF): Measures how often a word appears in a document.

TF=Number of times the word appears in the document/Total words in the document

Inverse Document Frequency (IDF): Measures how unique a word is across multiple documents.

IDF=log(Total number of documents/Number of documents containing the word)

Final TF-IDF Score:A **higher TF-IDF score** means the phrase is **important and unique** in the document.

Compute TF-IDF weights for each chunk and return the top weighted phrases

Assume we have a corpus of three sentences:

"Artificial intelligence is a branch of computer science."

"Deep learning and machine learning are subsets of artificial intelligence."

"Natural language processing is an AI application."

Phrase	TF	IDF	TF-IDF
artificial intelligence	2	0.477	0.954
deep learning	1	0.693	0.693
machine learning	1	0.693	0.693
computer science	1	0.693	0.693
natural language processing	1	0.693	0.693

From the TF-IDF scores, **"artificial intelligence"** has the highest importance in the text.

Topic Modeling

- Latent Semantic Indexing
- Latent Dirichlet Allocation
- Non-negative Matrix Factorization
  - Extracting Topics from Product Reviews(Case Study)

Automated Document Summarization

- Latent Semantic Analysis
- TextRank
  - Summarizing a Product Description(Case Study)

Latent Semantic Indexing

Key Idea:

- **Dimensionality Reduction** – It applies **SVD** to a term-document matrix, reducing it to a lower-dimensional space while preserving the most important semantic structures.
- **Captures Latent Semantics** – It identifies ***hidden relationships between words*** and documents by analyzing co-occurrence patterns, making it robust against synonymy and polysemy.
- Improves Retrieval & Clustering – Documents with similar meanings but different wording are mapped closer in the reduced space, enhancing topic modeling, search accuracy, and text categorization.
- Limitations – LSI assumes a linear structure in language and does not handle polysemy as effectively as more advanced models like LDA (Latent Dirichlet Allocation) or neural embeddings.

Example for Latent Semantic Indexing

m=6 terms
- t1: bak(e,ing)
- t2: recipe(s)
- t3: bread
- t4: cake
- t5: pastr(y,ies)
- t6: pie

n=5 documents
- d1: How to bake bread without recipes
- d2: The classic art of Viennese Pastry
- d3: Numerical recipes: the art of scientific computing
- d4: Breads, pastries, pies and cakes: quantity baking recipes
- d5: Pastry: a book of best French recipes

$$A = \begin{pmatrix} 0.5774 & 0.0000 & 0.0000 & 0.4082 & 0.0000 \\ 0.5774 & 0.0000 & 1.0000 & 0.4082 & 0.7071 \\ 0.5774 & 0.0000 & 0.0000 & 0.4082 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.4082 & 0.0000 \\ 0.0000 & 1.0000 & 0.0000 & 0.4082 & 0.7071 \\ 0.0000 & 0.0000 & 0.0000 & 0.4082 & 0.0000 \end{pmatrix}$$

Example 2 for Latent Semantic Indexing (2)

$$A = \begin{pmatrix} 0.2670 & -0.2567 & 0.5308 & -0.2847 \\ 0.7479 & -0.3981 & -0.5249 & 0.0816 \\ 0.2670 & -0.2567 & 0.5308 & -0.2847 \\ 0.1182 & -0.0127 & 0.2774 & 0.6394 \\ 0.5198 & 0.8423 & 0.0838 & -0.1158 \\ 0.1182 & -0.0127 & 0.2774 & 0.6394 \end{pmatrix}$$

U

$$\times \begin{pmatrix} 1.6950 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 1.1158 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.8403 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.4195 \end{pmatrix}$$

Δ

$$\times \begin{pmatrix} 0.4366 & 0.3067 & 0.4412 & 0.4909 & 0.5288 \\ -0.4717 & 0.7549 & -0.3568 & -0.0346 & 0.2815 \\ 0.3688 & 0.0998 & -0.6247 & 0.5711 & -0.3712 \\ -0.6715 & -0.2760 & 0.1945 & 0.6571 & -0.0577 \end{pmatrix}$$

V^T

Example 2 for Latent Semantic Indexing (3)

$$A_3 = \begin{pmatrix} 0.4971 & -0.0330 & 0.0232 & 0.4867 & -0.0069 \\ 0.6003 & 0.0094 & 0.9933 & 0.3858 & 0.7091 \\ 0.4971 & -0.0330 & 0.0232 & 0.4867 & -0.0069 \\ 0.1801 & 0.0740 & -0.0522 & 0.2320 & 0.0155 \\ -0.0326 & 0.9866 & 0.0094 & 0.4402 & 0.7043 \\ 0.1801 & 0.0740 & -0.0522 & 0.2320 & 0.0155 \end{pmatrix} = U_3 \times \Delta_3 \times V_3^T$$

LSI implementation in Python

Steps in the Implementation

1.Preprocess the text: Convert raw text into a numerical representation using TF-IDF.

2.Apply Singular Value Decomposition (SVD): Reduce dimensions while capturing latent topics.

3.Visualize the reduced dimensions: Explore document clustering in a 2D space.

CODE:

```
import gensim
from gensim import corpora, models
from gensim.parsing.preprocessing import preprocess_string
# Sample toy corpus
documents =["The fox jumps over the dog",
"The fox is very clever and quick",
"The dog is slow and lazy",
"The cat is smarter than the fox and the dog",
"Python is an excellent programming language",
"Java and Ruby are other programming languages",
"Python and Java are very popular programming languages",
"Python programs are smaller than Java programs"]
# Step 1: Preprocess the text
processed_docs = [preprocess_string(doc) for doc in documents]
```

```
# Step 2: Create a Dictionary (word -> id mapping)
dictionary = corpora.Dictionary(processed_docs)

# Step 3: Convert documents to Bag-of-Words format
corpus = [dictionary.doc2bow(doc) for doc in processed_docs]

# Step 4: Apply TF-IDF transformation
tfidf = models.TfidfModel(corpus)
corpus_tfidf = tfidf[corpus]

# Step 5: Apply LSI (Latent Semantic Indexing)
num_topics = 2  # Number of topics to extract
lsi_model = models.LsiModel(corpus_tfidf, id2word=dictionary, num_topics=num_topics)

# Print the extracted topics
for idx, topic in lsi_model.print_topics(num_topics=num_topics, num_words=5):
    print(f"Topic {idx+1}: {topic}")
```

OUTPUT:

Topic 1: 0.460*"program" + 0.401*"language" + 0.397*"java" + 0.397*"python" + 0.304*"popular"
Topic 2: -0.459*"fox" + -0.459*"dog" + -0.444*"jump" + -0.322*"smarter" + -0.322*"cat“

OR(If Num_topics=3)

Topic 1: -0.460*"program" + -0.401*"languag" + -0.397*"java" + -0.397*"python" + -0.304*"popular"
Topic 2: 0.459*"dog" + 0.459*"fox" + 0.444*"jump" + 0.322*"cat" + 0.322*"smarter"
Topic 3: -0.474*"quick" + -0.474*"clever" + 0.474*"lazi" + 0.474*"slow" + -0.224*"fox"

Latent Dirichlet Allocation

Key Idea :

LDA is a **generative probabilistic model** used for **topic modeling** in NLP.

It assumes that:

1.Documents are mixtures of topics – Each document is composed of multiple topics, where each topic contributes a certain proportion to the document.

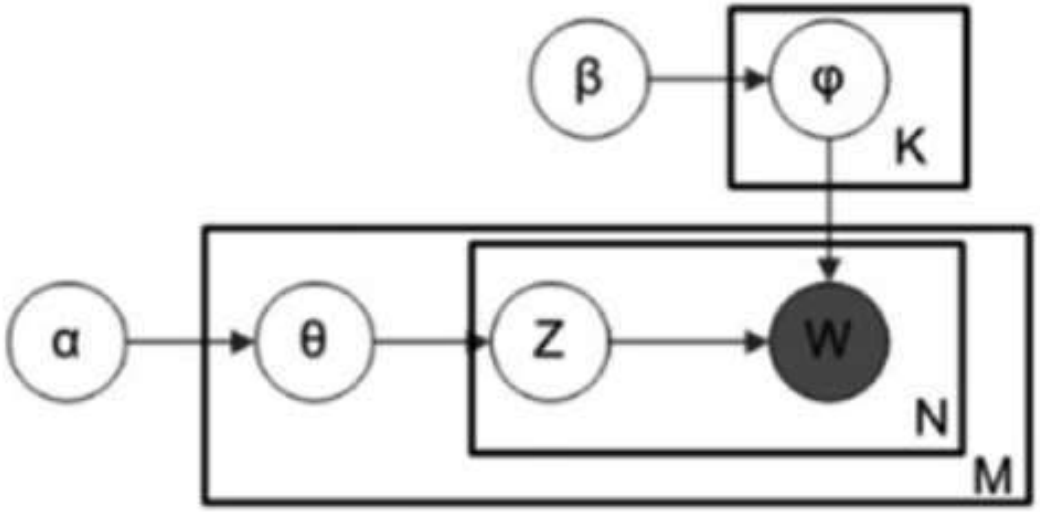
2.Topics are distributions over words – Each topic is a probability distribution over words in the vocabulary.

3.Dirichlet Priors for Distribution Control –

1. A **Dirichlet prior** is applied to the **document-topic distribution**, ensuring sparsity (documents have a few dominant topics).
2. Another **Dirichlet prior** is applied to the **topic-word distribution**, ensuring each topic has a few dominant words.

4.Word Assignment via Probabilistic Sampling – LDA uses **Bayesian inference** techniques like **Gibbs Sampling** or **Variational Inference** to determine the topic assignments for words and documents.

LDA plate notation



- K is the number of topics
- N is the number of words in the document
- M is the number of documents to analyse
- α is the Dirichlet-prior concentration parameter of the per-document topic distribution
- β is the same parameter of the per-topic word distribution
- φ(k) is the word distribution for topic k
- θ(i) is the topic distribution for document i
- z(i,j) is the topic assignment for w(i,j)
- w(i,j) is the j-th word in the i-th document
- φ and θ are Dirichlet distributions, z and w are multinomials.

Plates (Rectangles) Represent Repeated Variables:

The outer plate (M) represents documents (each document has its own topic distribution).

The inner plate (N) represents words within a document (each word is assigned a topic).

Random Variables:

α (alpha): Dirichlet prior controlling topic sparsity in documents (higher = more balanced, lower = fewer dominant topics).

θ (theta, inside D-plate): Document-topic distribution (each document has a probability distribution over topics).

β (beta): Dirichlet prior controlling word distribution in topics.

ϕ (phi, outside plates): Topic-word distribution (each topic has a probability distribution over words).

z (inside N-plate): Topic assignment for each word in a document (which topic a word elongs to).

w (inside N-plate, observed): The actual words in the document (these are observed in the data).

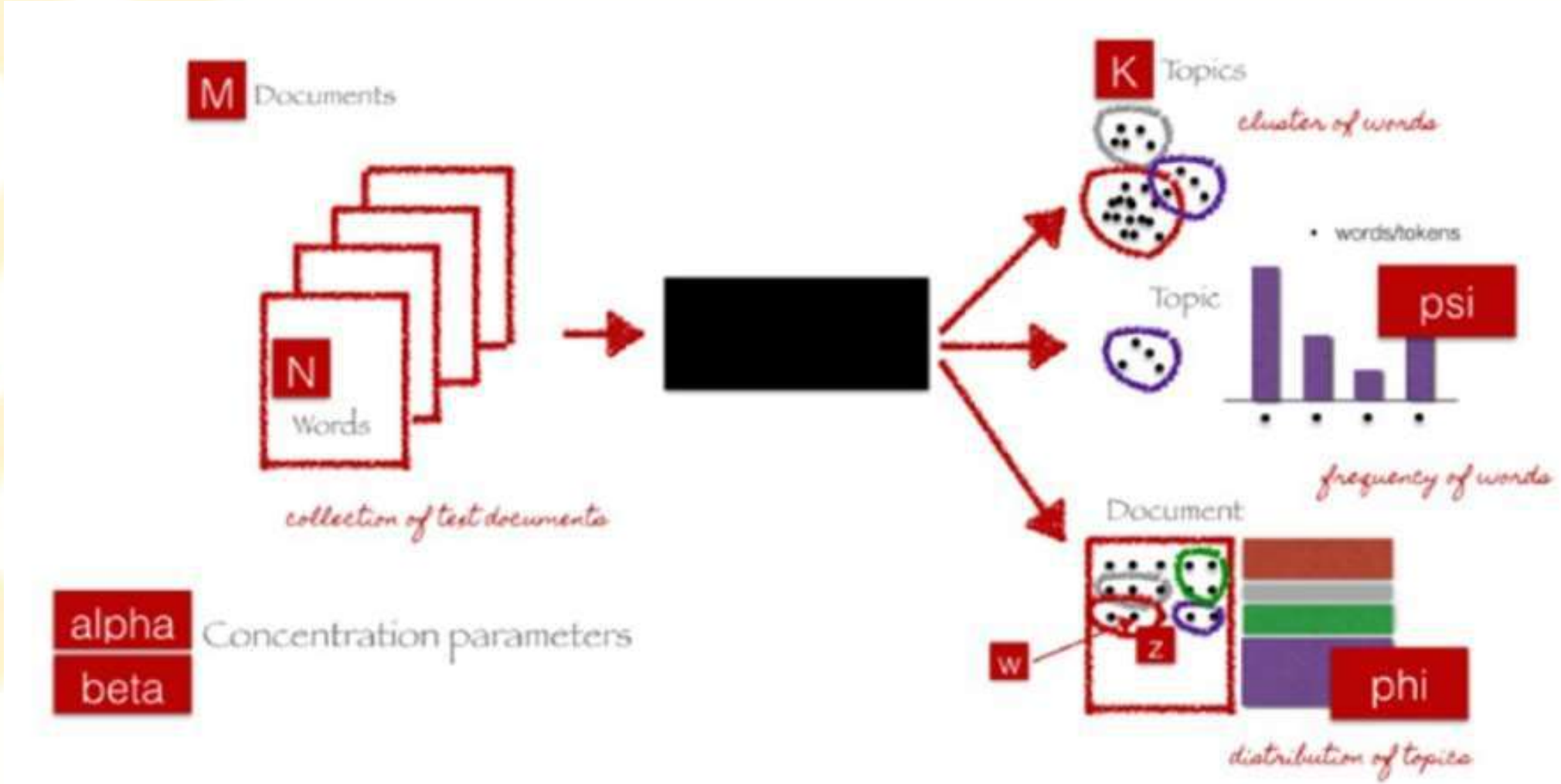
Arrows Represent Dependencies:

θ → z → w: First, pick a topic from the document's topic distribution (θ), then generate a word from that topic's word distribution (ϕ).

β → ϕ: Topics are distributions over words, drawn from a Dirichlet prior.

α → θ: Documents are distributions over topics, drawn from a Dirichlet prior.

M documents, N number of words in the documents,
and K total number of topics we want to generate.



LDA implementation using Python


```
from sklearn.feature_extraction.text import TfidfVectorizer, ENGLISH_STOP_WORDS
from sklearn.decomposition import LatentDirichletAllocation
import re

# Step 1: Normalize the corpus by removing stopwords and non-alphanumeric characters
def normalize_corpus(corpus):
    normalized_corpus = []
    for doc in corpus:
        # Convert to lowercase
        doc = doc.lower()
        # Remove non-alphanumeric characters (e.g., punctuation)
        doc = re.sub(r'[^a-z\s]', '', doc)
        # Remove stopwords
        doc = ' '.join([word for word in doc.split() if word not in ENGLISH_STOP_WORDS])
        normalized_corpus.append(doc)
    return normalized_corpus

# Step 2: Build the TF-IDF feature matrix
def build_feature_matrix(corpus, feature_type='tfidf'):
    if feature_type == 'tfidf':
        vectorizer = TfidfVectorizer(max_df=0.95, min_df=2)
        tfidf_matrix = vectorizer.fit_transform(corpus)
    return vectorizer, tfidf_matrix
```

```
# Step 3: Extract top terms from the topic's components and weights
def get_topics_terms_weights(weights, feature_names, n_top_words=10):
    topics = []
    for topic_idx, topic in enumerate(weights):
        topic_terms = [feature_names[i] for i in topic.argsort()[:-n_top_words - 1:-1]]
        topics.append(f"Topic {topic_idx}: {', '.join(topic_terms)}")
    return topics

# Step 4: Main program
def main():
    # Example corpus (replace with your actual corpus)
    toy_corpus = ["The fox jumps over the dog",
"The fox is very clever and quick",
"The dog is slow and lazy",
"The cat is smarter than the fox and the dog",
"Python is an excellent programming language",
"Java and Ruby are other programming languages",
"Python and Java are very popular programming languages",
"Python programs are smaller than Java programs"]

    # Step 4.1: Normalize the corpus
    norm_corpus = normalize_corpus(toy_corpus)

    # Step 4.2: Build the TF-IDF feature matrix
    vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus, feature_type='tfidf')
```

```
# Step 4.3: Fit the LDA model
    total_topics = 2  # You can change the number of topics as needed
    lda = LatentDirichletAllocation(n_components=total_topics, max_iter=100, learning_method='online',
learning_offset=50., random_state=42)
    lda.fit(tfidf_matrix)

    # Step 4.4: Get the feature names and the weights for topics
    feature_names = vectorizer.get_feature_names_out()
    weights = lda.components_

    # Step 4.5: Generate and print topics from their terms and weights
    topics = get_topics_terms_weights(weights, feature_names)
    for topic in topics:
        print(topic)

# Run the program
if __name__ == '__main__':
    main()
```

Output:

Topic 0: python, programming, java, languages, dog, fox

Topic 1: fox, dog, python, java, programming, languages

Non-negative Matrix Factorization

KeyIdea:

- Given a matrix **V** (with non-negative values, e.g., image pixels, word counts), NNMF finds two matrices **W** and **H** such that when multiplied:

  $V \approx W \times H$

  W(Document-Topic Matrix): This matrix represents the **basis components** (features).
  H(Topic-Term Matrix): This matrix represents the **coefficients** or **weights** for each basis component.
- NNMF attempts to **factorize** the original matrix **V** into two smaller matrices **W** (basis) and **H** (coefficients), which can be combined to reconstruct **V**. The process ensures that all matrices involved (**V**, **W**, and **H**) have **non-negative values**.
- The non-negativity constraint makes NNMF particularly suitable for applications where negative values don't make sense. This is common in areas such as:

  Text mining (e.g., word counts in documents).
  Image processing (e.g., pixel intensities).
  Recommendation systems (e.g., user-item interaction counts).

How NNMF Works:

1. **Initialization:** Start with random non-negative values for **W** and **H**.
2. **Optimization:** Use iterative optimization (like gradient descent) to update **W** and **H** such that the product **W × H** gets closer to **V**. The objective is to minimize the difference between **V** and **W × H**.
3. **Result:** The two matrices **W** and **H** provide a compact, interpretable representation of the original data.

The implementation of NNMF


```
from sklearn.feature_extraction.text import TfidfVectorizer, ENGLISH_STOP_WORDS
from sklearn.decomposition import NMF
import re

# Step 1: Normalize the corpus (remove stopwords, punctuation, and lowercase)
def normalize_corpus(corpus):
    normalized_corpus = []
    for doc in corpus:
        # Convert to lowercase
        doc = doc.lower()
        # Remove non-alphanumeric characters (e.g., punctuation)
        doc = re.sub(r'[^a-z\s]', '', doc)
        # Remove stopwords
        doc = ' '.join([word for word in doc.split() if word not in ENGLISH_STOP_WORDS])
        normalized_corpus.append(doc)
    return normalized_corpus

# Step 2: Build the TF-IDF feature matrix
def build_feature_matrix(corpus, feature_type='tfidf'):
    if feature_type == 'tfidf':
        vectorizer = TfidfVectorizer(max_df=0.95, min_df=2)
        tfidf_matrix = vectorizer.fit_transform(corpus)
    return vectorizer, tfidf_matrix
```

```
# Step 3: Extract top terms from the topic's components and weights
def get_topics_terms_weights(weights, feature_names, n_top_words=10):
    topics = []
    for topic_idx, topic in enumerate(weights):
        topic_terms = [feature_names[i] for i in topic.argsort()[:-n_top_words - 1:-1]]
        topics.append(f"Topic {topic_idx}: {', '.join(topic_terms)}")
    return topics
# Step 4: Main program
def main():
    # Example corpus (replace with your actual corpus)
    toy_corpus = [
        "I love programming in Python.",
        "Machine learning is fascinating.",
        "Data science involves programming and machine learning.",
        "Python is great for data analysis.",
        "Data science and AI are connected fields."
    ]
    # Step 4.1: Normalize the corpus
    norm_corpus = normalize_corpus(toy_corpus)
    # Step 4.2: Build the TF-IDF feature matrix
    vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus, feature_type='tfidf')
```

```
# Step 4.3: Fit the NMF model (NNMF) - Fixed the alpha issue
    total_topics = 2  # You can change the number of topics as needed
    nmf = NMF(n_components=total_topics, random_state=42, alpha_W=.1, alpha_H=.1, l1_ratio=.5)
    nmf.fit(tfidf_matrix)

    # Step 4.4: Get the feature names and the weights for topics
    feature_names = vectorizer.get_feature_names_out()
    weights = nmf.components_

    # Step 4.5: Generate and print topics from their terms and weights
    topics = get_topics_terms_weights(weights, feature_names)
    for topic in topics:
        print(topic)

# Run the program
if __name__ == '__main__':
    main()
```

Output:

Topic 0: python, programming, data, analysis, machine, learning, ai, science, connected, great

Topic 1: learning, machine, data, science, ai, programming, python, analysis, great, fields

Extracting Topics from Product Reviews

HOMEWORK for STUDENTS

Automated Document Summarization

- Latent Semantic Analysis
- TextRank
- Summarizing a Product Description

Latent Semantic Analysis (LSA)

is a mathematical technique used in **automated document summarization** to identify hidden relationships between words and sentences in a text.
It is based on **Singular Value Decomposition (SVD)**, which helps reduce dimensionality and extract the most relevant information.

1. Text Representation as a Matrix

The input text is converted into a Term-Document Matrix (TDM) or TF-IDF matrix using a vectorizer (e.g., TfidfVectorizer in scikit-learn).

Each row represents a unique word, and each column represents a sentence/document.

2. Applying Singular Value Decomposition (SVD)

SVD decomposes the matrix into three smaller matrices:

$A=U \cdot S \cdot V^T$

where: U: Represents words and their relationships.
S: Contains singular values (importance scores).
V: Represents sentences and their importance in the text.

SVD helps remove noise and captures the latent structure in the text.

3. Ranking Sentences for Summarization

Sentences with the highest singular values are considered the most important.

Top-ranked sentences are selected for the summary.

LSA Implementation using Python


```
import numpy as np
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize, word_tokenize
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
import string

nltk.download('punkt')
nltk.download('stopwords')

def preprocess_text(text):
    """Normalize and preprocess text for LSA"""
    sentences = sent_tokenize(text)

    # Lowercase, remove punctuation and stopwords
    processed_sentences = []
    for sentence in sentences:
        words = word_tokenize(sentence.lower())
        words = [word for word in words
                 if word not in stopwords.words('english')
                 and word not in string.punctuation]
        processed_sentences.append(' '.join(words))
```

```
return sentences, processed_sentences  # Return both original and processed

def build_feature_matrix(sentences, feature_type='tfidf'):
    """Create feature matrix using specified method"""
    if feature_type == 'frequency':
        vectorizer = CountVectorizer()
    elif feature_type == 'tfidf':
        vectorizer = TfidfVectorizer()
    else:
        raise ValueError("feature_type must be 'frequency' or 'tfidf'")

    return vectorizer.fit_transform(sentences)

def lsa_summarizer(text, num_sentences=3, num_topics=5, feature_type='tfidf', sv_threshold=0.5):
    """
    LSA-based document summarizer

    Parameters:
    - text: Input document text
    - num_sentences: Number of sentences in summary
    - num_topics: Number of latent topics
    - feature_type: 'frequency' or 'tfidf'
    - sv_threshold: Singular value threshold for sentence selection
    """
```

```
# Preprocess text
    original_sentences, processed_sentences = preprocess_text(text)
    # Build feature matrix
    feature_matrix = build_feature_matrix(processed_sentences, feature_type)
    # Perform SVD
    svd = TruncatedSVD(n_components=num_topics)
    lsa_matrix = svd.fit_transform(feature_matrix)
    # Get sentence scores based on singular values
    sentence_scores = np.sum(np.abs(lsa_matrix), axis=1)
    normalized_scores = sentence_scores / np.max(sentence_scores)
    # Select sentences above threshold
    selected_indices = np.where(normalized_scores >= sv_threshold)[0]
    # If not enough sentences meet threshold, take top scoring ones
    if len(selected_indices) < num_sentences:
        selected_indices = np.argsort(sentence_scores)[-num_sentences:]
    # Sort selected indices to maintain original order
    selected_indices = sorted(selected_indices)
    # Generate summary
    summary = ' '.join([original_sentences[i] for i in selected_indices[:num_sentences]])
    return summary
```

```
# Example usage with Skyrim description
skyrim_text = """Skyrim is an open world action role-playing video
game developed by Bethesda Game Studios and published by Bethesda
Softworks.
Players may navigate the game world more quickly by riding horses, or
by utilizing a fast-travel system which allows them to warp to
previously Players have the option to develop their character.
At the beginning of the game, players create their character by selecting
one of several races, including humans, orcs, elves and anthropomorphic
cat or lizard-like creatures, and then customizing their character's
appearance.discovered locations...."""  # Your full text here
summary = lsa_summarizer(
    skyrim_text,
    num_sentences=3,
    num_topics=5,
    feature_type='frequency',
    sv_threshold=0.5
)
print("LSA Summary:")
print(summary)
```

Output:

LSA Summary:

Skyrim's dragon-filled world offers unparalleled freedom to explore. As the Dragonborn, you'll develop your skills through epic battles. The rich fantasy lore unfolds through dynamic quests and characters.

TextRank

- TextRank is a **graph-based ranking algorithm** inspired by Google's **PageRank**.
- PageRank ranks web pages based on **link importance**, treating links as **votes** from one page to another.
- The key idea is that a page's importance depends not only on the number of incoming links but also on the importance of the linking pages.
- In **TextRank**, sentences in a document are treated as **nodes in a graph**, and **edges represent similarity** between sentences.
- Sentences that are more connected (i.e., similar to many important sentences) receive **higher ranks**, and the **top-ranked sentences** form the summary.

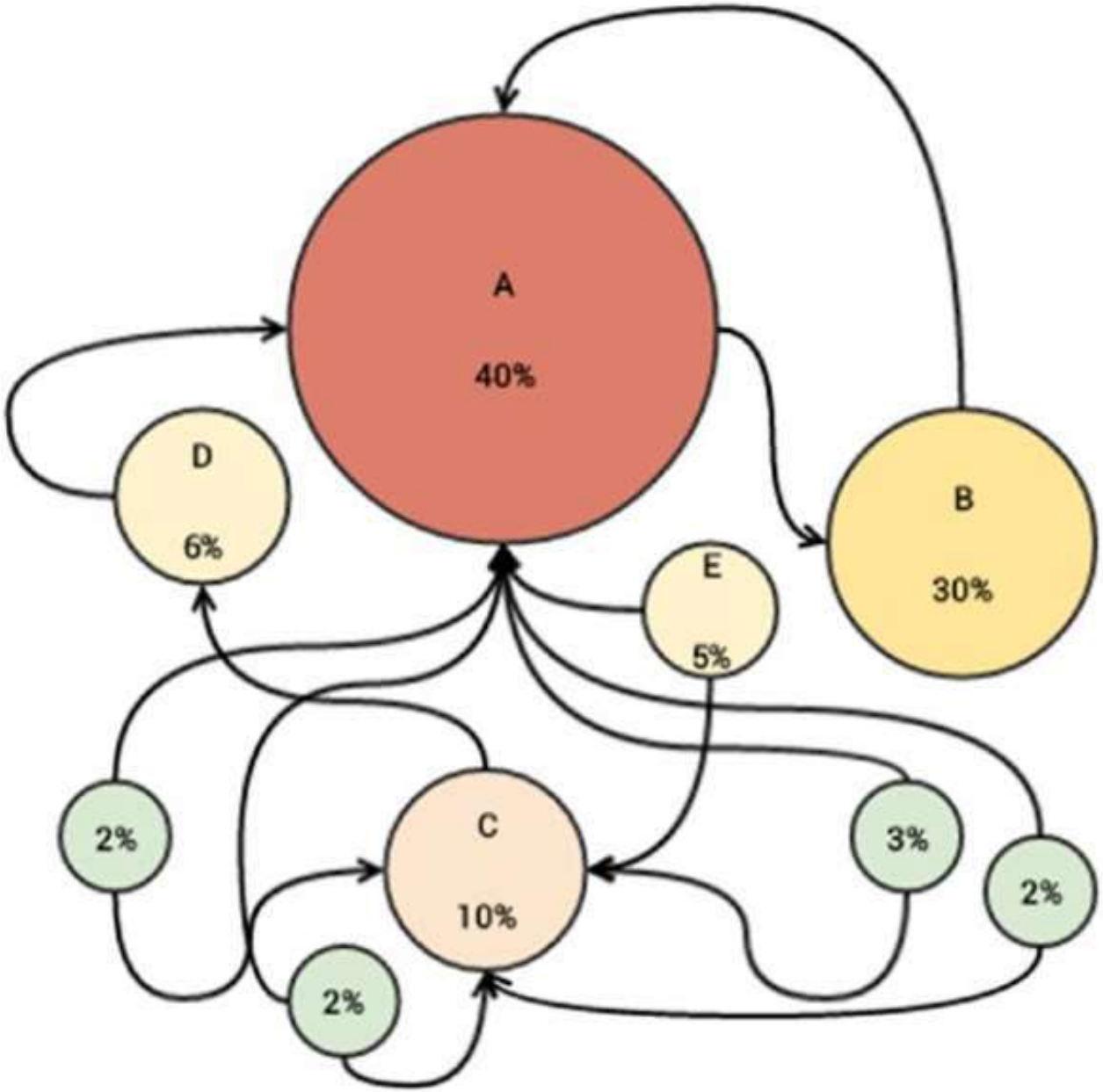


Figure 5-4. *PageRank scores for a simple network*

TextRank

Consider a directed graph represented as G V E= ( ), such that

V represents the set of vertices or pages and

E represents the set of edges or links, and E is a subset of V V´ .

Assuming we have a given page Vi for which we want to compute the PageRank, we canmathematically define it as

$$S(V_i) = (1-d) + d * \sum_{j \in In(v_i)} \frac{1}{|Out(V_j)|} S(V_j)$$

- S(Vi) - the weight of webpage i
- d - damping factor, in case of no outgoing links
- In(Vi) - inbound links of i, which is a set
- Out(Vj) - outgoing links of j, which is a set
- |Out(Vj)| - the number of outbound links

damping factor usually having a value between 0 to 1—ideally it is set to 0.85

Steps in TextRank

TextRank algorithm main steps :

1. Tokenize and extract sentences from the document to be summarized.
2. Decide on the number of sentences k that we want in the final summary.
3. Build document term feature matrix using weights like TF-IDF or Bag of Words.
4. Compute a document similarity matrix by multiplying the matrix with its transpose.
5. Use these documents (sentences in our case) as the vertice and the similarities between each pair of documents as the weight or score coefficient mentioned earlier and feed them to the PageRank algorithm.
6. Get the score for each sentence.
7. Rank the sentences based on score and return the top k sentences

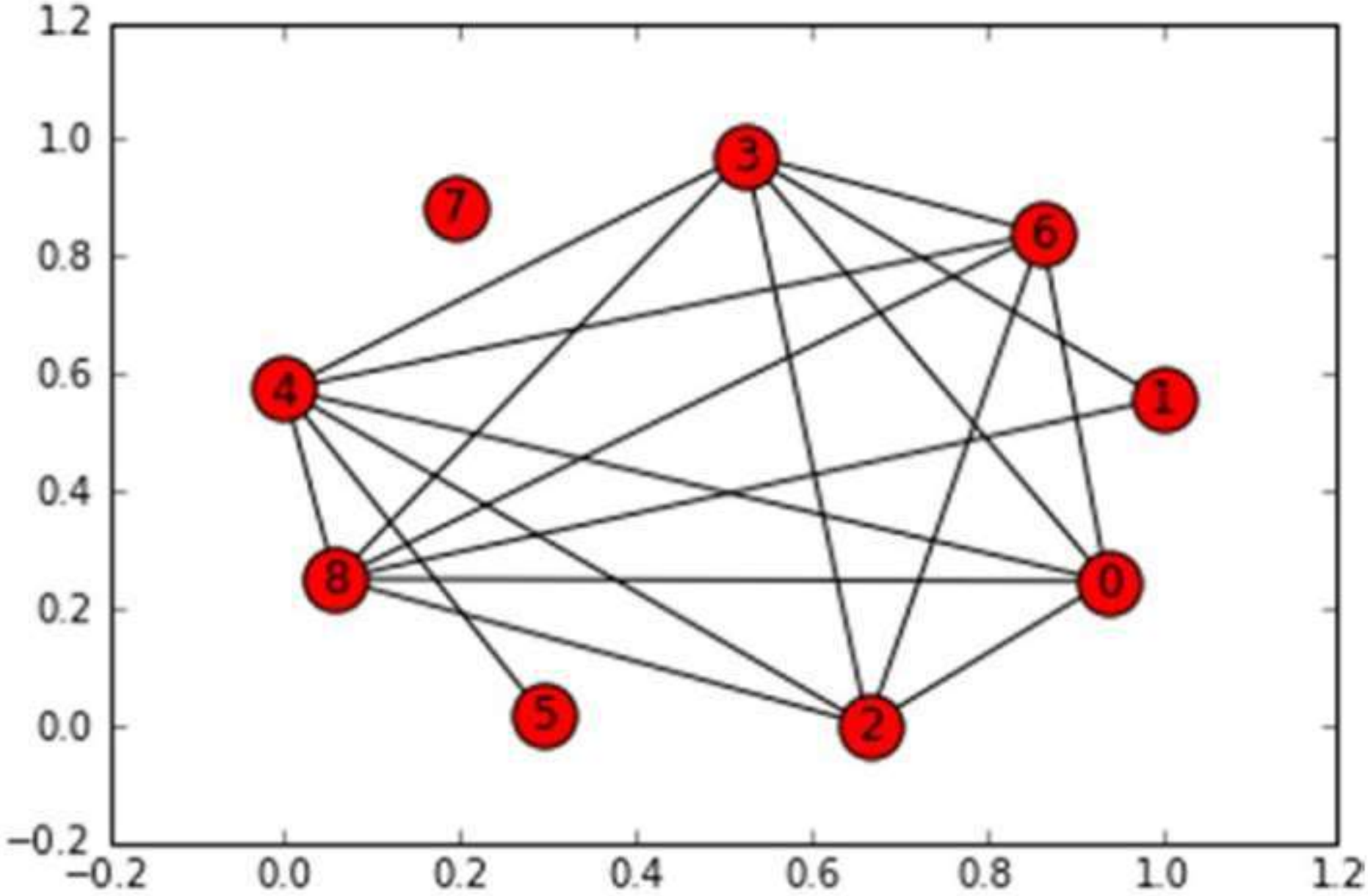


Figure 5-5. *Similarity graph showing connections between sentences*

REVA UNIVERSITY
Bengaluru, India

Summarizing a Product Description

Assignment for students

REVA UNIVERSITY
Bengaluru, India

nirf

THANK YOU

Text summarization techniques

https://www.topcoder.com/thrive/articles/text-summarization-in-nlp

https://aws.amazon.com/blogs/machine-learning/techniques-for-automatic-summarization-of-documents-using-language-models/

https://turbolab.in/feature-extraction-in-natural-language-processing-nlp/

https://www.capitalone.com/tech/machine-learning/understanding-tf-idf/

SVD computation example

Example: Find the SVD of A, $U\Sigma V^T$, where $A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}$.

First we compute the singular values σ_i by finding the eigenvalues of AA^T.

$$AA^T = \begin{pmatrix} 17 & 8 \\ 8 & 17 \end{pmatrix}.$$

The characteristic polynomial is $det(AA^T - \lambda I) = \lambda^2 - 34\lambda + 225 = (\lambda - 25)(\lambda - 9)$, so the singular values are $\sigma_1 = \sqrt{25} = 5$ and $\sigma_2 = \sqrt{9} = 3$.

Now we find the right singular vectors (the columns of V) by finding an orthonormal set of eigenvectors of A^TA. It is also possible to proceed by finding the left singular vectors (columns of U) instead. The eigenvalues of A^TA are 25, 9, and 0, and since A^TA is symmetric we know that the eigenvectors will be orthogonal.

For $\lambda = 25$, we have

$$A^TA - 25I = \begin{pmatrix} -12 & 12 & 2 \\ 12 & -12 & -2 \\ 2 & -2 & -17 \end{pmatrix}$$

which row-reduces to $\begin{pmatrix} 1 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$. A unit-length vector in the kernel of that matrix is $v_1 = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{pmatrix}$.

For $\lambda = 9$ we have $A^TA - 9I = \begin{pmatrix} 4 & 12 & 2 \\ 12 & 4 & -2 \\ 2 & -2 & -1 \end{pmatrix}$ which row-reduces to $\begin{pmatrix} 1 & 0 & -\frac{1}{4} \\ 0 & 1 & \frac{1}{4} \\ 0 & 0 & 0 \end{pmatrix}$.

A unit-length vector in the kernel is $v_2 = \begin{pmatrix} 1/\sqrt{18} \\ -1/\sqrt{18} \\ 4/\sqrt{18} \end{pmatrix}$.

For the last eigenvector, we could compute the kernel of A^TA or find a unit vector perpendicular to v_1 and v_2. To be perpendicular to $v_1 = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ we need $-a = b$. Then the condition that $v_2^Tv_3 = 0$ becomes $2a/\sqrt{18} + 4c/\sqrt{18} = 0$ or $-a = 2c$. So $v_3 = \begin{pmatrix} a \\ -a \\ -a/2 \end{pmatrix}$ and for it to be unit-length we need $a = 2/3$ so $v_3 = \begin{pmatrix} 2/3 \\ -2/3 \\ -1/3 \end{pmatrix}$.

So at this point we know that

$$A = U\Sigma V^T = U \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 1/\sqrt{18} & -1/\sqrt{18} & 4/\sqrt{18} \\ 2/3 & -2/3 & -1/3 \end{pmatrix}.$$

Finally, we can compute U by the formula $\sigma u_i = Av_i$, or $u_i = \frac{1}{\sigma} Av_i$. This gives $U = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$. So in its full glory the SVD is:

$$A = U\Sigma V^T = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 1/\sqrt{18} & -1/\sqrt{18} & 4/\sqrt{18} \\ 2/3 & -2/3 & -1/3 \end{pmatrix}.$$

REVA UNIVERSITY
Bengaluru, India

https://slidesgpt.com/l/u1tL