Text Mining

1. Introduction

- Finding similar documents in high-dimensional space Large document dataset
- Customers who bought similar documents
- Similar news articles in a set of news sites
- Pages with similar words, e.g., for classification by topic.
- NetFlix users with similar tastes in movies for recommendation systems.
- Images of related things
- Find “Somehow” similar documents.

2. Document Representation:

- There are several data structures to represent documents:
    - Bag of Words (BOW)
    - N-gram
    - Term Frequency-Inverse Document Frequency (TF-IDF)
    - K-shingles
    - Word Embeddings

- BOW Representation:
    - Represents text as a collection of words, ignoring grammar and order.
    - Words are represented by their frequencies or as 0/1 (binary) values.
    - Frequency BOW:
        - Each word is represented by its frequency.
        - Generally used in NLP
        - Example:
            - Sentences:

        - D1: "The quick brown fox jumps over the lazy dog."
        - D2: "The brown fox terrier jumps over the lazy cat"
        - Lowercase everything
        - Remove punctuation (none here)
        - Tokenize (split into words)
            - D1: ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
            - D2: ['the', 'brown', 'fox', 'terrier', 'jumps', 'over', 'the', 'lazy', 'cat']
        - Build Vocabulary (unique words)

          ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'lazy', 'dog', 'terrier', 'cat']
        - BOW

          BOW(D1) = [2, 1, 1, 1, 1, 1, 1, 1, 0, 0]

          BOW(D2) = [2, 0, 1, 1, 1, 1, 1, 0, 1, 1]:

- Binary BOW:
    - Denotes whether a word exists or not.
    - Generally used when presence is more important than repetition (e.g., spam classification).
    - Example:
        - Sentences:
            - D1: "The quick brown fox jumps over the lazy dog."

        - D2: "The brown fox terrier jumps over the lazy cat"
        - Lowercase everything
        - Remove punctuation (none here)
        - Tokenize (split into words)
            - D1: ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
            - D2: ['the', 'brown', 'fox', 'terrier', 'jumps', 'over', 'the', 'lazy', 'cat']
        - Build Vocabulary (unique words)

          ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'lazy', 'dog', 'terrier', 'cat']
        - BOW:

          BOW(D1) = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0]

          BOW(D2) = [1, 0, 1, 1, 1, 1, 1, 0, 1, 1]

- TF-IDF Representation
    - Weighs words based on their importance in a document relative to a collection.
    - The weighted scheme of each term in a vector (document) is defined as follows:

      w(tji)= L(tji) . G(tj)   (1)

      where,

      L(tji) is the local weight for term j in document
      G(tj) is the global weight for term j in the dataset.

- The local weights are:
  - No weight (TF): L(tji) = tf(tji)
  - Binary weight: L(tji) = 1, if tf(tji) 1, L(tji) = 0, otherwise
  - Augmented weight: L(tji) = 0.5 + 0.5 * (tf(tji)/tf(max)) where, tf(max) = max{ tf(t1i), tf(t2i), ... , tf(tmi)} and m is the max number of terms in the dataset.
  - Logarithm weight: L(tji) = log (1 + tf(tji))

- The global weights are:
  - No weighting: G(tj) = 1
  - Inverse document frequency (IDF):

    G(tj) = log(N/n(tj)) where, N is the total number of documents in the dataset, and n(tj) is the number of documents that contain term j.

- Normalization
  - Normalize the document Di by its length |Di|.

- TF*IDF:
  - Term Frequency * Inverse Document Frequency

    TF*IDF(tji) = L(tji) * G(tj)

    Where

    L(tji) = tf(tji)

    and G(tj) = log(N/n(tj))

  - It reduces the impact of common words and highlights significant terms.

3. Drawback of Vector Representation:

- It does not catch the semantic information of words:
  - Cherry likes Apple
  - Apple looks like Cherry
-

  Similarity Distance
  - Regular sets:
    - Jaccard Similarity of two sets A and B:
      - It is the cardinality of the union of the two sets divided by their union.

$$\boldsymbol{Sim(A, B) = \frac{A \cap B}{A \cup B}}$$

    - Jaccard Distance:
      - It's the opposite of similarity
      - It measures of difference between the sets

Jaccard Distance=1−Jaccard Similarity

  - Binary Vectors:
    - For binary data, the Jaccard similarity is defined as follows:

$$\boldsymbol{Sim(A, B) = \frac{a}{a + b + c}}$$

Where

a = Number of positions where both vectors have 1 (i.e., 1 in both A and B)

b= Number of positions where A is 1 and B is 0

c= Number of positions where A is 0 and B is 1

    - Jaccard Distance:

Jaccard Distance=1−Jaccard Similarity

- Jaccard Distance Interpretation:

Jaccard Distance	Interpretation
0.0	Perfect match (identical sets)
0.1 – 0.3	High similarity
0.4 – 0.6	Moderate similarity
0.7 – 0.9	Low similarity
1.0	Completely disjoint (no overlap)

4. k-Shingle Representation

- K-Shingle Representation:
- Convert each document into sets of characters/words of length k.
- Similar documents would share more shingles.
- Recording paragraphs would have no implications.
- A k-shingle (or k-gram) is a sequence of k tokens that appear in the document.
- Tokens can be characters or words.
- Pick large k or most of the documents will have most shingles.
- In general, k values of 7-10 are used.
  - k=5 for short documents
  - k=10 for large documents.
- Example: Shingles are n-grams
  - D=[The quick brown fox jumps over the lazy dog]
  - Set of 4-shingles:

    S(D) = {{"The ", "he q", "e qu", " qui", "quic", "uick", "ick ", "ck b", "k br", " bro", "brow", "rown", "own ", "wn f", "n fo", " fox", "fox ", "ox j", "x ju", " jum", "jump", "umps", "mps ", "ps o", "s ov", " ove", "over", "ver ", "er t", "r th", " the", "the ", "he l", "e la", " laz", "lazy", "azy ", "zy d", "y do", " dog"}

- Example: Shingles are words
  - D=[The quick brown fox jumps over the lazy dog]
  - The set of 2-shingles:

    S(D1) = { [The quick], [quick brown], [brown fox], [fox jumps], [jumps over], [over the], [the lazy], [lazy dog]}

- Document Representation:
  - Each document is represented by a 0/1 vector in the space of k-shingles where each shingle is a dimension
  - Vectors are very sparse
  - Documents that share a large number of shingles are similar.
  - Shingle-Document Matrix:
    - Rows = shingles
    - Columns = Documents
    - The matrix is generally very sparse.
  - Similarity between two documents (Columns) is calculated using Jaccard similarity
  - Example:
    - Given the following two sentences:
      - D1: The quick brown fox jumps over the lazy dog.
      - D2: The brown fox terrier jumps over the lazy cat.

- 2-Shingle Representation:

2-Shingle Dimension Space	D1	D2
the quick	1	0
quick brown	1	0
brown fox	1	1
fox jumps	1	0
jumps over	1	1
over the	1	1
the lazy	1	1
lazy dog	1	0
the brown	0	1
fox terrier	0	1
terrier jumps	0	1
lazy cat	0	1

$$\boldsymbol{Sim(A,B) = \frac{4}{12} = \frac{1}{3} = 0.33}$$

- 4-Shingle Representation:

4-Shingle Dimension Space	D1	D2
the quick brown fox	1	0
quick brown fox jumps	1	0
brown fox jumps over	1	0
fox jumps over the	1	0
jumps over the lazy	1	1
over the lazy dog	1	0
the brown fox terrier	0	1
brown fox terrier jumps	0	1
fox terrier jumps over	0	1
terrier jumps over the	0	1
over the lazy cat	0	1

$$\boldsymbol{Sim(A,B) = \frac{1}{11} = 0.09}$$

5. Min-Hashing

- Problem Statement:
  - Given a high dimensional dataset with large data points to compare (Millions or billions)
  - Example: A NxN image ⇒ a vector of N^2 data elements.
  - A distance function between two documents Di and Dj: d(Di,Dj)
- Objective:
  - Find all pairs of points Di and Dj that are similar within a threshold s: d(Di,Dj) <= s
- Solutions:
  - Brute force method:
    - compare all pairs ⇒ $O(N^2)$ where N is the number of data points.
    - Suppose we want to find similarity for every pair of documents in a dataset of N documents:
    - We need N(N-1)/2 about $5*10^{11}$ comparisons.

⇒ We need Minhashing

  - Better solution:
    - Can we do O(N)?

- Min-hashing is also known as Min-wise independent permutations locality sensitive hashing scheme.
- It was introduced by Andrei Broder in 1997.
- It converts large sets into short signatures while preserving similarity.
- "hash" each column C to a small signature h(C), such that:
- h (C) is small enough that we can fit in the main memory.
- Find Similarity using Small Signature:
  - Sim(D1,D2) is the same as the similarity of signatures h(D1) and h(D2)
  - If Sim(D1,D2) is high, then with high probability h(D1) = h(D2)

- If Sim(D1,D2) is low, then with high probability h(D1) is different than h(D2)
- Sim (C1, C2) is the same as the "similarity" of Sig (C1) and Sig (C2).

- Algorithm:
  - Shingling: Convert documents to sets: The set of strings of length k that appear in the document.
  - Min-hashing: Convert large documents to short signatures (Short vectors) while preserving similarities
  - Locality Sensitive Hashing: Generate candidate pairs of signatures that we need for similarity.

- Computing Minhash Values:
  - Imagine the rows of the matrix representation are permuted using a random permutation p.
  - hp(D) = the index of the fist (in the permuted order p) row in which D has a value of 1: hp(D) = minp p(D)
  - Use several (e.g. 100) independent has functions(permutation) to create a signature of a document.
  - Example:
    - Given the following documents:
      D1 = {abc, bcd, cde, efg}
      D2 = {bcd, cde, def, efg}
      D3 = {abc, cde, efg}

    - List of all shingles is:
      Shingles = {abc, bcd, cde, def, efg}

Shingle	Index
abc	0
bcd	1
cde	2
def	3
efg	4

    - Characteristic (shingle-document) matrix:

Shingle	D1	D2	D3
abc	1	0	1
bcd	1	1	0
cde	1	1	1
def	0	1	0
efg	1	1	1

- Define Hash Functions (on row index i):
  - Let's use 3 hash functions:

    $h_1(i) = (i + 1) \% 5$

    $h_2(i) = (3i + 1) \% 5$

    $h_3(i) = (2i + 3) \% 5$
  - Apply each hash function to the shingle row index (0 to 4)
- Create the final signature matrix:
  - For each shingle row i:
    - Compute hash values: $h_1(i)$, $h_2(i)$, ...
    - For each document that has this shingle (i.e., value = 1 in the shingle-document matrix):
      - Look at the current signature value in that row and column
      - Update it if the new hash value is smaller
  - The MinHash signature stores the smallest hash value encountered for each document across all shingles it contains
- Initialize Signature Matrix (3 × 3 filled with ∞):

Hash Fn	D1	D2	D3
h_1	∞	∞	∞
h_2	∞	∞	∞
h_3	∞	∞	∞

- Row 0 — Shingle abc, i = 0
  - Documents D1 and D3 have 1
  - $h_1(0) = 1$, $h_2(0) = 1$, $h_3(0) = 3$
  - Update D1 and D3:

HashFn	D1	D2	D3
h_1	1	∞	1
h_2	1	∞	1
h_3	3	∞	3

- Row 1 — Shingle bcd, i = 1
  - Docs: D1, D2 have 1
  - $h_1 = 2$, $h_2 = 4$, $h_3 = 0$
  - Update D1:
    - $h_1 = \min(1, 2) \rightarrow 1$
    - $h_2 = \min(1, 4) \rightarrow 1$
    - $h_3 = \min(3, 0) \rightarrow 0$
  - Update D2:
    - $h_1 = 2$
    - $h_2 = 4$
    - $h_3 = 0$

HashFn	D1	D2	D3
h_1	1	2	1
h_2	1	4	1
h_3	0	0	3

- Row 2 — Shingle cde, i = 2
  - Docs: D1, D2, D3 have 1
  - $h_1 = 3$, $h_2 = 2$, $h_3 = 2$
  - Update D1:
    - $h_1 = \min(1, 3) \rightarrow 1$
    - $h_2 = \min(1, 2) \rightarrow 1$
    - $h_3 = \min(0, 2) \rightarrow 0$
  - Update D2:
    - $h_1 = \min(2, 3) \rightarrow 2$
    - $h_2 = \min(4, 2) \rightarrow 2$
    - $h_3 = \min(0, 2) \rightarrow 0$

- Update D3:
    - $h_1 = \min(1, 3) \rightarrow 1$
    - $h_2 = \min(1, 2) \rightarrow 1$
    - $h_3 = \min(3, 2) \rightarrow 2$

HashFn	D1	D2	D3
h_1	1	2	1
h_2	1	2	1
h_3	0	0	2

- Row 3 — Shingle def, i = 3
    - Docs: D2 has 1
    - $h_1 = 4$, $h_2 = 0$, $h_3 = 4$
    - Update D2:
        - $h_1 = \min(2, 4) \rightarrow 2$
        - $h_2 = \min(2, 0) \rightarrow$ **0**
        - $h_3 = \min(0, 4) \rightarrow 0$

Hash Values	D1	D2	D3
h_1	1	2	1
h_2	1	0	1
h_3	0	0	2

- Row 4 — Shingle efg, i = 4
    - Docs: D1, D2, D3 have 1
    - $h_1 = 0$, $h_2 = 3$, $h_3 = 1$
    - Update D1:
        - $h_1 = \min(1, 0) \rightarrow 0$
        - $h_2 = \min(1, 3) \rightarrow 1$
        - $h_3 = \min(0, 1) \rightarrow 0$
    - Update D2:
        - $h_1 = \min(2, 0) \rightarrow 0$
        - $h_2 = \min(0, 3) \rightarrow 0$
        - $h_3 = \min(0, 1) \rightarrow 0$

- Update D3:
    - $h_1 = \min(1, 0) \rightarrow 0$
    - $h_2 = \min(1, 3) \rightarrow 1$
    - $h_3 = \min(2, 1) \rightarrow 1$

- Final Signature Matrix:

Hash Fn	D1	D2	D3
h_1	0	0	0
h_2	1	0	1
h_3	0	0	1

6. Locality-Sensitive Hashing (LSH):

- Introduced by Indyk Motwani
- Hashing is generally used for an exact search and tries to avoid or minimize collision.
- Here, we need to make collision happen if two data points are nearly (similar)
- Hash family H is locally sensitive if
    - $Pr[h(x) = h(y)]$ is high if x is close to y
    - $Pr[h(x) = h(y)]$ is low if x is far from y
- Focus on pairs of signatures likely to be from similar documents ⇒ Candidate pairs
- Algorithm:
    - Splitting the signature matrix into bands
    - Hashing each band (group of rows) per document into buckets
    - If two documents land in the same bucket in any band, they are considered candidates for similarity
- Apply LSH to our previous example:
    - Signature Matrix (3 hash functions × 3 documents)

Hash Values	D1	D2	D3
h_1	0	0	0
h_2	1	0	1
h_3	0	0	1

- Step 1: Choose bands and rows per band
  - We have 3 rows. A common LSH choice is:
    - b = 3 bands
    - r = 1 row per band
  - Each band will be a single row of the signature matrix.
    - Band 1 = row 0 (h_1)
    - Band 2 = row 1 (h_2)
    - Band 3 = row 2 (h_3)
- Step 2: Hash documents into buckets band by band
  - Each band's row is considered as a "signature slice" for that document and use it as a key.
  - Band 1 (row = h_1):

Doc	Value
D1	0
D2	0
D3	0

    - All 3 have the same value → go to the same **bucket**
      - Bucket B1: [D1, D2, D3]

  - Band 2 (row = h_2):

Doc	Value
D1	1
D2	0
D3	1

    - D1 and D3 go to bucket 1
    - D2 goes to bucket 0
    - Buckets:
      - B2a: [D1, D3]
      - B2b: [D2]

- Band 3 (row = h_3):

Doc	Value
D1	0
D2	0
D3	1

  - D1 and D2 → bucket 0
  - D3 → bucket 1
  - Buckets:
    - B3a: [D1, D2]
    - B3b: [D3]

- Step 3: Candidate Pairs
  - If two docs appear in the same bucket, they are candidates for similarity.
  - Pairs:
    - D1 & D2 → Same in Band 1 and Band 3
    - D1 & D3 → Same in Band 1 and Band 2
    - D2 & D3 → Same in Band 1

Pair	Candidate?	Band(s)
D1 & D2	Yes	1, 3
D1 & D3	Yes	1, 2
D2 & D3	Yes	1

- Compute Similarity:
  - Compute actual or estimated Jaccard similarity only for these pairs, saving time in large-scale data.
  - Example: Original Documents (Shingles):
    - D1 = {abc, bcd, cde, efg}
    - D2 = {bcd, cde, def, efg}
    - D3 = {abc, cde, efg}
  - Actual Jaccard Similarity:
    - J(D1, D2) = 3 / 5 = 0.60

- J(D1, D3) = 3 / 4 = 0.75
- J(D2, D3) = 2 / 5 = 0.40

- Estimated Jaccard Similarity (MinHash Signature Matrix):

Hash valued	D1	D2	D3
h_1	0	0	0
h_2	1	0	1
h_3	0	0	1

  - Count matches between signature columns:
    - Here we don't ignore the 0's as in the exact Jaccard similarity because 0 is just a **hash value**, not a Boolean absence.
    - D1 vs D2: Matches in h_1 and h_3 → 2/3 = 0.67
    - D1 vs D3: Matches in h_1 and h_2 → 2/3 = 0.67
    - D2 vs D3: Matches in h_1 only → 1/3 = 0.33

- Comparison:

Pair	Estimated Jaccard	Actual Jaccard
D1 vs D2	0.67	0.60
D1 vs D3	0.67	0.75
D2 vs D3	0.33	0.40

- Hamming Similarity:
  - Consider our initial shingle-document matrix:

Shingle	D1	D2	D3
abc	1	0	1
bcd	1	1	0
cde	1	1	1
def	0	1	0
efg	1	1	1

- The bit vectors are:

  D1 = [1, 1, 1, 0, 1]

  D2 = [0, 1, 1, 1, 1]

  D3 = [1, 0, 1, 0, 1]

Pair	Hamming Similarity
D1 vs D2	3/5=0.6
D1 vs D3	4/5=0.8
D2 vs D3	2/5= 0.4

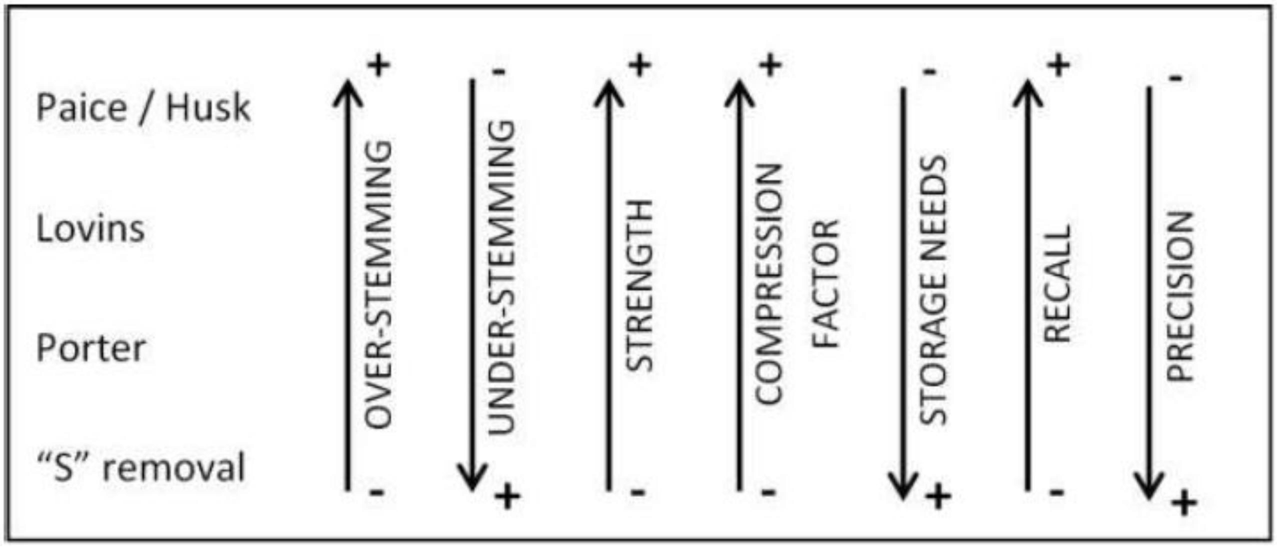
- Comparison:

Pair	Estimated Jaccard	Actual Jaccard	Hamming Similarity
D1 vs D2	0.67	0.60	0.6
D1 vs D3	0.67	0.75	0.8
D2 vs D3	0.33	0.40	0.4

7. Stemming Algorithms

- Used to improve retrieval effectiveness and to reduce the size of indexing files.
- Stemmers attempt to reduce morphological variations of words to a common stem – usually involves removing suffixes
- Can be done at pre-processing step.
- Definition
  - Many morphological variations of words
    - Inflectional (plurals, tenses)
    - Derivational (making verbs nouns etc.)
  - Stemmers attempt to reduce morphological variations of words to a common stem – usually involves removing suffixes.

- Stemming is the process of reducing inflected words to their word stem.
- A Stem is a base or root form of a work
- Example:
  - Networks, Networking, Networked ➔ Network
- Porter algorithm:
  - The Porter algorithm consists of a set of condition/action rules.
  - Porter's stemmer is heuristic, in that it is a practical method not guaranteed to be optimal
  - The condition fall into three classes
    - Conditions on the stem
    - Conditions on the suffix
    - Conditions on rules
  - Stemming is the determination of the stem of a given word
    - Word = Stem + Affix(es)
    - E.g., generalizations = general + ization + s
  - Porter's stemmer is a rule-based algorithm
    - E.g., ational → ate (apply: relational → relate)
  - Affix removal algorithms:
    - They remove suffixes and/or prefixes from terms leaving a stem
    - If a word ends in "ies" but not "eies" or "aies " (Harman 1991) Then "ies" -> "y"
    - If a word ends in "es" but not "aes" , or "ees " or "oes" Then "es" -> "e"
    - If a word ends in "s" but not "us" or "ss " Then "s" -> "NULL"
  - Additional rules…
  - https://www2.seas.gwu.edu/~bell/csci243/lectures/stemming_algorithm_preprocessing.pdf
- A survey of stemming algorithms in information retrieval By Cristian Moral et al.



8. Lemmatization:

- It is the process of reducing the inflectional/variant forms of a word to its base or dictionary form (lemma) while ensuring that it remains a valid word in the language.
- Lemmatization looks at surrounding text to determine a given word's part of speech.
- Unlike stemming, which just chops off suffixes, lemmatization considers the word's meaning and context.
- Example:
  - am, are, is ➔ be
  - car, cars, car's, cars' ➔ car
  - the boy's cars are different colors ➔ the boy car be different color
  - It is better if you are leaving me alone

Word	POS	Lemma
It	(pronoun)	It
is	(verb)	be

better	(adjective)	good
if	(conjunction)	if
you	(pronoun)	you
are	(verb)	be
leaving	(verb)	leave
me	(pronoun)	me
alone	(adverb)	alone

- Lemmatization implies doing “proper” reduction to dictionary headword form.
- Algorithm:
    - Tokenization:
        - Split the input text into individual words (tokens).
    - POS Tagging:
        - Assign Part of Speech (POS) tags to each word (e.g., noun, verb, adjective).
    - Lookup in a Lexical Database (WordNet or Dictionary):
        - Identify the lemma (root form) of each word based on its POS.
    - Apply Rules for Irregular Words:
        - Some words do not follow standard suffix rules (e.g., "better" → "good", "mice" → "mouse").
    - Return the Lemmatized Sentence:
        - Replace each word with its lemma and return the final text.

- Lemmatization advantages and disadvantages
    - Advantages:
        - Accuracy: It accurately determines the lemma of a word.
        - Understanding text: It helps NLP tools, such as AI chatbots, understand full-sentence input from end users.
        - Contextual understanding: It helps determine the context of the word and its part of speech.

        - Dimensionality reduction.
    - Disadvantages:
        - Computational overhead.
        - Slower processing speed. Lemmatization algorithms are slower than stemming algorithms due to the morphological analysis.
        - Language dependency. Some languages are more complex to analyze.