1 Stemming of Words

Goal: Reduce words to their root form (stems) by removing suffixes.

Algorithm: Porter's Stemming Algorithm

Concept: Applies a series of condition-based rules sequentially to strip suffixes.
Notation:

- `C`: Consonant
- `V`: Vowel
- `m`: The "measure" or count of `(VC)` sequences in the stem.
- **Format:** `[C](VC){m}[V]`

Rules & Steps

1. **SS** $\rightarrow$ **SS:** (e.g., *caress* $\rightarrow$ *caress*, *boss* $\rightarrow$ *boss*).
2. **IES** $\rightarrow$ **I:** (e.g., *ponies* $\rightarrow$ *poni*).
3. **SSES** $\rightarrow$ **SS:** (e.g., *caresses* $\rightarrow$ *caress*).
4. **S** $\rightarrow \emptyset$**:** (e.g., *cats* $\rightarrow$ *cat*).
5. **ING** $\rightarrow \emptyset$**:** Condition: Stem must contain a vowel. (e.g., *motoring* $\rightarrow$ *motor*).
6. **y** $\rightarrow$ **i:** Condition: Stem ends in C. (e.g., *happy* $\rightarrow$ *happi*).

Step-by-Step Example

Word: `Replacement`

1. **Identify Suffix:** `EMENT`
2. **Check Condition:** Does the remaining stem `REPLAC` have $m > 1$?
   - Sequence: R(C) E(V) P(C) L(C) A(V) C(C).
   - m count for `VC` groups is 2. Condition met.
3. **Apply Rule:** Remove `EMENT`.
4. **Result:** `REPLAC`

2 Distance Metrics

Goal: Measure the difference/similarity between two strings.

A. Hamming Distance

- **Definition:** Number of positions at which corresponding symbols are different.
- **Constraint:** Strings must be of **equal length**.
- **Step-by-Step Calculation:**
  1. Align strings.
  2. Compare character at index i of String A with index i of String B.
  3. Count mismatches.
- **Example:**
  - Str1: `11001`
  - Str2: `10110`
  - Mismatch at positions 2, 3, 4, 5. **Distance = 4**.

B. Levenshtein Distance

- **Definition:** Minimum number of single-character edits (insertions, deletions, substitutions) to transform String A to String B.
- **Step-by-Step Calculation:**
  1. Compare strings.
  2. Find minimal operations to match them.
- **Example (Lecture 2):** *kitten* $\rightarrow$ *sitting*
  1. k $\rightarrow$ s (Substitution) $\rightarrow$ *sitten*
  2. e $\rightarrow$ i (Substitution) $\rightarrow$ *sittin*
  3. Insert g at end (Insertion) $\rightarrow$ *sitting*
- **Distance = 3**.

3 Calculations (TF-IDF, Vectors, Matrices)

A. Bag of Words (BoW) / Count Vectoriser

- **Concept:** Represents text as a fixed-length vector of word counts.
- **Step-by-Step:**
  1. Build Vocabulary from all docs.
  2. For each doc, count occurrences of each vocab word.
  3. Create vector $[c_1, c_2, \ldots, c_n]$.
- **Example:**
  - **Doc 1:** "Blue sky" | **Doc 2:** "Blue sea"

  - **Vocabulary:** [Blue, Sea, Sky]
  - **Vector Doc 1:** [1, 0, 1] (1 Blue, 0 Sea, 1 Sky)
  - **Vector Doc 2:** [1, 1, 0] (1 Blue, 1 Sea, 0 Sky)

B. Term-Document Matrix

- **Concept:** A matrix where rows represent terms and columns represent documents.
- **Values:** Can be binary (1/0) or Counts.
- **Example:**
  - Doc 1: "Antony Brutus"
  - Doc 2: "Antony Caesar"
  - Matrix (Binary):
    * Antony: [1, 1]
    * Brutus: [1, 0]
    * Caesar: [0, 1]

C. TF-IDF Calculation

Formulas (Lecture 3):

1. **TF (Term Frequency):**
$$tf_{t,d} = \frac{\text{count of } t \text{ in } d}{\text{total terms in } d}$$
2. **IDF (Inverse Document Frequency):**
$$idf_t = \log_{10}\left(\frac{N}{df_t}\right)$$
where N is total docs and df_t is number of docs containing t.
3. **Weight:**
$$W_{t,d} = tf_{t,d} \times idf_t$$

Example Calculation:

- **Corpus:** D1: "apple banana", D2: "apple cherry". ($N = 2$)
- **Target:** Calculate weight of "banana" in D1.
- **Step 1 (TF):** "banana" appears 1 time. Total words in D1 is 2.
$$tf = 1/2 = 0.5$$
- **Step 2 (IDF):** "banana" appears in 1 document (D1).
$$idf = \log_{10}(2/1) = \log_{10}(2) \approx 0.301$$
- **Step 3 (Weight):**
$$W = 0.5 \times 0.301 = 0.1505$$

D. Cosine Similarity

- **Formula:**

$$\text{Cosine}(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum(q_i d_i)}{\sqrt{\sum q_i^2}\sqrt{\sum d_i^2}}$$

- **Step-by-Step:**
  1. Compute Dot Product (numerator).
  2. Compute Euclidean Norm (length) of vectors (denominator).
  3. Divide.
- **Example:**
  - Vector A: [1, 2] | Vector B: [3, 4]
  - **Dot Product:** $(1 \times 3) + (2 \times 4) = 3 + 8 = 11$
  - **Lengths:**

$$|A| = \sqrt{1^2 + 2^2} = \sqrt{5} \approx 2.236$$
$$|B| = \sqrt{3^2 + 4^2} = \sqrt{25} = 5$$

  - **Result:** $11/(2.236 \times 5) = 11/11.18 \approx \mathbf{0.98}$

E. Jaccard Similarity

- **Formula:**

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

- **Step-by-Step:**
  1. Treat text as Sets (remove duplicates).
  2. Count Intersection size.
  3. Count Union size.
  4. Divide.
- **Example:**
  - Doc A: "apple banana" $\rightarrow$ {apple, banana}
  - Doc B: "banana cherry" $\rightarrow$ {banana, cherry}
  - **Intersection:** {banana} (Size = 1)
  - **Union:** {apple, banana, cherry} (Size = 3)
  - **Result:** $1/3 \approx \mathbf{0.33}$

4 Advantages and Disadvantages of Models

A. Jaccard Similarity

- **Advantages:**
  - Simple to compute for sets (binary presence).
  - Useful when duplicates don't matter.
- **Disadvantages (Lecture 3):**
  - **Ignores Term Frequency:** "The" appearing 10 times is treated same as appearing once.
  - **Ignores Rare Terms:** Doesn't weight informative words; stop words affect score equally.
  - **Ignores Order:** "Dog bites man" = "Man bites dog".

B. Bag of Words (BoW) / Count Vectors

- **Advantages:**
  - Simple fixed-length representation for ML algorithms.
- **Disadvantages (Lecture 3):**
  - **Loss of Order:** Structural info lost (e.g., "John runs faster than Mary" vs "Mary runs faster than John" = same vector).
  - **Loss of Semantics:** Synonyms (car, auto) are orthogonal.
  - **Frequency Bias:** Frequent stop words dominate vector magnitude.

C. Term-Document Matrix

- **Disadvantages (Lecture 4):**
  - **Sparsity:** Most entries are 0.
  - **Storage:** Size is $O(V \times D)$. For Wikipedia (6.89M docs, 1.7M terms), size is $\approx$ 46TB. Impractical to store fully.

D. TF-IDF

- **Advantages:**
  - **Handles Stop Words:** Down-weights common words (low IDF).
  - **Relevance:** Up-weights rare, informative terms (high IDF).
- **Disadvantages:**
  - Still ignores word order.
  - High dimensionality ($|V|$).

E. Cosine Similarity vs. Euclidean

- **Euclidean Disadvantage (Lecture 3):** Sensitive to document length. Long docs are "far" from short docs even if topics match.
- **Cosine Advantage:** Measures angle, normalizing length. Makes short queries comparable to long documents.

5 Inverted Index & Merge Algorithm

A. Inverted Index Construction

Step-by-Step:

1. **Tokenize** documents.
2. **Preprocess** (Stemming, Stop words).
3. **Create Pairs:** `(Term, DocID)`.
4. **Sort:** Alphabetically by Term, then by DocID.
5. **Group:** Create Postings List (List of DocIDs) for each unique Term.

B. Merge Algorithm (for AND queries)

Task: Find intersection of two sorted lists L_1, L_2.
Step-by-Step:

1. Initialize pointers `p1`, `p2` at heads of lists.
2. While both valid:
   - If `docID(p1) == docID(p2)`: Add to result, advance both.
   - If `docID(p1) < docID(p2)`: Advance `p1`.
   - Else: Advance `p2`.

Complexity: $O(x + y)$ (Linear).

C. Complete Example

Building the Inverted Index

Given Documents:

- Doc 1: "The cat sat on the mat"
- Doc 2: "The dog sat on the log"
- Doc 3: "Cats and dogs are animals"

Step 1 – Tokenize:

- Doc 1: [The, cat, sat, on, the, mat]
- Doc 2: [The, dog, sat, on, the, log]

- Doc 3: [Cats, and, dogs, are, animals]

Step 2 – Preprocess (lowercase, stem, remove stop words):

- Doc 1: [cat, sat, mat]
- Doc 2: [dog, sat, log]
- Doc 3: [cat, dog, animal]

Step 3 – Create (Term, DocID) pairs:

- (cat, 1), (sat, 1), (mat, 1), (dog, 2), (sat, 2), (log, 2), (cat, 3), (dog, 3), (animal, 3)

Step 4 – Sort alphabetically by Term, then by DocID:

- (animal, 3), (cat, 1), (cat, 3), (dog, 2), (dog, 3), (log, 2), (mat, 1), (sat, 1), (sat, 2)

Step 5 – Group into Postings Lists:

- animal: [3]
- cat: [1, 3]
- dog: [2, 3]
- log: [2]
- mat: [1]
- sat: [1, 2]

Merge Algorithm Example 1

Query: "cat AND dog" (Find documents containing both terms)
Postings Lists: cat: [1, 3], dog: [2, 3]

Execution:

1. **Initialize:** p1 $\rightarrow$ 1 (cat), p2 $\rightarrow$ 2 (dog), Result = []
2. **Compare:** $1 < 2 \rightarrow$ Advance p1 to 3
3. **Compare:** $3 > 2 \rightarrow$ Advance p2 to 3
4. **Compare:** $3 == 3 \rightarrow$ Add 3 to result, advance both (reach end)
5. **Result:** [3]

Answer: Document 3 contains both "cat" and "dog".

Merge Algorithm Example 2

Query: "cat AND sat"
Postings Lists: cat: [1, 3], sat: [1, 2]

Execution:

1. **Initialize:** p1 $\rightarrow$ 1, p2 $\rightarrow$ 1, Result = []
2. **Compare:** $1 == 1 \rightarrow$ Add 1 to result, advance both (p1 $\rightarrow$ 3, p2 $\rightarrow$ 2)
3. **Compare:** $3 > 2 \rightarrow$ Advance p2 (p2 reaches end)

4. **Result:** [1]

Answer: Document 1 contains both "cat" and "sat".

6 N-grams, Maximum Likelihood, Probability of Sentences

A. N-Grams

Definition: A contiguous sequence of N items (words) from a text (Lecture 4).

- **Unigram ($N = 1$):** Words are independent.
- **Bigram ($N = 2$):** Word depends on 1 previous word.
- **Trigram ($N = 3$):** Word depends on 2 previous words.

Use Case: Predicting next word, Spell check, Speech recognition.

B. Maximum Likelihood Estimation (MLE)

Goal: Estimate probability parameters from count data (Lecture 5).
Formula (Bigram):

$$P(w_i|w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})}$$

Step-by-Step:

1. Count occurrences of the bigram pair (w_{i-1}, w_i) in corpus.
2. Count occurrences of the prefix word (w_{i-1}).
3. Divide.

Example:

- **Corpus:** "I want food. I want water."
- **Goal:** $P(\text{food}|\text{want})$
- Count(want, food) = 1.
- Count(want) = 2.
- **Result:** $1/2 = 0.5$.

C. Probability of a Sentence

Goal: Compute $P(S) = P(w_1, w_2, \ldots, w_n)$.

1. Chain Rule (Theoretical):

$$P(S) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_1, w_2) \ldots$$

2. Markov Assumption (Approximation):
We assume limited history (e.g., Bigram: history is just previous word).

$$P(w_i|w_1 \ldots w_{i-1}) \approx P(w_i|w_{i-1})$$

3. Final Formula (Bigram Model):

$$P(S) \approx \prod_{i=1}^{n} P(w_i|w_{i-1})$$

(Note: Use `<s>` *for start and* `</s>` *for end).*

Log Space Calculation (to prevent underflow):

$$\log P(S) \approx \sum_{i=1}^{n} \log P(w_i|w_{i-1})$$

7 Understanding of Language (Probabilities)

(This point refers to interpreting given probabilities like $P(\text{english}|\text{want}) = .0011$. This implies using the Chain Rule or Bigram models discussed in Point 6 to calculate sentence probability based on provided conditional probabilities).

8 Smoothing on Bi-grams

Algorithm: Laplace (Add-One) Smoothing

Problem: MLE assigns zero probability to unseen bigrams, making the whole sentence probability zero.
Formula:

$$P_{\text{Laplace}}(w_i|w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i) + 1}{\text{Count}(w_{i-1}) + V}$$

where V is the Size of Vocabulary (unique words).

Step-by-Step:

1. Calculate Vocabulary Size V.
2. Get raw bigram count $C(w_{i-1}, w_i)$.
3. Get raw unigram count $C(w_{i-1})$.
4. Apply formula: Add 1 to numerator, add V to denominator.

Example:

- **Corpus:** "I want food." ($V = 3$)
- **Goal:** Probability of unseen bigram "want water".
- Raw Count(want, water) = 0. Raw Count(want) = 1.
- **Calculation:** $(0 + 1)/(1 + 3) = 1/4 = 0.25$.

9 Compute Perplexity

Definition: A metric to evaluate Language Models. Inverse probability of test set normalized by number of words.
Formula:

$$PP(W) = P(w_1 \dots w_N)^{-1/N} = \sqrt[N]{\frac{1}{P(w_1 \dots w_N)}}$$

Interpretation: Lower perplexity is better.

10 Neural Network Models (Logic Gates)

- **AND / OR Gates:** Can be solved by a **Perceptron** (Single neural unit, linear classifier).
- **XOR Problem:**
  - XOR is **not linearly separable** (cannot draw a straight line to separate outputs).
  - **Solution:** Requires a **Multi-Layer Perceptron (MLP)** or Neural Network with a hidden layer.

11 Computation Graphs & Backward Passes

A. Computation Graph

Concept: Representing a function as a graph of operations.
Example: $L = c(a + 2b)$

- $d = 2b$
- $e = a + d$
- $L = c \times e$

B. Backward Pass (Backpropagation)

Goal: Calculate gradients ($\frac{\partial L}{\partial w}$) for updates.
Method: Chain Rule.
Step-by-Step (from L backwards):

1. $\frac{\partial L}{\partial L} = 1$
2. $\frac{\partial L}{\partial c} = e$
3. $\frac{\partial L}{\partial e} = c$
4. $\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial a} = c \cdot 1 = c$
5. $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = c \cdot 1 \cdot 2 = 2c$

12 Word2Vec

A. Conceptual Questions

- **Document Embeddings (Mean Pooling):** To generate a document embedding from word embeddings, you can calculate the **mean (average)** of all word vectors in the document.
  - **Dimension:** The dimension of the document embedding will be the **same** as the dimension of the word embeddings (e.g., if word vectors are size 300, the mean document vector is size 300).
- **Synonyms:** Words with similar meanings have vectors that are close (high cosine similarity). Word2vec captures this by learning from context (distributional hypothesis).
- **Evolution of Meaning:** By training embeddings on corpora from different time periods, you can track how a word's vector shifts (e.g., "gay" shifting from "cheerful" to "homosexual").
- **Polysemy (Multiple Meanings):** Standard Word2vec assigns **one vector per word**, so "bank" (river) and "bank" (finance) are averaged into one vector. This is a limitation.

B. Steps of the Model (Skip-gram)

Goal: Learn embeddings by predicting context words (c) given a target word (t).
Process:

1. **Input:** One-hot vector of target word t.
2. **Projection:** Multiply by weight matrix W (Target Embeddings) to get hidden layer vector h (the embedding of t).
3. **Output:** Multiply h by context matrix C to get scores for all vocabulary words.
4. **Softmax:** Convert scores to probabilities.
5. **Loss:** Maximize probability of actual context words.

Conceptual Example:

- **Sentence:** "The cat sat". Target=cat, Context=sat.
- **Input:** One-hot vector for cat (e.g., [0, 1, 0...]).
- **Forward:** The model outputs probabilities for every word in vocab. Ideally, $P(sat|cat)$ should be high.
- **Update:** If the model predicts "dog" instead of "sat", Backprop adjusts the vector of "cat" to be closer to "sat".

C. Negative Sampling

- **Why used?** Computing Softmax over the entire vocabulary (denominator $\sum e^z$) is computationally expensive ($O(V)$).
- **Method:** Instead of multiclass classification, train a binary classifier (Logistic Regression).
  - **Positive pairs:** (t, c_{real}) from the text. Label = 1.

  - **Negative pairs:** (t, c_{noise}) randomly sampled. Label = 0.
- **Objective:** Maximize $P(+|t, c_{\text{real}}) + \sum P(-|t, c_{\text{noise}})$.
- **Trade-off:**
  - More negative samples (k) = Better accuracy for small data, but slower training.
  - Fewer negative samples = Faster training, sufficient for large data (5–20 for small, 2–5 for large datasets).
- **Rare Words:** Noise words are sampled based on unigram frequency raised to power 0.75 ($P(w)^{0.75}$). This increases the probability of sampling **rare words** as negatives, ensuring they are represented in the learning process.

D. Comparisons (GloVe, FastText)

- **Word2Vec:** Predictive model. Local context window. Fails on OOV (Out of Vocabulary) words.
- **GloVe (Global Vectors):** Count-based model. Uses **Global Co-occurrence Matrix** to capture global statistics. Factorizes the log-count matrix.
- **FastText:** Extension of Word2vec. Represents words as bags of **character n-grams** (sub-words).
  - **Advantage:** Can handle OOV words by summing vectors of their sub-word n-grams. Handles morphology well.

13 BERT

A. Concept & Architecture

- **BERT (Bidirectional Encoder Representations from Transformers):**
- **Mechanism:** Uses the **Transformer Encoder** stack.
- **Key Innovation:** It is **Bidirectional** (looks at both left and right context simultaneously) using the "Masked Language Model" (MLM) objective.

B. Main Difference from Word2Vec

- **Word2Vec (Static):** Generates a **static** embedding. "Bank" has the exact same vector in "river bank" and "bank account".
- **BERT (Contextual):** Generates **contextual** embeddings. The vector for "bank" changes dynamically based on the surrounding words in the specific sentence.

C. Attention Mechanisms

- **Self-Attention:** Allows each word in a sequence to attend to all other words to compute a representation.

- **Scaled Dot-Product Attention:**

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

  - **Scaling Factor ($\sqrt{d_k}$):** The dot product $Q \cdot K^T$ can become very large for large dimensions (d_k). Large values push the Softmax function into regions with extremely small gradients (vanishing gradients). Dividing by $\sqrt{d_k}$ scales the values back down to a range where gradients are stable.

- **Multi-Head Attention:** Runs attention h times in parallel with different linear projections. Allows the model to focus on different types of relationships (e.g., one head focuses on syntax, another on semantics).

14 Graph Analytics

A. Definitions & Computation

- **Graph (G):** A pair (V, E) where V are nodes and E are edges.
- **Degree:**
  - **Undirected:** Number of edges connected to a node.
  - **Directed: In-degree** (edges entering) + **Out-degree** (edges leaving).
- **Path:** A sequence of distinct nodes/edges connecting two nodes.
- **Adjacency Matrix (A):** A $|V| \times |V|$ matrix.
  - $A_{ij} = 1$ if there is an edge between i and j.
  - $A_{ij} = 0$ otherwise.
  - For weighted graphs, $A_{ij} = w_{ij}$ (weight).
- **Example:**
  - Nodes: 1, 2, 3. Edges: 1-2, 2-3.
  - Matrix (Rows are node 1, 2, 3):

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

- **Connected Component:** A subgraph where every node is reachable from every other node.

15 Graph Embeddings

A. Node2Vec

- **Goal:** Learn continuous feature representations (embeddings) for nodes in a network.
- **Intuition:** Map nodes to vectors such that nodes that share a "neighborhood" in the graph are close in the vector space.

B. Random Walks

- **Importance:** Random walks are used to generate "sentences" of nodes. A random walk starting at node u generates a sequence of nodes $u, v_1, v_2 \ldots$. These sequences are treated as text sentences and fed into **Skip-gram** (Word2Vec) to learn embeddings.

C. Parameters p and q (Biased Random Walks)

Node2vec introduces parameters to bias the random walk, trading off between **BFS** (Breadth-First Search) and **DFS** (Depth-First Search) behaviors.

- **Return parameter (p):** Controls the likelihood of immediately revisiting a node in the walk.
  - **Low p:** High probability of returning to the previous node. Encourages local exploration (**BFS-like** behavior).
  - **High p:** Less likely to return.
- **In-out parameter (q):** Controls the likelihood of moving "outward" to nodes further away.
  - **Low q:** High probability of moving to nodes far from the source. Encourages global exploration (**DFS-like** behavior).
  - **High q:** Discourages moving outward, stays local.

D. Loss Functions

- **Objective:** Maximize the likelihood of preserving network neighborhoods.
- **Loss:** Similar to Word2Vec Skip-gram with Negative Sampling.
  - Maximize dot product of node u and its neighbor v (from random walk).
  - Minimize dot product of node u and random negative nodes n_{neg}.