Unit 3

Language Modelling

Contents

- Probabilistic language modelling
- Markov models
- Generative models of language
- Log Liner Models
- Graph-based Models N-gram models: Simple n-gram models
- Estimation parameters and smoothing
- Evaluating language models
- Word Embeddings/ Vector Semantics: Bag-of-words, TFIDF, word2vec, doc2vec
- Contextualized representations (BERT) Topic Modelling: Latent Dirichlet Allocation (LDA), Latent Semantic Analysis, Non Negative Matrix Factorization

Generative models of language

- Generative models of language **are AI models that generate human-like text** by predicting and producing sequences of words based on patterns learned from large datasets.
- These models use **probabilistic techniques, deep learning, and neural networks** to create meaningful and contextually relevant text.

Some key types of generative language models include:

- **n-gram Models** – Use statistical probabilities to predict the next word based on previous words in a sequence.
- **Hidden Markov Models (HMMs)** – Utilize probabilistic state transitions to generate text sequences.

- **Recurrent Neural Networks (RNNs)** – Process sequential data and maintain contextual dependencies.

- **Long Short-Term Memory (LSTM) Networks** – A type of RNN that mitigates long-term dependency issues.

- **Transformers (e.g., GPT, BERT, T5)** – Use self-attention mechanisms to model long-range dependencies and generate high-quality text.

- **Variational Autoencoders (VAEs)** – Learn probabilistic latent representations to generate diverse text.

- **Generative Adversarial Networks (GANs) for Text** – Use a generator-discriminator framework to produce realistic text.

Applications:

- Chatbots and Virtual Assistants (Domain Specific)
- Machine Translation (e.g., Google Translate)

- Text Summarization (ChatGPT, BERTSUM (Google), T5, Pegasus (Google), Hugging Face Transformers, SummarizeBot, etc
- Content Generation (e.g., blog writing, code generation- ChatGPT, Jasper AI, Copy.ai, Writesonic, Rytr)

- Poetry and Creative Writing (ChatGPT, inferkit, Verse by verse (Google), etc)
- Personalized Recommendations (Netflix's Context-Aware Model, Microsoft Azure Personalizer, Amazon Personalize, etc)

Log Linear Model

- A Log-Linear Model (LLM) in Natural Language Processing (NLP) is a **type of probabilistic model** used for tasks like **text classification, part-of-speech tagging, and named entity recognition.**

- It generalizes logistic regression by modeling the probability of an output class given an input using an exponential function of a weighted feature set.

- **Log-Linear Model (LLM) implementation** for **text classification** using Python and **Scikit-learn**. We'll use the **Logistic Regression** classifier, which is a basic log-linear model.

Implementation: Sentiment Classification using a Log-Linear Model

We'll classify movie reviews as **positive** or **negative** using the IMDb dataset.

Untitled9.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

```python
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Expanded dataset
data = {
    "text": [
        "I loved the movie! It was fantastic.",
        "The film was terrible and boring.",
        "Amazing plot and great acting!",
        "Worst movie I have ever seen.",
        "I enjoyed every moment of the film.",
        "It was a complete waste of time.",
        "Brilliant storytelling and great performances.",
        "Awful experience, the plot made no sense.",
        "Fantastic visuals and stunning cinematography.",
        "The movie was dull and predictable.",
        "Best movie of the year! Highly recommend.",
        "Disappointed. I expected a lot more."
    ],
    "label": [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]  # 1 = Positive, 0 = Negative
}

df = pd.DataFrame(data)
```

```python
# TF-IDF Vectorization
vectorizer = TfidfVectorizer(stop_words="english")
X = vectorizer.fit_transform(df["text"])
y = df["label"]

# Train-Test Split (Ensure stratification)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=42)

# Train Log-Linear Model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predictions and Accuracy
y_pred = model.predict(X_test)
train_acc = model.score(X_train, y_train)
test_acc = accuracy_score(y_test, y_pred)

# Print Results
print(f"Training Accuracy: {train_acc:.2f}")
print(f"Testing Accuracy: {test_acc:.2f}")

# Predict New Reviews
new_reviews = ["The movie was absolutely fantastic!", "I hated every second of it."]
new_X = vectorizer.transform(new_reviews)
predictions = model.predict(new_X)

for review, pred in zip(new_reviews, predictions):
    sentiment = "Positive" if pred == 1 else "Negative"
    print(f"Review: '{review}' -> Sentiment: {sentiment}")
```

```
Training Accuracy: 1.00
Testing Accuracy: 0.50
Review: 'The movie was absolutely fantastic!' -> Sentiment: Positive
Review: 'I hated every second of it.' -> Sentiment: Negative
```

Why is Testing Accuracy Low?

1. **Overfitting**
    - Training accuracy is **1.00 (100%)**, meaning the model memorized the small dataset.
    - Test accuracy is **0.50 (50%)**, meaning predictions on new data are only slightly better than random guessing.
2. **Small Dataset**
    - Only **12 samples** are not enough for a robust model.
    - **Solution:** Use a **larger dataset** (e.g., IMDb movie reviews).
3. **Feature Sparsity in TF-IDF**
    - **Solution:** Use **n-grams (bigrams/trigrams)** to improve context learning.

What is TF-IDF?

TF-IDF is a technique used to **convert text into numerical values** while giving **importance to words** that are **frequent in a document but rare across all documents**. It helps in reducing the weight of commonly used words (like "the", "is", "and") and increasing the weight of important words.

How does TfidfVectorizer work?

1. **Tokenization** → Splits text into words.
2. **TF Calculation** → Computes Term Frequency (TF), i.e., how often a word appears in a document.
3. **IDF Calculation** → Computes **Inverse Document Frequency (IDF),** which reduces the weight of commonly occurring words.
4. **TF-IDF Score Computation** → Multiplies TF and IDF to get the final feature values.

Graph Based Models in NLP

- Graph-based models in Natural Language Processing (NLP**) use graph structures** to **represent and analyze** relationships between **words, sentences, or documents**.

- Unlike traditional vector-based models (e.g., TF-IDF, Word2Vec), **graph-based** models **capture complex dependencies** and structures in textual data

Importance of Graph Based models:

Language is inherently structured, with relationships between words, phrases, and sentences. Graphs help model:

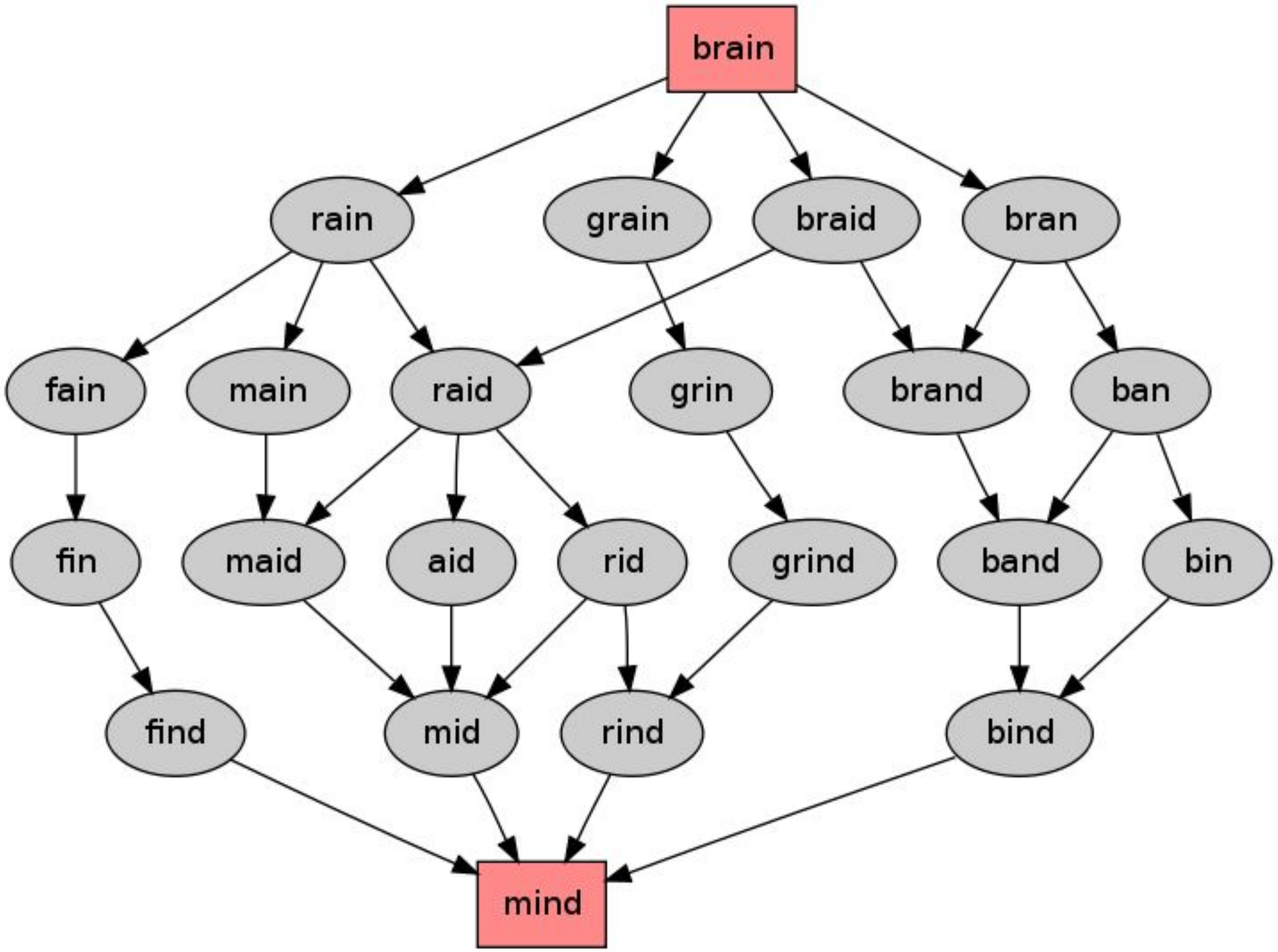
- **Semantic relationships** (e.g., synonyms, antonyms, hypernyms)
- **Syntactic dependencies** (e.g., subject-verb-object structures)
- **Document-level structures** (e.g., citations, hyperlinks)

Types of Graph-Based Models in NLP

A. Text as Graphs

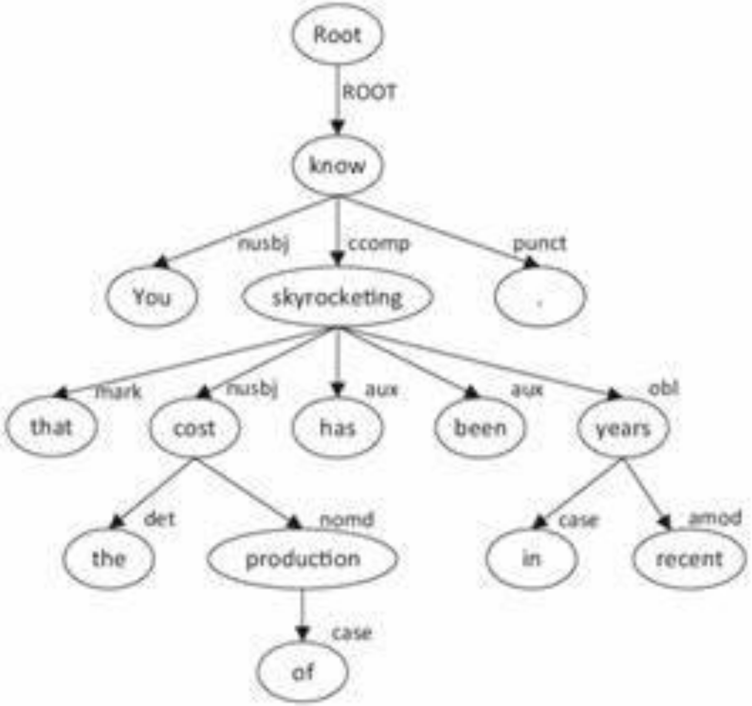
1. **Word Graphs → Nodes represent words**, and edges represent relationships (e.g., co-occurrence in sentences).
   - Example: **TextRank** (used in keyword extraction and summarization).
2. **Sentence Graphs → Nodes represent sentences**, and edges represent similarity scores.
   - Example: Used in **text summarization** and **document clustering**.
3. **Document Graphs → Nodes represent documents**, and edges represent document similarity (e.g., citation networks in research papers).
   - Example: Used in **document recommendation systems**.

Word Graph


brain
rain
grain
braid
bran
fain
main
raid
grin
brand
ban
fin
maid
aid
rid
grind
band
bin
find
mid
rind
bind
mind

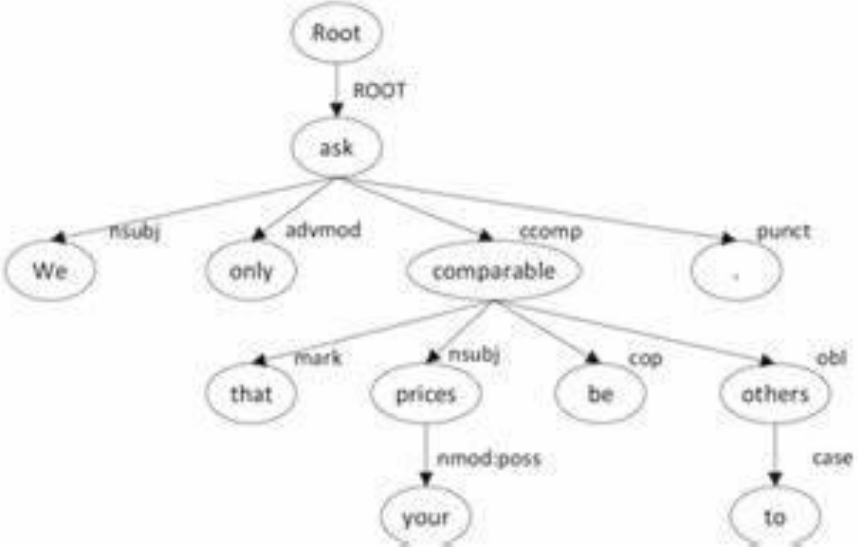
Utterance 5:

You know that the cost of production has been skyrocketing in recent years.

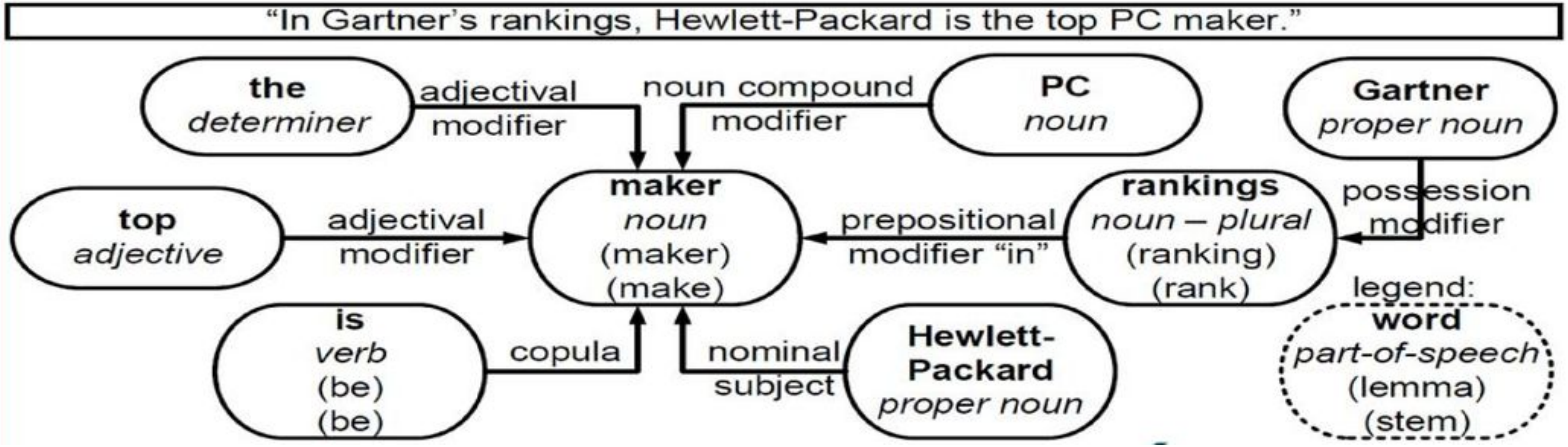


Current Turn:

We only ask that your prices be comparable to others.



Graph representation of sentence



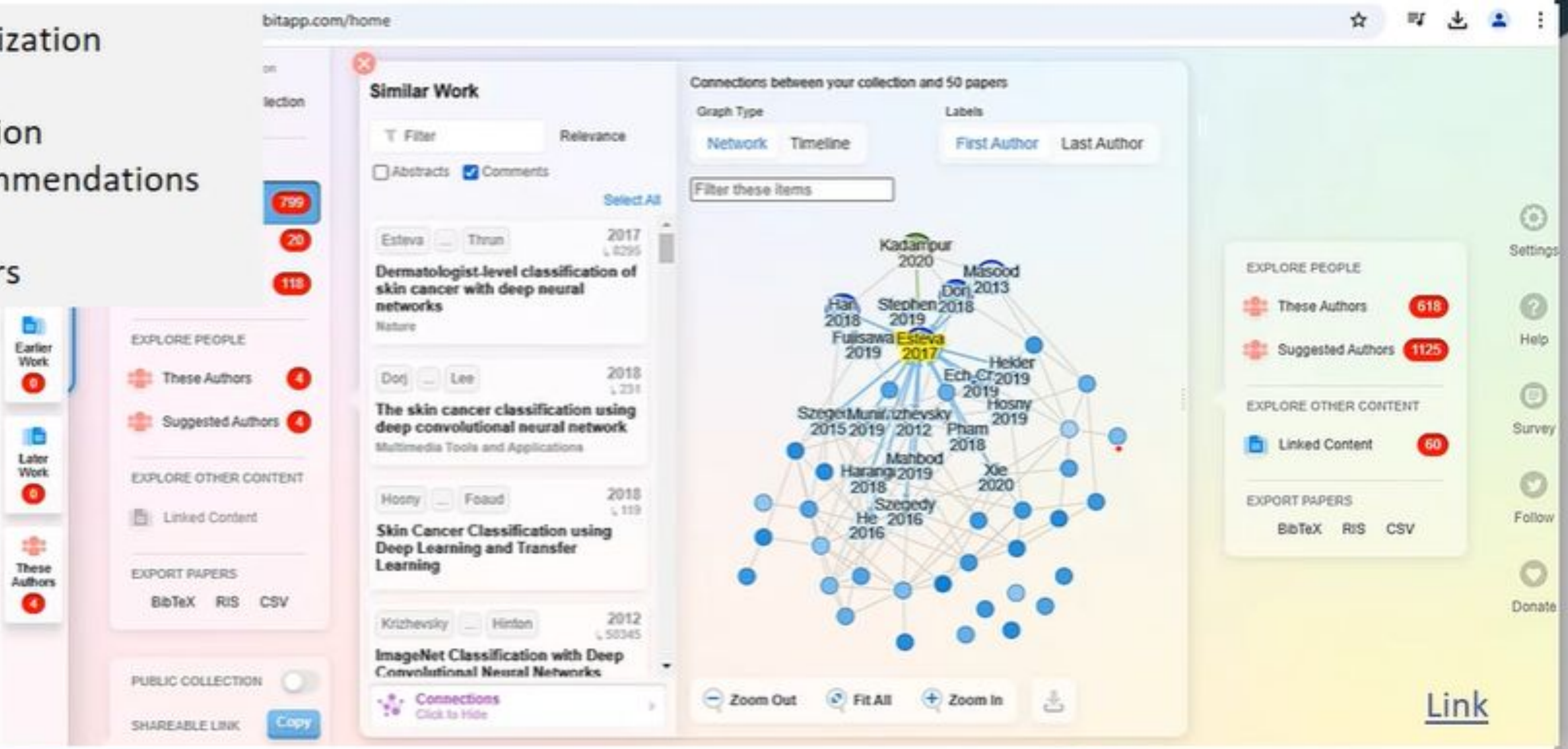
Research Rabbit

ResearchRabbit

Search Ref : https://doi.org/10.1016/j.imu.2019.100282

Search for new papers and manage literature survey

- ✓ Powerful visualization
- ✓ Rapidly explore
- ✓ Curated collection
- ✓ Improves recommendations
- ✓ Collaborate
- ✓ Discover authors



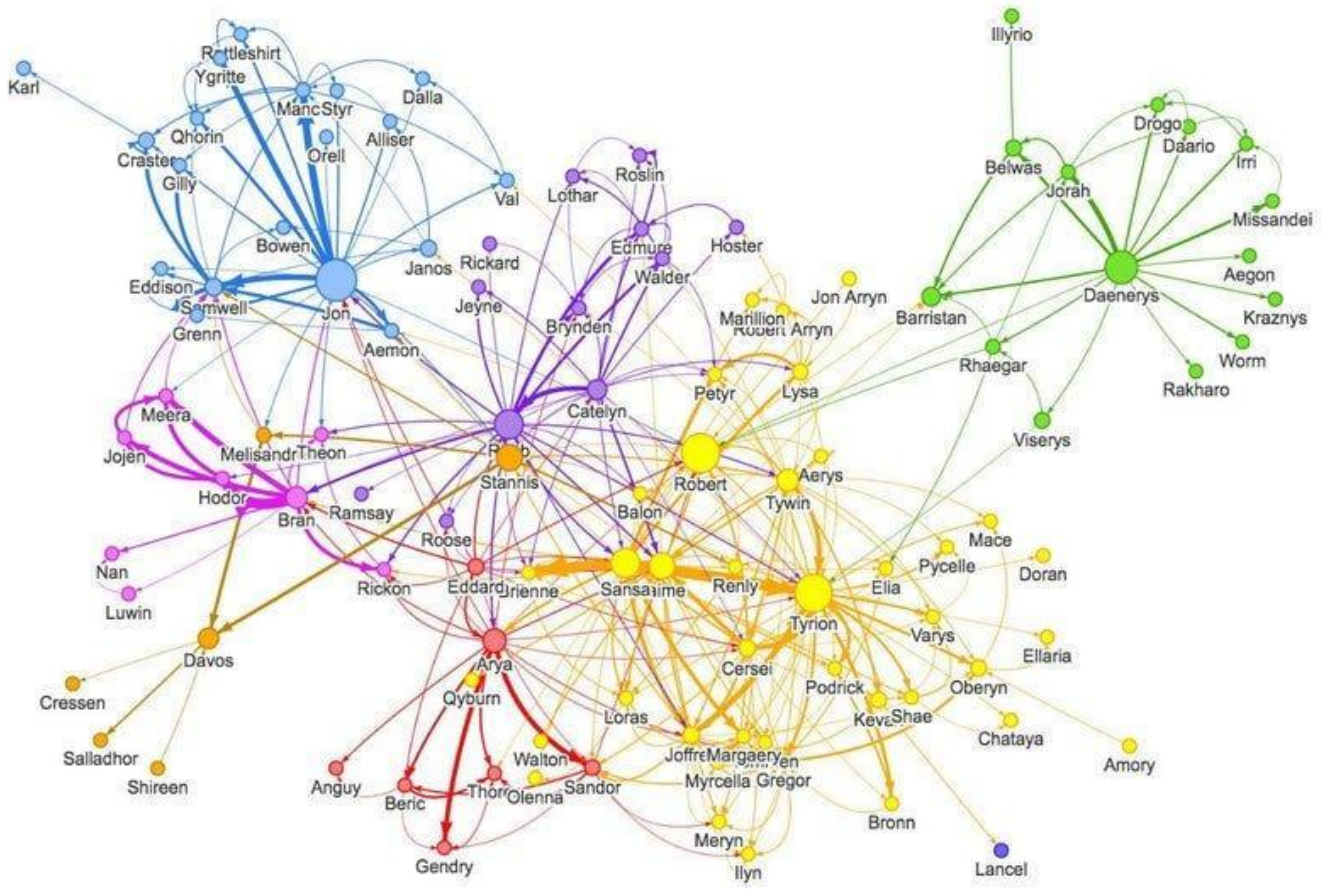
Link

B. Graph Neural Networks (GNNs) in NLP

- Graph Neural Networks (GNNs) generalize deep learning to graph structures and are used in advanced NLP tasks.

1. **Graph Convolutional Networks (GCN)**
   - Apply convolution operations on graphs to learn node embeddings.
   - Used for **text classification**, **named entity recognition (NER)**.
2. **Graph Attention Networks (GAT)**
   - Assign different weights to neighboring nodes based on importance.
   - Used in **knowledge graph completion**, **semantic similarity**.
3. **Graph-based Transformers**
   - Extend Transformer models (e.g., BERT) to graphs.
   - Example: **GraphBERT** processes text structured as a graph.

Graph Neural Network



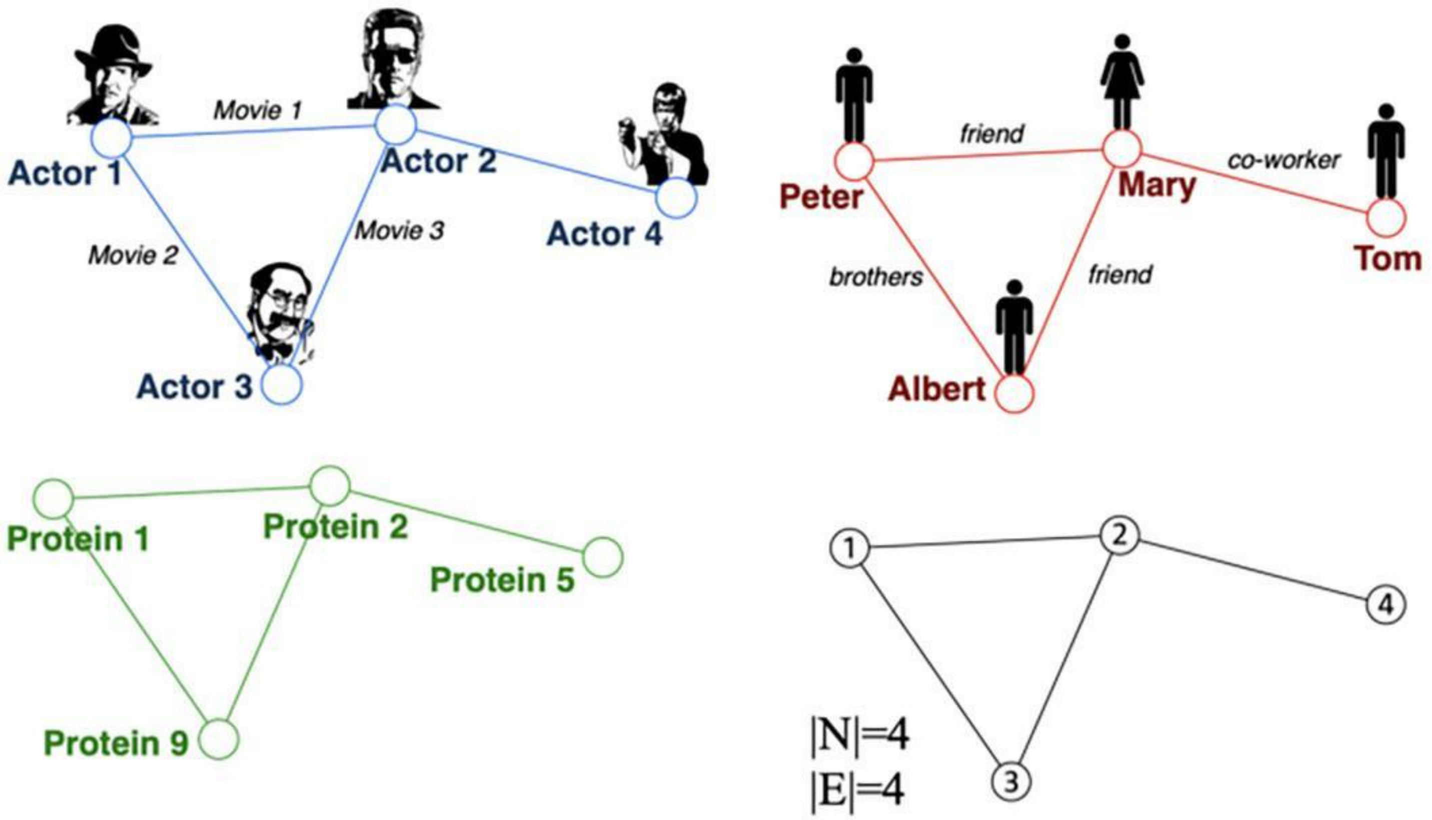
Graph Convolutional Networks

Graph Representation

- A graph G=(V,E)G = (V, E)G=(V,E) consists of:
- V: A set of nodes (vertices)
- E: A set of edges (connections between nodes)
- A: The adjacency matrix representing connections
- X: The feature matrix of nodes

Each node has a feature vector xi, and the ***goal is to learn a meaningful node representation by aggregating information from neighbors.***

Graph Convolutional Networks



- Apply convolution operations on graphs to learn node embeddings.
- Used for **text classification**, **named entity recognition (NER)**.

Graph Attention Networks (GAT)

- Graph Attention Networks (GAT) are a type of neural network architecture designed for processing graph-structured data.

- They extend Graph Convolutional Networks (GCNs) by **incorporating attention mechanisms to dynamically weight the importance of neighboring nodes** when aggregating information.

Key Points

Graph Representation:

- A graph G=(V,E) consists of nodes V and edges E.
- Each **node has feature vectors**, and the goal is to **learn meaningful node representations**.

Self-Attention Mechanism:

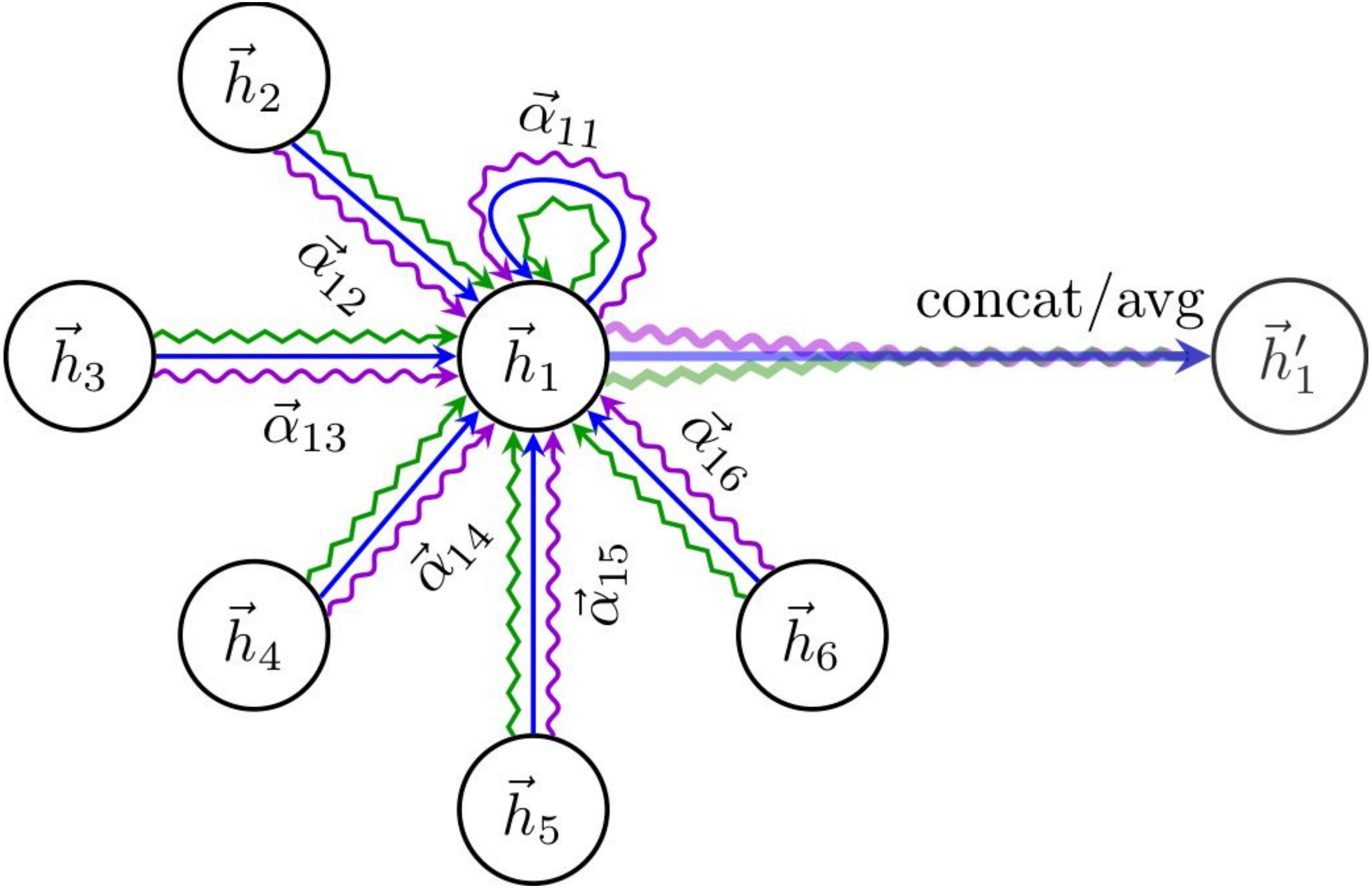
- Unlike GCNs that apply a fixed weighting scheme, **GAT learns edge weights dynamically** Using an attention mechanism.
- The **attention score between two nodes** is computed **based on their feature similarity**.

Multi-Head Attention:

- Instead of a single attention mechanism, **multiple attention heads are used in parallel**.
- The outputs are then aggregated (concatenated or averaged) for better learning stability.

Propagation and Aggregation:

- Each node **attends to its neighbors and aggregates their features** using learned attention weights.
- This allows the model to **focus more on important connections.**



$\vec{h}_2$
$\vec{\alpha}_{11}$
$\vec{\alpha}_{12}$
$\vec{h}_3$
$\vec{h}_1$
concat/avg
$\vec{h}'_1$
$\vec{\alpha}_{13}$
$\vec{\alpha}_{16}$
$\vec{\alpha}_{14}$
$\vec{\alpha}_{15}$
$\vec{h}_4$
$\vec{h}_6$
$\vec{h}_5$

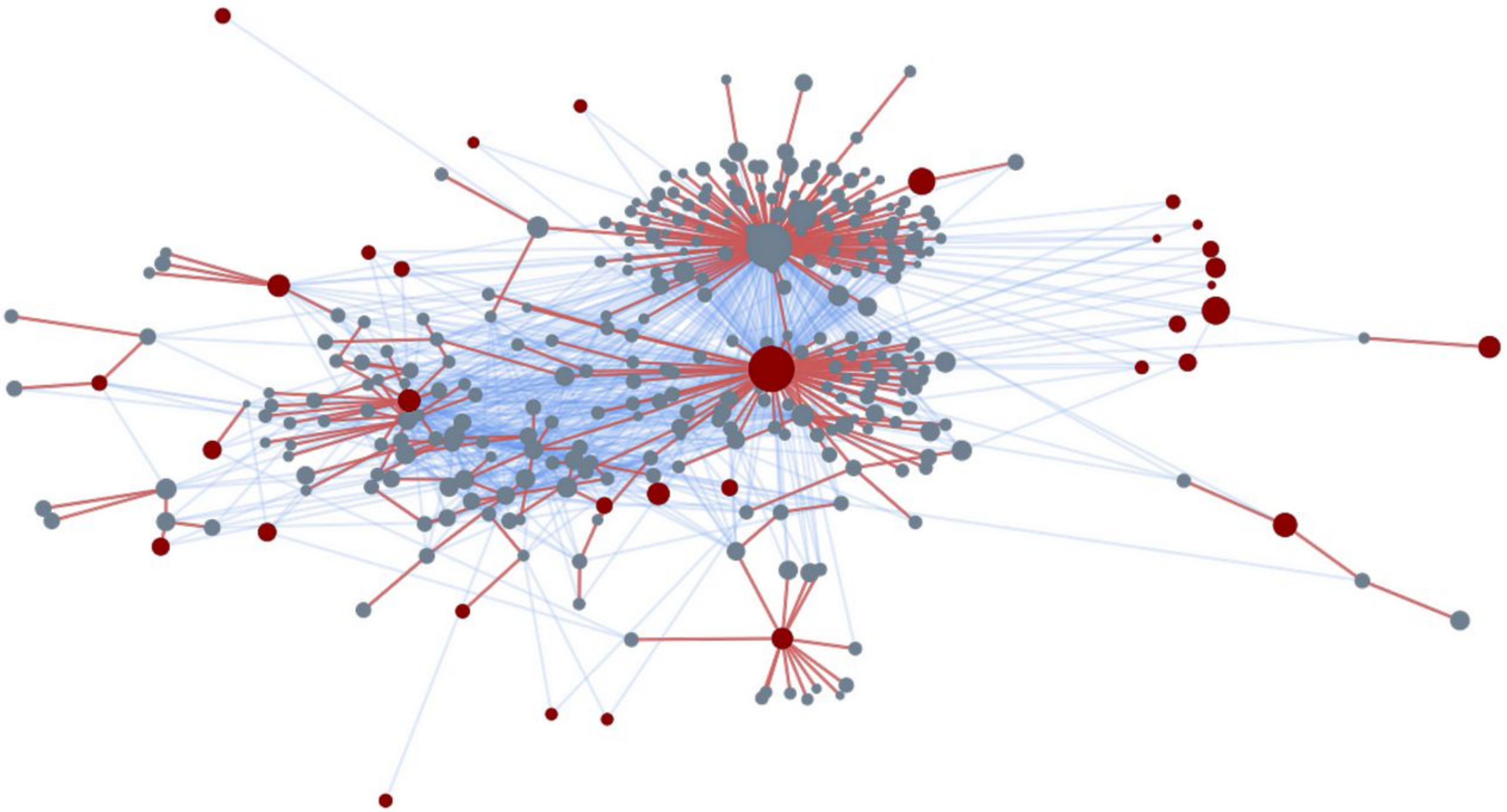


Figure 1: Example of a single news story spreading on a subset of the Twitter social network. Social connections between users are visualized as light-blue edges. A news URL is tweeted by multiple users (cascade roots denotes in red), each producing a cascade propagating over a subset of the social graph (red edges). Circle size represents the number of followers. Note that some cascades are small, containing only the root (the tweeting user) or just a few retweets.

Applications:- GCN/GAN

- **Social Networks:** Predicting user interests or influence.
- **Knowledge Graphs:** Enhancing relational reasoning.
- **Biomedical Networks:** Drug discovery and protein interaction predictions.
- **Natural Language Processing (NLP):** Semantic role labeling and text classification.

Applications of Graph-Based Models in NLP

Task	Graph Representation	Example Model
Keyword Extraction	Words as nodes, co-occurrence as edges	**TextRank**
Text Summarization	Sentences as nodes, similarity as edges	**LexRank**
Knowledge Graphs	Entities as nodes, relationships as edges	**WordNet, ConceptNet**
Question Answering	Entities as nodes, relations as edges	**QA over Knowledge Graphs**
Fake News Detection	News articles as nodes, citation links as edges	**Graph Neural Networks (GNNs)**

N Gram Models

- N-gram models in Natural Language Processing (NLP) are used to **predict the next word in a sequence based on the previous words**.
- They are built by analyzing contiguous sequences of "n" words in a text.

Types of N-Grams

1. **Unigram (n = 1)** – Single-word model
2. **Bigram (n = 2)** – Two-word sequence model
3. **Trigram (n = 3)** – Three-word sequence model
4. **Higher-order N-grams** (n > 3) – Longer word sequences

Example 1: Sentence Completion

Let’s take a simple sentence:
"I love machine learning.“

- **Unigram Model**: Breaks the sentence into single words:
  → ["I", "love", "machine", "learning"]

- **Bigram Model**: Looks at two-word sequences:
  → ["I love", "love machine", "machine learning"]

- **Trigram Model**: Looks at three-word sequences:
  → ["I love machine", "love machine learning"]

If we want to predict the next word after "machine", a **trigram model** trained on a large corpus would look at past occurrences and suggest the most likely word (e.g., "learning" or "vision").

Example 2: Autocomplete in Search Engines

When you type "best place to", a search engine might use N-grams to predict:

- **Bigram model:** Suggests "visit in"
- **Trigram model:** Suggests "visit in Europe"
- **Four-gram model:** Suggests "visit in Europe during"

Example 3: Spelling Correction

- If a user types "I hve a dreem", a bigram model can recognize that "hve" should be "have" because "I have" is more common than "I hve".
- Similarly, "dreem" can be corrected to "dream" based on common word pairs.

Formula for N-gram probability:

$$P(w_n|w_{n-1}, w_{n-2}, \ldots, w_1) = \frac{C(w_{n-1}, w_n)}{C(w_{n-1})}$$

where:

- $C(w_{n-1}, w_n)$ is the count of the bigram (or n-gram) appearing in the corpus.
- $C(w_{n-1})$ is the count of the previous word appearing in the corpus.

Example:

If we have the sentence:

"I love natural language processing."

- **Bigram model:**
  - $P(\text{"}language\text{"}|\text{"}natural\text{"}) = \frac{C(\text{"}naturallanguage\text{"})}{C(\text{"}natural\text{"})}$
  - If "natural language" appears 500 times and "natural" appears 1000 times in the dataset, then:

$$P(\text{"}language\text{"}|\text{"}natural\text{"}) = \frac{500}{1000} = 0.5$$

- **Trigram model:**
  - $P(\text{"}processing\text{"}|\text{"}naturallanguage\text{"}) = \frac{C(\text{"}naturallanguageprocessing\text{"})}{C(\text{"}naturallanguage\text{"})}$

Estimating Parameters & Smoothing

- Since raw probabilities from corpus data can be sparse (many word combinations may not appear in training data), smoothing techniques help handle unseen N-grams.
- **Common Smoothing Techniques:**
- **Laplace Smoothing (Add-One Smoothing)**
  - Adds 1 to all counts to avoid zero probabilities.

Formula:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + V}$$

where V is the vocabulary size.

Add-k Smoothing (Generalization of Laplace)

- Instead of adding 1, we add a small constant kkk.
- Helps control over-smoothing.

Backoff & Interpolation

- **Backoff:** If a higher-order N-gram (like trigram) is missing, fall back to lower-order models (bigram, unigram).
- **Interpolation:** Combines probabilities from multiple models:

$$P(w_n|w_{n-1}, w_{n-2}) = \lambda_1 P(w_n|w_{n-2}, w_{n-1}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n)$$

where $\lambda_1, \lambda_2, \lambda_3$ are weights summing to 1.

Example of Smoothing Impact:

If the bigram "blue car" has never appeared in training data, naive probability estimation would give $P(car \mid blue)=0$, but smoothing ensures it gets a small probability instead of zero.

Evaluating Language Models

- To measure how well an N-gram model performs, common evaluation metrics include:

Perplexity (PP)

  - Measures how well the model predicts a sample text.
  - Lower perplexity = better model.

Word Error Rate (WER)

- Used in speech recognition to check the difference between the predicted and actual words.

BLEU Score (for Machine Translation)

- Compares model-generated text with a reference translation.

Perplexity is calculated as:

$$PP(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}}$$

or equivalently:

$$PP(W) = 2^{H(W)}$$

where H(W) is the **entropy** (average uncertainty) of the model.

- **Lower perplexity means the model is more confident** in its predictions.
- **Higher perplexity means the model is more confused** and spreads probability over many possible next words.

Example

Let’s compare two models predicting the next word for:
"I love machine ____"

Model A's probability distribution:

1. "learning" → **0.8**
2. "tools" → 0.1
3. "cars" → 0.05
4. "games" → 0.05

Perplexity of Model A: ~1.3 (low, good model)

- Since "learning" has the highest probability (0.8), the model is confident.

Assumption:

•The correct word is **"learning"**.
•We calculate **PP for this correct word**.

$$PP_A = \frac{1}{P(\text{"}learning\text{"})}$$

$$PP_A = \frac{1}{0.8} = 1.25$$

Model B's probability distribution:

1. "learning" → **0.3**
2. "tools" → 0.3
3. "cars" → 0.2
4. "games" → 0.2

Perplexity of Model B: ~3.2 (higher, worse model)

- The model is uncertain because it spreads probability more evenly across words.

Again, assuming **"learning"** is the correct word:

$$PP_B = \frac{1}{P(\text{"}learning\text{"})}$$

$$PP_B = \frac{1}{0.3} = 3.33$$

Model A has lower perplexity (1.25) → Better

- It assigns **high probability (0.8)** to the correct word, meaning it is confident in its prediction.

Model B has higher perplexity (3.33) → Worse

- It spreads probability more evenly, meaning it is more **uncertain** about the next word.

Word Error Rate (WER) in NLP & Speech Recognition

- Word Error Rate (WER) is a **common metric** used to **evaluate the accuracy of speech-to-text systems, machine translation, and text generation models**.
- It measures **how different** the **predicted text (hypothesis**) is from the correct text **(reference).**

WER is calculated as:

$$WER = \frac{S + D + I}{N}$$

where:

- S = **Substitutions** (wrong word in place of the correct one)
- D = **Deletions** (missing word)
- I = **Insertions** (extra words added)
- N = **Total words in the reference sentence**

Example

Reference Sentence (Ground Truth)

✅ "I love natural language processing"

Hypothesis Sentence (Predicted by Model)

❌ "I like natural processing“

Now, let’s identify the errors:

Substitution (S): "love" → "like" (**1 error**)

Deletion (D): "language" is missing (**1 error**)

Insertion (I): No extra words (**0 errors**)

Total words in reference sentence (N): **5**

$$WER = \frac{1(S) + 1(D) + 0(I)}{5(N)}$$

✅ **Final Word Error Rate (WER) = 40%**

- **Lower WER is better** because it means the model makes fewer errors.
- **Higher WER means the model struggles** with accuracy.

For example, in speech recognition:

- **WER = 5%** → Very accurate system
- **WER = 40%** → Many mistakes, needs improvement

BLEU Score (Bilingual Evaluation Understudy) in NLP

- The **BLEU score** is a metric used to evaluate the quality of machine-generated text, particularly in **machine translation**, by comparing it to one or more **human reference translations**.
- The BLEU score is based on **precision** (how many predicted words match the reference) and **brevity penalty** (to avoid overly short translations).

$$BLEU = BP \times \exp\left(\sum_{n=1}^{N} w_n \log P_n\right)$$

where:

- BP = **Brevity Penalty** (penalizes short translations)
- P_n = **Precision for n-grams** (unigrams, bigrams, trigrams, etc.)
- w_n = **Weight assigned to each n-gram precision**

Typically, **BLEU-4 (4-gram precision)** is most used.

Example

- **Reference Translation (Human):**
- ✅ "The cat is on the mat"
- **Machine Translation (Predicted):**
- ❌ "The cat on mat"
- **Step 3: Compute BLEU Score**

Unigram Precision (1-gram matches):

1. **Matching words:** "The", "cat", "mat"
2. **Total words in prediction:** 4
3. **Precision = 3/4 = 0.75**

Bigram Precision (2-gram matches):

- **Matching bigrams:** ("The cat"), ("on mat")
- **Total bigrams in prediction:** 3
- **Precision = 2/3 ≈ 0.67**

Trigram Precision (3-gram matches):

- **Matching trigram:** None
- **Precision = 0/2 = 0**

Brevity Penalty (BP)

- Reference length = 6, Predicted length = 4
- Since prediction is shorter, we apply

$$BP = e^{(1-\text{ref length}/\text{hyp length})} = e^{(1-6/4)} = e^{-0.5} \approx 0.6$$

Final BLEU Score Calculation

Combining **1-gram, 2-gram, 3-gram** precisions (simplified example)

$$BLEU = 0.6 \times e^{\frac{1}{3}(\log 0.75 + \log 0.67 + \log 0)}$$

After applying a smoothing factor, we might get **BLEU ≈ 0.45 (or 45%)**.

- **BLEU = 1 (or 100%)** → Perfect translation
- **BLEU > 0.7 (or 70%)** → High-quality translation
- **BLEU ≈ 0.4 - 0.6 (40-60%)** → Understandable, but some errors
- **BLEU < 0.3 (30%)** → Poor translation
- 🔹 **Higher BLEU = better translation quality**
🔹 **BLEU penalizes missing words and incorrect word order**

Applications BLEU:

- Machine Translation (Google Translate, DeepL, etc.)
- Text Summarization & Paraphrasing Models
- Chatbot & Conversational AI Evaluation