

Natural Language Processing (315329)

Model Question Paper — Set A · Model Answers

Q.1

a) Define NLP; two applications.

2 M

Natural Language Processing (NLP) is a branch of Artificial Intelligence that enables computers to **understand, interpret, process and generate human (natural) language** — text or speech — using techniques from computational linguistics, statistics and machine learning.

Applications (any two): machine translation (Google Translate), chatbots and virtual assistants (Alexa, Siri), sentiment analysis of reviews, spam filtering, text summarization, speech recognition.

(Definition 1 M, two applications 1 M)

b) Regular expression; PIN code pattern.

2 M

A **regular expression (regex)** is a sequence of characters that defines a **search pattern**; it is used to search, match, extract or replace strings in text (e.g. `[0-9]+` matches one or more digits).

Regex for PIN code: `^[1-9][0-9]{5}$`

`[1-9]` → first digit 1–9, `[0-9]{5}` → exactly five more digits, `^$` → whole string. Example: 431601 ✓, 031601 ✗ (checked with Python `re.match`).

(Definition 1 M, correct regex 1 M)

c) Stemming; two examples.

2 M

Stemming is a text normalization technique that reduces an inflected or derived word to its **stem (root form)** by chopping off suffixes (and sometimes prefixes) using simple rules. The stem need not be a valid dictionary word.

Examples (Porter stemmer output): playing → play, connected → connect, studies → studi.

(Definition 1 M, two examples 1 M)

d) Bag-of-Words model.

2 M

The **Bag-of-Words (BoW)** model represents a document as a vector of **word counts (frequencies)** over a fixed vocabulary, ignoring grammar and word order — the text is treated like a “bag” of words.

Example: vocabulary [I, like, NLP, Python]; “I like NLP, I like Python” → [2, 2, 1, 1].

(Definition 1 M, example 1 M)

e) NER; four entity types.

2 M

Named Entity Recognition (NER) is the NLP task of locating **named entities** (proper names of real-world objects) in text and classifying them into predefined categories.

Types: PERSON (Sachin Tendulkar), ORGANIZATION (ISRO), LOCATION / GPE (Pune, India), DATE (15 August 1947), also TIME, MONEY, PERCENT.

(Definition 1 M, four types 1 M)

f) Cosine similarity formula; range.

2 M

Cosine similarity measures the similarity of two vectors by the cosine of the angle between them:

$$\cos \theta = (\mathbf{A} \cdot \mathbf{B}) / (|\mathbf{A}| \times |\mathbf{B}|) = \sum A_i B_i / (\sqrt{\sum A_i^2} \times \sqrt{\sum B_i^2})$$

In general it lies between -1 and $+1$; for term-frequency (non-negative) vectors it lies between **0 (no common terms)** and **1 (same direction / identical distribution)**.

(Formula 1 M, range 1 M)

g) Ambiguity; lexical ambiguity example.

2 M

Ambiguity in NLP means that a word, phrase or sentence can be interpreted in **more than one way** (it has more than one meaning or structure), so the machine must choose the correct meaning from context.

Lexical ambiguity example: “I went to the *bank*.” — *bank* may mean a financial institution or a river bank.

(Definition 1 M, example 1 M)

Q.2

a) Semantic network with example.

4 M

A **semantic network** is a knowledge-representation technique in which knowledge is stored as a **directed graph**: **nodes** represent concepts / objects and labelled **arcs (edges)** represent the semantic relations between them. It shows how word meanings are related and supports inheritance of properties.

Common relations: *is-a* (class membership / hyponymy), *has* / *has-part* (meronymy), *can*, *lives-in*, *instance-of*.

Example: “Tommy is a dog. A dog is a mammal. A mammal is an animal. A dog has a tail and can bark. A mammal has fur.”

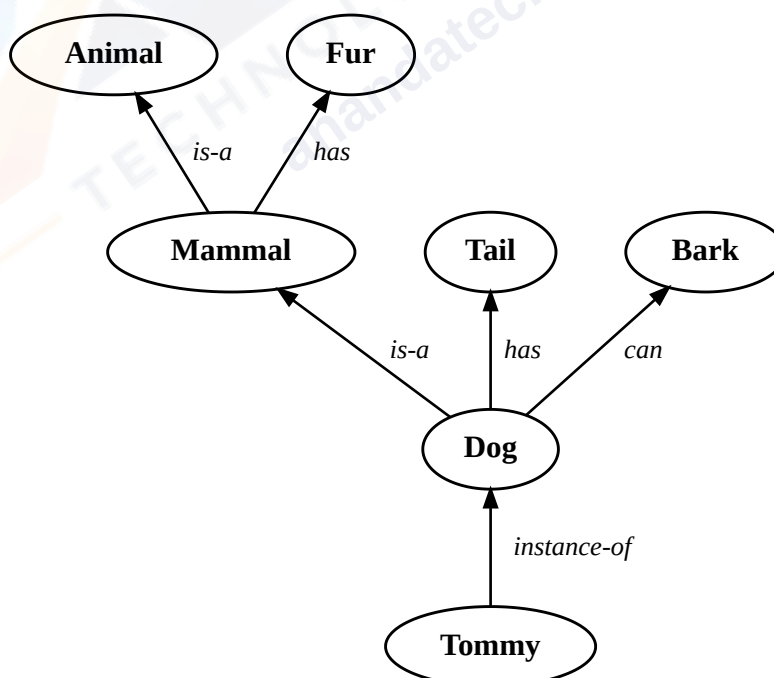


Fig.: Semantic network for the example

Inheritance: from the graph the system can infer “Tommy has fur” and “Tommy is an animal” by following the *is-a* links upward. WordNet is a large semantic network of English.

(Explanation 2 M, diagram/example 2 M)

b) Stemming vs lemmatization.

4 M

Sr.	Point	Stemming	Lemmatization
1	Meaning	Cuts suffixes using rules to get the stem.	Uses vocabulary (dictionary) and morphological analysis to get the dictionary form (lemma).
2	Output	Stem may not be a valid word (studies → studi).	Lemma is always a valid word (studies → study).
3	Context / POS	Does not consider context or part of speech.	Uses POS of the word (better as adjective → good).
4	Speed	Faster, simple rule-based.	Slower, needs dictionary lookup (WordNet).
5	Accuracy	Less accurate; over-/under-stemming.	More accurate and meaningful.
6	Tools	Porter, Snowball, Lancaster stemmers.	WordNet lemmatizer, spaCy lemmatizer.

Example (NLTK):

```
from nltk.stem import PorterStemmer, WordNetLemmatizer

ps = PorterStemmer()
lm = WordNetLemmatizer()
words = ['studies', 'running', 'better', 'caring', 'feet', 'wolves', 'happily']
pos = {'running': 'v', 'better': 'a', 'caring': 'v', 'happily': 'r'} # POS hints

print(f"{'Word':<10}{'Stem (Porter)':<15}{'Lemma (WordNet)'}")
for w in words:
    print(f"{'w':<10}{'ps.stem(w):<15}{'lm.lemmatize(w, pos.get(w, 'n'))}"))
```

Output:

Word	Stem (Porter)	Lemma (WordNet)
studies	studi	study
running	run	run
better	better	good
caring	care	care
feet	feet	foot
wolves	wolv	wolf
happily	happili	happily

(Any four points 1 M each)

c) Text corpus; corpus annotation.

4 M

A **text corpus** (plural *corpora*) is a large, structured collection of machine-readable texts (written or transcribed speech) used for linguistic analysis and for training / testing NLP models. Examples: Brown Corpus, Penn Treebank, Reuters, Gutenberg, WordNet (lexical database).

Corpus annotation is the process of adding extra linguistic information (tags / labels) to the raw text of a corpus so that it can be used for supervised learning and analysis. Types:

- **POS annotation** — each word gets a part-of-speech tag: The/DT dog/NN barked/VBD (Brown, Penn Treebank).
- **Syntactic annotation (treebank)** — sentences are given parse trees: (S (NP The dog) (VP barked)).
- **Named entity annotation** — [Sachin]PER lives in [Mumbai]LOC, often in BIO format.
- **Semantic / sentiment annotation** — word senses, sentiment labels (positive / negative), coreference.

Utilities: NLTK's corpus readers provide `words()`, `sents()`, `tagged_words()`, `categories()`, `fileids()` to access annotated corpora, e.g. `brown.tagged_words()[ :2]` → `[('The', 'AT'), ('Fulton', 'NP-TL')]`.

(Corpus 1 M, annotation and types 3 M)

d) POS tagging of the given sentence.

4 M

POS tagging assigns a part-of-speech tag (Penn Treebank tag set) to every token. Output of NLTK's `pos_tag()`:

```
import nltk

sentence = "Virat Kohli scored a century in Mumbai yesterday"
tokens = nltk.word_tokenize(sentence)
tags = nltk.pos_tag(tokens)          # Penn Treebank tag set
for word, tag in tags:
    print(f"{word:<10} {tag}")
```

Output:

```
Virat      NNP
Kohli      NNP
scored     VBD
a          DT
century    NN
in         IN
Mumbai     NNP
yesterday  NN
```

Word	Tag	Meaning of tag
Virat	NNP	Proper noun, singular
Kohli	NNP	Proper noun, singular
scored	VBD	Verb, past tense
a	DT	Determiner
century	NN	Noun, singular
in	IN	Preposition
Mumbai	NNP	Proper noun, singular
yesterday	NN	Noun (time noun; used here as an adverb of time)

(Correct tags 3 M, explanation 1 M)

Q.3

a) Byte-Pair Encoding with example.

4 M

Byte-Pair Encoding (BPE) is a **subword tokenization** algorithm (used in GPT, RoBERTa). It starts with individual characters and **repeatedly merges the most frequent adjacent pair of symbols** into a new symbol, until the required vocabulary size / number of merges is reached. Frequent words become single tokens; rare / unknown words are split into known subwords, so there is no out-of-vocabulary problem.

Algorithm: 1) Split each word into characters + end-of-word marker '_'. 2) Count all adjacent symbol pairs (weighted by word frequency). 3) Merge the most frequent pair everywhere and add it to the vocabulary. 4) Repeat step 2–3 k times.

Step 0 (initial split into characters, '_' = end of word): p l a y _ (6) · p l a y e d _ (3) · p l a y e r _ (4) · r a y _ (2)

Initial vocabulary = {p, l, a, y, _, e, d, r} (8 symbols)

Merge	Most frequent pair	Freq.	New token	Corpus after merge
1	a + y	15	ay	p l a y _ (6) · p l a y e d _ (3) · p l a y e r _ (4) · r a y _ (2)
2	p + l	13	pl	p l a y _ (6) · p l a y e d _ (3) · p l a y e r _ (4) · r a y _ (2)
3	pl + ay	13	play	play _ (6) · play e d _ (3) · play e r _ (4) · r a y _ (2)
4	play + e	7	playe	play _ (6) · playe d _ (3) · playe r _ (4) · r a y _ (2)
5	play + _	6	play_	play_ (6) · playe d _ (3) · playe r _ (4) · r a y _ (2)

Final vocabulary (13 tokens): {p, l, a, y, _, e, d, r, ay, pl, play, playe, play_}

A new word is segmented by applying the same merges in order, e.g. *plays* → play + s + _; *rayed* → r + ay + e + d + _.

(Explanation 2 M, worked example 2 M. Ties (equal frequency) are broken by the first pair found.)

b) Attention (self-attention) mechanism.

4 M

Attention lets a model focus on the most relevant words of a sequence while producing each output. In **self-attention**, every word of a sentence looks at all other words of the same sentence and builds a context-aware representation. E.g. in “The animal didn't cross the street because it was tired”, attention links *it* strongly to *animal*.

1. Each input embedding x is multiplied by three learned matrices to give a **Query (Q)**, **Key (K)** and **Value (V)** vector.
2. Scores = $Q \cdot K^T$ (how much a word should attend to every other word), scaled by $\sqrt{d_k}$.
3. Softmax converts the scores to attention weights (sum = 1).
4. Output = weighted sum of the V vectors: **Attention(Q, K, V) = softmax(QK^T / $\sqrt{d_k}$) V.**

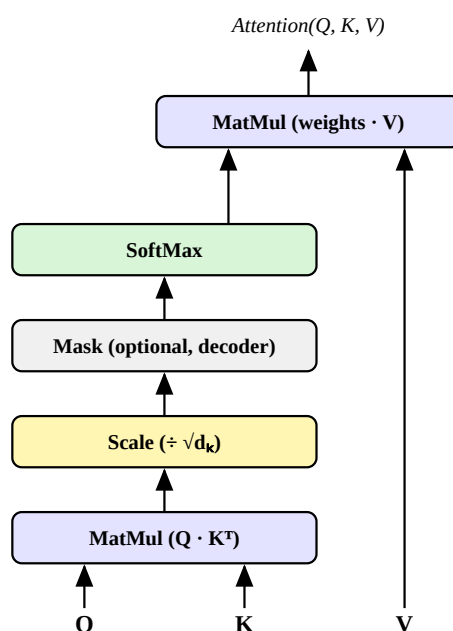


Fig.: Scaled dot-product (self-) attention

Multi-head attention runs several such attention ‘heads’ in parallel and concatenates them, so different heads capture different relations (syntax, coreference etc.).

(Explanation 2 M, formula/diagram 2 M)

c) Unigrams, bigrams and trigrams.

4 M

An **n-gram** is a contiguous sequence of n tokens. A sentence with T tokens has $T - n + 1$ n-grams.

```
from nltk import word_tokenize, ngrams

text = "Natural language processing helps computers understand human language"
tokens = word_tokenize(text.lower())
for n, name in [(1, 'Unigrams'), (2, 'Bigrams'), (3, 'Trigrams')]:
    grams = [' '.join(g) for g in ngrams(tokens, n)]
    print(f"{name} ({len(grams)}):", grams)
```

Output:

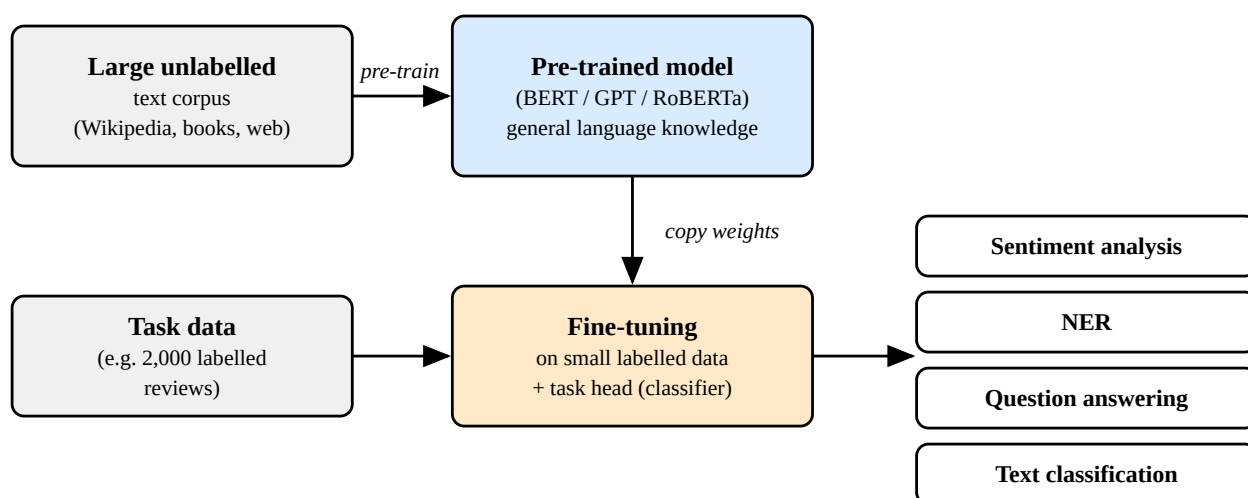
```
Unigrams (8): ['natural', 'language', 'processing', 'helps', 'computers', 'understand', 'human', 'language']
Bigrams (7): ['natural language', 'language processing', 'processing helps', 'helps computers', 'computers understand', 'understand human', 'human language']
Trigrams (6): ['natural language processing', 'language processing helps', 'processing helps computers', 'helps computers understand', 'computers understand human', 'understand human language']
```

(Each of unigram, bigram, trigram list 1 M; program/explanation 1 M)

d) Transfer learning in NLP.

4 M

Transfer learning is a technique in which a model trained on one (large, general) task is **reused as the starting point** for a different but related task. In NLP, a language model is first **pre-trained** on a huge unlabelled corpus and then **fine-tuned** on a small labelled dataset of the target task.



Source task: language modelling → Target task: a downstream NLP task

Fig.: Transfer learning in NLP — pre-train once, fine-tune for many tasks

1. **Pre-training:** model (BERT, GPT) learns grammar, word meaning and context by self-supervised tasks such as masked-word prediction or next-word prediction.
2. **Fine-tuning:** a small task-specific layer (e.g. a classifier) is added and all/some weights are trained further on task data.

Benefits: needs much less labelled data, less training time and computing cost, higher accuracy, one pre-trained model can be reused for many tasks.

(Explanation 2 M, diagram 1 M, benefits 1 M)

Q.4

a) Parse tree for “The old man sat on the bench.”

4 M

A **parse tree** (syntax tree) shows the syntactic structure of a sentence according to a context-free grammar. Grammar used:

$S \rightarrow NP VP$ $NP \rightarrow Det Adj N \mid Det N$ $VP \rightarrow V PP$ $PP \rightarrow P NP$

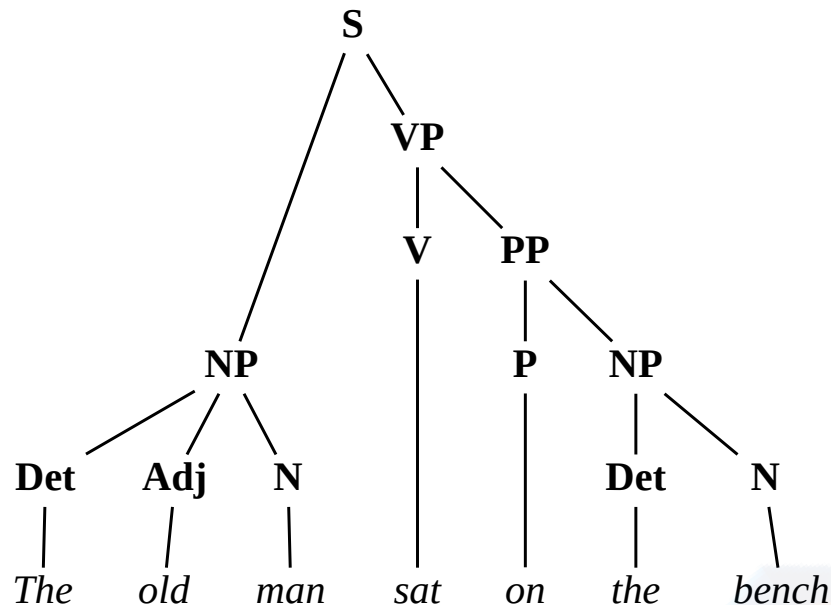


Fig.: Parse tree (generated with NLTK ChartParser using the above grammar)

Components:

- **Root node** — S (sentence).
- **Non-terminal (phrase) nodes** — NP (noun phrase), VP (verb phrase), PP (prepositional phrase).
- **Pre-terminal (POS) nodes** — Det, Adj, N, V, P.
- **Terminal (leaf) nodes** — the actual words; read left to right they give the sentence.
- **Edges / branches** — show which constituent is made of which parts (grammar rules applied).

Bracketed form: [S [NP [Det The] [Adj old] [N man]] [VP [V sat] [PP [P on] [NP [Det the] [N bench]]]]]

(Correct tree 2 M, components 2 M)

b) Chunking; NP extraction with NLTK.

4 M

Chunking (shallow / partial parsing) groups POS-tagged words into non-overlapping phrases (chunks) such as noun phrases (NP), verb phrases (VP) and prepositional phrases, without building a full parse tree. It is done with a chunk grammar of regular expressions over POS tags, e.g. NP: {<DT>?<JJ>*<NN.*>} = optional determiner, any number of adjectives, then a noun.

Steps: tokenize → POS-tag → apply chunk grammar (RegexpParser) → read chunks from the tree.

```

import nltk

sentence = "The young students read an interesting book in the big library"
tagged = nltk.pos_tag(nltk.word_tokenize(sentence))
print(tagged)

grammar = "NP: {<DT>?<JJ>*<NN.*>}" # optional determiner, any adjectives, a noun
parser = nltk.RegexpParser(grammar)
tree = parser.parse(tagged)
print(tree)

print("Noun phrases:")
  
```

```
for sub in tree.subtrees(filter=lambda t: t.label() == 'NP'):
    print(' ', ' '.join(word for word, tag in sub.leaves()))
```

Output:

```
[('The', 'DT'), ('young', 'JJ'), ('students', 'NNS'), ('read', 'VBP'), ('an', 'DT'),
('interesting', 'JJ'), ('book', 'NN'), ('in', 'IN'), ('the', 'DT'), ('big', 'JJ'),
('library', 'NN')]
```

```
(S
  (NP The/DT young/JJ students/NNS)
  read/VBP
  (NP an/DT interesting/JJ book/NN)
  in/IN
  (NP the/DT big/JJ library/NN))
```

Noun phrases:

```
The young students
an interesting book
the big library
```

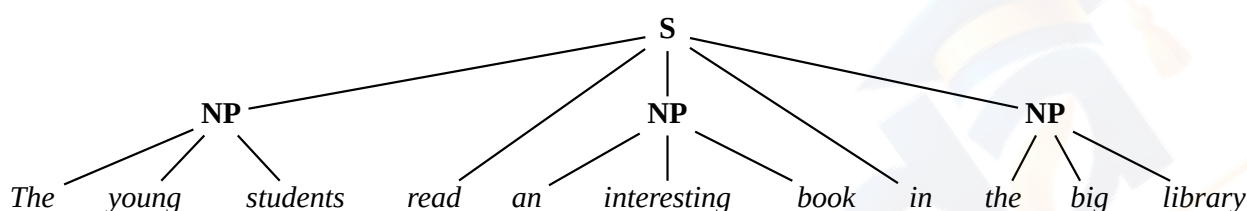


Fig.: Chunk tree (words shown below their chunks)

(Explanation 2 M, program with output 2 M)

c) TF-IDF calculation.

4 M

TF-IDF (Term Frequency – Inverse Document Frequency) gives a high weight to a term that is frequent in a document but rare in the collection.

Documents after lower-casing and tokenizing:

Doc	Tokens	No. of words
D1	nlp is fun and nlp is useful	7
D2	python is useful	3
D3	nlp uses python	3

Step 1 — Term Frequency $TF(t, d) = (\text{count of } t \text{ in } d) / (\text{total words in } d)$

Term	TF in D1	TF in D2	TF in D3
nlp	$2/7 = 0.286$	0	$1/3 = 0.333$
is	$2/7 = 0.286$	$1/3 = 0.333$	0
fun	$1/7 = 0.143$	0	0
and	$1/7 = 0.143$	0	0
useful	$1/7 = 0.143$	$1/3 = 0.333$	0
python	0	$1/3 = 0.333$	$1/3 = 0.333$
uses	0	0	$1/3 = 0.333$

Step 2 — Inverse Document Frequency $IDF(t) = \log_{10}(N / df(t))$, $N = 3$

Term	df(t)	N / df	IDF = $\log_{10}(N/df)$
nlp	2	1.5	0.1761
is	2	1.5	0.1761
fun	1	3	0.4771
and	1	3	0.4771
useful	2	1.5	0.1761
python	2	1.5	0.1761
uses	1	3	0.4771

Step 3 — $TF\text{-}IDF(t, d) = TF \times IDF$

Term	D1	D2	D3
nlp	0.0503	0.0000	0.0587
is	0.0503	0.0587	0.0000
fun	0.0682	0.0000	0.0000
and	0.0682	0.0000	0.0000
useful	0.0252	0.0587	0.0000
python	0.0000	0.0587	0.0587
uses	0.0000	0.0000	0.1590

Interpretation: nlp, is, useful, python occur in 2 of the 3 documents, so their IDF is low (0.1761). The highest weight is **uses** in D3 (0.1590) — it appears only in that document, so it best describes D3. A term present in all documents would get $IDF = \log_{10}(1) = 0$.

(TF 1 M, IDF 1 M, TF-IDF table 2 M (calculated with Python, values rounded to 4 decimals))

d) Synonymy, antonymy, hyponymy, polysemy.

4 M

- **Synonymy:** two different words having the same or nearly the same meaning. E.g. *big* – *large*, *buy* – *purchase*.
- **Antonymy:** words having opposite meanings. E.g. *good* – *bad*, *hot* – *cold*.
- **Hyponymy:** an ‘is-a-kind-of’ relation; the specific word is the hyponym and the general word is the hypernym. E.g. *car* is a hyponym of *motor vehicle*; *rose* is a hyponym of *flower*.
- **Polysemy:** one word having several related meanings (senses). E.g. *bank* — sloping land beside a river / financial institution / a pile; *head* of a body / head of a department.

These relations are stored in WordNet; NLTK output:

```
from nltk.corpus import wordnet as wn

# Synonymy: lemmas of the first sense of 'big'
print("Synonyms of big :", wn.synsets('big', pos='a')[0].lemma_names())
# Antonymy
good = wn.synset('good.a.01').lemmas()[0]
print("Antonym of good :", [a.name() for a in good.antonyms()])
# Hyponymy / hypernymy
```

```
car = wn.synset('car.n.01')
print("Hypernym of car :", [h.name() for h in car.hypernyms()])
kinds = [h.lemma_names()[0] for h in car.hyponyms()]
show = ('ambulance', 'cab', 'convertible', 'jeep', 'limousine')
print("Some hyponyms of car:", [k for k in kinds if k in show])
# Polysemy: number of senses of 'bank' (noun)
for s in wn.synsets('bank', pos='n')[:3]:
    print("bank ->", s.definition())
```

Output:

```
Synonyms of big : ['large', 'big']
Antonym of good : ['bad']
Hypernym of car : ['motor_vehicle.n.01']
Some hyponyms of car: ['cab', 'limousine', 'ambulance', 'convertible', 'jeep']
bank -> sloping land (especially the slope beside a body of water)
bank -> a financial institution that accepts deposits and channels the money into lending
activities
bank -> a long ridge or pile
```

(Each relation with example 1 M)

e) Hugging Face Hub and AutoTokenizer.

4 M

The **Hugging Face Hub** (huggingface.co) is an online platform that hosts thousands of **pre-trained models, datasets and demo apps (Spaces)**. Models are identified by names such as bert-base-uncased; they can be downloaded with `from_pretrained()` and shared with `push_to_hub()`. Each model has a model card (description, licence, usage).

AutoTokenizer automatically loads the correct tokenizer for a given model name. It converts text → tokens → input IDs (and attention mask) in the exact format expected by that model, and can decode IDs back to text. BERT uses WordPiece subwords ('##' = continuation of previous token).

```
from transformers import AutoTokenizer

# downloads the tokenizer files of the model from the Hugging Face Hub
tok = AutoTokenizer.from_pretrained("bert-base-uncased")
text = "Tokenizers make transformers unbelievably powerful"
print("Tokens   :", tok.tokenize(text))
enc = tok(text)
print("Input IDs:", enc["input_ids"])
print("Decoded   :", tok.decode(enc["input_ids"]))
```

Output:

```
Tokens   : ['token', '##izer', '##s', 'make', 'transformers', 'un', '##bel', '##ie',
'##va', '##bly', 'powerful']
Input IDs: [101, 19204, 17629, 2015, 2191, 19081, 4895, 8671, 2666, 3567, 6321, 3928,
102]
Decoded   : [CLS] tokenizers make transformers unbelievably powerful [SEP]
```

[CLS] (id 101) and [SEP] (id 102) are special tokens added by the BERT tokenizer.

(Hub 1 M, AutoTokenizer 1 M, example 2 M)

Q.5

a) Steps of raw text preprocessing.

6 M

Raw text from the web, social media or documents is noisy. **Text preprocessing (normalization)** converts it into a clean, uniform form before feature extraction.

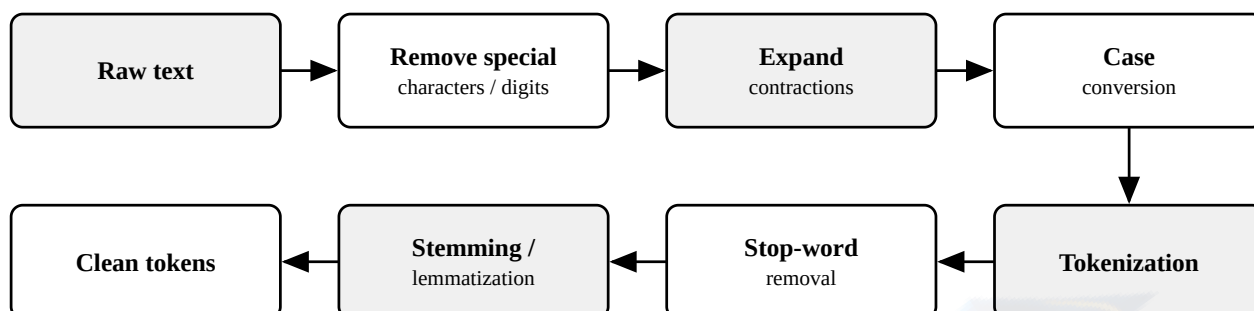


Fig.: Text preprocessing pipeline (order may vary with the task)

- 1. Removing special characters / HTML / digits:** using regex, e.g. `re.sub(r'[^a-z\s]', '', text)` removes punctuation, emojis, symbols.
- 2. Expanding contractions:** can't → can not, it's → it is, using a contraction dictionary.
- 3. Case conversion:** lower-casing so that 'Movie' and 'movie' are the same token.
- 4. Tokenization:** splitting text into words / sentences (`word_tokenize`, `sent_tokenize`).
- 5. Stop-word removal:** removing very common words (the, is, we, it) that carry little meaning. (For sentiment tasks negations like 'not' are often kept.)
- 6. Stemming / lemmatization:** reducing words to their base form (movies → movie).

Python example (NLTK):

```

import re, nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

raw = "OMG!! I can't believe it's 9:30 PM... The 3 movies we watched were AMAZING :)"
print("Raw text      :", raw)

# 1. expand contractions (small dictionary)
contractions = {"can't": "can not", "it's": "it is", "won't": "will not", "i'm": "i am"}
text = raw.lower()
for c, full in contractions.items():
    text = text.replace(c, full)
print("Lower+expanded:", text)

# 2. case conversion

# 3. remove special characters and digits (keep letters and spaces)
text = re.sub(r'[^a-z\s]', " ", text)
text = re.sub(r"\s+", " ", text).strip()
print("Cleaned      :", text)

tokens = nltk.word_tokenize(text)
print("Tokens      :", tokens)

# 4. tokenization

stop = set(stopwords.words('english'))
tokens = [t for t in tokens if t not in stop]
print("No stop words :", tokens)

# 5. stop-word removal
  
```

```
lem = WordNetLemmatizer() # 6. lemmatization
print("Lemmatized : ", [lem.lemmatize(t) for t in tokens])
```

Output:

```
Raw text      : OMG!! I can't believe it's 9:30 PM... The 3 movies we watched were
AMAZING :)
Lower+expanded: omg!! i can not believe it is 9:30 pm... the 3 movies we watched were
amazing :)
Cleaned       : omg i can not believe it is pm the movies we watched were amazing
Tokens       : ['omg', 'i', 'can', 'not', 'believe', 'it', 'is', 'pm', 'the', 'movies',
'we', 'watched', 'were', 'amazing']
No stop words : ['omg', 'believe', 'pm', 'movies', 'watched', 'amazing']
Lemmatized    : ['omg', 'believe', 'pm', 'movie', 'watched', 'amazing']
```

(Steps 3 M, program/example 3 M)

b) Vector space model; cosine similarity numerical.

6 M

In the **Vector Space Model (VSM)** each document (or word) is represented as a **vector in an n-dimensional space**, where each dimension is one term of the vocabulary and the value is its weight (count, TF or TF-IDF). Similar documents point in similar directions, so similarity is measured by the angle between vectors (cosine similarity), not by their length.

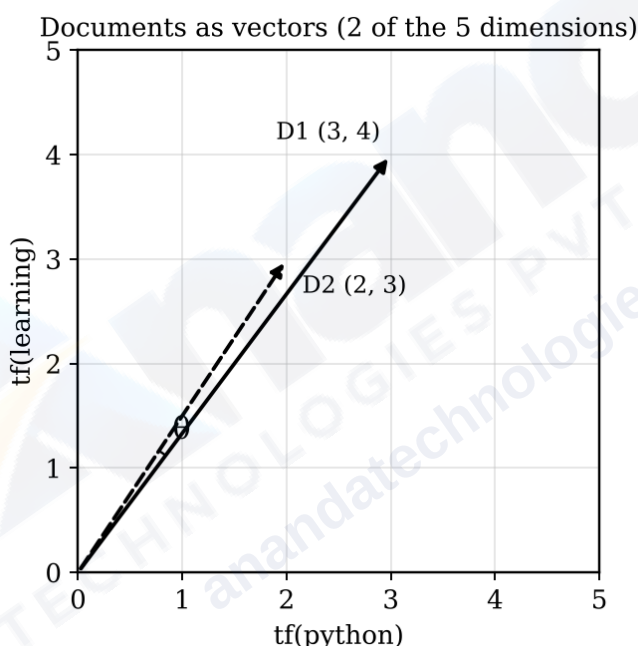


Fig.: VSM — the smaller the angle θ , the more similar the documents

Given: vocabulary = ['data', 'science', 'python', 'model', 'learning']; D1 = [2, 0, 3, 1, 4], D2 = [1, 2, 2, 0, 3]

i	D1 _i	D2 _i	D1 _i × D2 _i	D1 _i ²	D2 _i ²
1	2	1	2	4	1
2	0	2	0	0	4
3	3	2	6	9	4
4	1	0	0	1	0
5	4	3	12	16	9
Σ			20	30	18

$D1 \cdot D2 = (2 \times 1) + (3 \times 2) + (4 \times 3) = 20$ (terms with a zero are omitted)

$|D1| = \sqrt{30} = 5.4772$; $|D2| = \sqrt{18} = 4.2426$

$\cos \theta = (D1 \cdot D2) / (|D1| \times |D2|) = 20 / (5.4772 \times 4.2426) = 20 / 23.2379 = 0.8607 \Rightarrow \theta = \cos^{-1}(0.8607) \approx 30.6^\circ$

Interpretation: cosine similarity = 0.8607 (close to 1), so D1 and D2 are **highly similar** — both are dominated by the terms ‘python’ and ‘learning’ and are probably about the same topic. A value near 0 would mean the documents share almost no terms.

(VSM explanation 2 M, calculation 3 M, interpretation 1 M (calculated with Python))

c) POS tagging; techniques.

6 M

Part-of-Speech (POS) tagging is the process of assigning a grammatical category (noun, verb, adjective, ...) to every word of a sentence, based on its definition and its **context**. Tags normally come from a tag set such as the Penn Treebank (NN, VB, JJ, ...) or the Universal tag set (NOUN, VERB, ...). It is the first step for parsing, NER, and information extraction.

Techniques:

1. **Rule-based tagging:** uses a dictionary to give each word its possible tags and hand-written rules to remove wrong tags. Example rule: *if a word follows a determiner (a, the) and is before a noun, tag it adjective; a word ending in -ly is an adverb*. Examples: ENGTWOL tagger.
2. **Stochastic (probabilistic) tagging:** learns probabilities from a tagged corpus. A **Hidden Markov Model (HMM)** chooses the tag sequence T that maximizes $P(T | W) \propto \prod P(w_i | t_i) \cdot P(t_i | t_{i-1})$ (emission × transition probabilities); the best sequence is found with the **Viterbi algorithm**.
3. **Transformation-based (Brill) tagging:** hybrid — first gives every word its most frequent tag, then applies learned transformation rules in order, e.g. *change NN to VB when the previous tag is TO* (“to/TO race/NN” → “to/TO race/VB”).
4. **Neural / deep learning taggers:** BiLSTM, transformer models (spaCy, Stanza) give state-of-the-art accuracy.

Example (context decides the tag): NLTK output for “They refuse to permit us to obtain the refuse permit”:

```
[('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'), ('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', 'NN')]
```

The same word ‘refuse’ is tagged VBP (verb) first and NN (noun) later, and ‘permit’ VB then NN — the tagger uses the surrounding words.

Q.6

a) Transformer architecture.

6 M

The **Transformer** (Vaswani et al., 2017, “Attention is all you need”) is a deep-learning architecture for sequence-to-sequence tasks that uses only **attention** (no RNN / CNN), so all words are processed **in parallel**. It is the basis of BERT, GPT and T5.

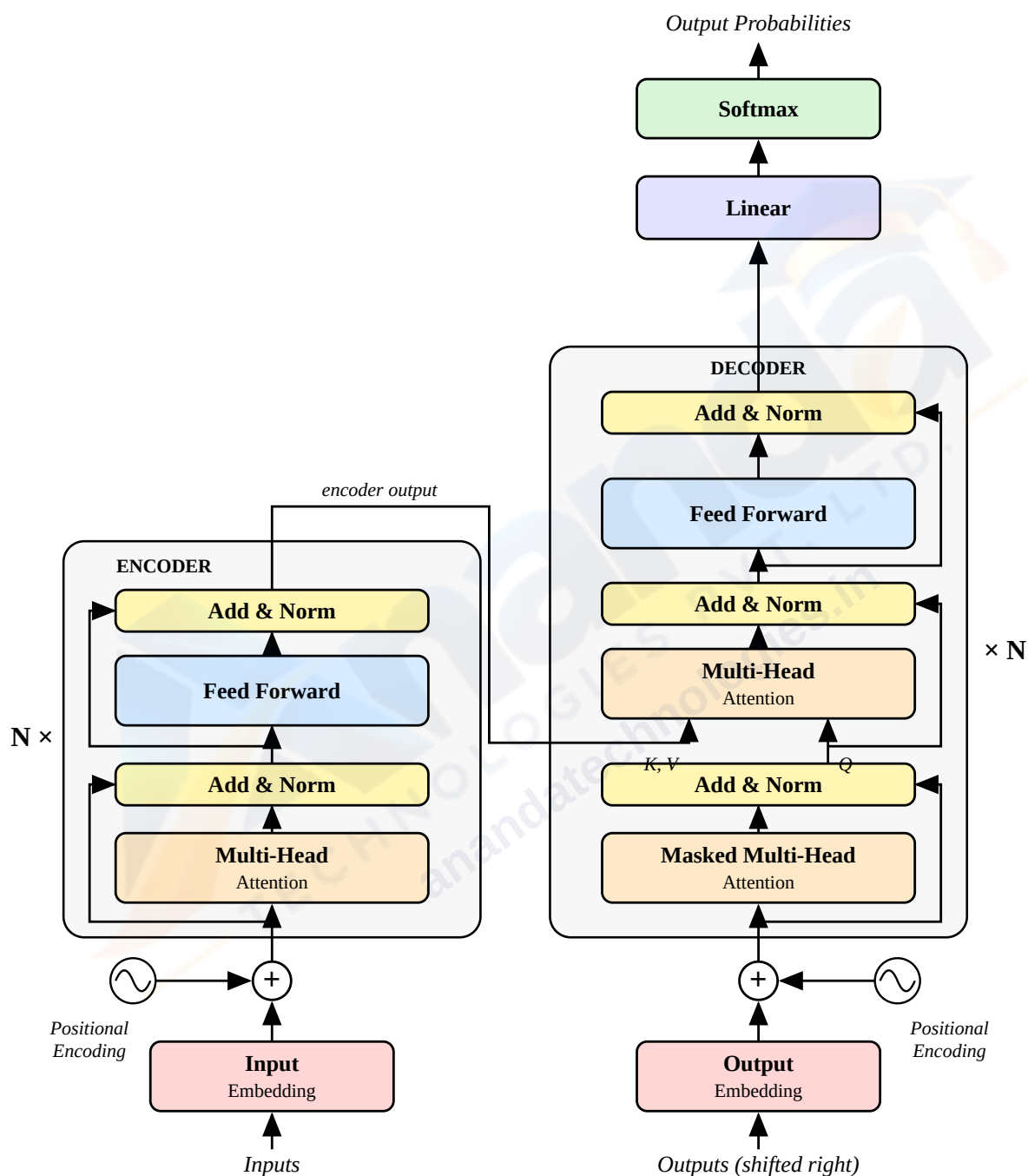


Fig.: Transformer architecture (encoder on the left, decoder on the right)

Working:

1. **Input embedding + positional encoding:** each input token is converted to a vector; sine/cosine positional encodings are added to give word-order information.

2. **Encoder (stack of N = 6 identical layers):** each layer has (i) multi-head self-attention — each word attends to all words of the input, and (ii) a position-wise feed-forward network. Each sub-layer has a residual connection followed by **Add & Norm** (layer normalization). The encoder output is a contextual representation of the input.
3. **Decoder (stack of N layers):** (i) **masked** multi-head self-attention over the outputs produced so far (future words are masked), (ii) encoder–decoder (cross) attention — Queries from the decoder, Keys and Values from the encoder output, (iii) feed-forward network; each followed by Add & Norm.
4. **Linear + Softmax:** converts the decoder output to a probability for every word of the vocabulary; the most probable word is the next output, which is fed back (shifted right) to generate the following word.

Example: English → Hindi translation: the encoder reads “I love India”, the decoder produces the Hindi words one by one.

(Diagram 3 M, explanation 3 M)

b) Python program — sentiment classification.

6 M

Steps: labelled reviews → train/test split → TF-IDF vectorization → train Multinomial Naive Bayes → evaluate accuracy → predict new text.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# small labelled dataset of product reviews (1 = positive, 0 = negative)
reviews = ["good product and good quality", "excellent quality, very happy",
           "very good battery and fast delivery", "happy with the excellent sound",
           "good value, excellent design", "fast delivery, very good packing",
           "excellent screen, happy customer", "good phone, happy with battery",
           "very happy, good sound quality", "excellent product, good price",
           "bad product and poor quality", "poor quality, very disappointed",
           "very bad battery and late delivery", "disappointed with the poor sound",
           "bad value, poor design", "late delivery, very bad packing",
           "poor screen, disappointed customer", "bad phone, disappointed with battery",
           "very poor, bad sound quality", "poor product, waste of money"]
labels = [1] * 10 + [0] * 10

X_train, X_test, y_train, y_test = train_test_split(
    reviews, labels, test_size=0.3, random_state=1, stratify=labels)
vec = TfidfVectorizer() # step 1: text -> TF-IDF feature vectors
model = MultinomialNB() # step 2: Naive Bayes classifier
model.fit(vec.fit_transform(X_train), y_train)

pred = model.predict(vec.transform(X_test)) # step 3: evaluate on unseen reviews
print("Test reviews :", len(X_test), " Accuracy:", accuracy_score(y_test, pred))

new = ["the sound quality is excellent", "poor battery, waste of money",
       "very happy with this phone"]
for r, p in zip(new, model.predict(vec.transform(new))): # step 4: use the model
    print(f"{r:32} -> {'Positive' if p == 1 else 'Negative'}")
```

Output:

```
Test reviews : 6    Accuracy: 1.0
the sound quality is excellent -> Positive
```

poor battery, waste of money -> Negative
very happy with this phone -> Positive

(The small dataset is only for demonstration; real applications train on thousands of reviews, e.g. the IMDB or Amazon reviews datasets.)

(Correct program 4 M, output 2 M)

c) Word2Vec — CBOW and Skip-gram.

6 M

Word2Vec (Mikolov et al., Google, 2013) is a shallow two-layer neural network that learns **dense word embeddings** (e.g. 100–300 numbers per word) from a large corpus, such that words used in similar contexts get similar vectors (king – man + woman $\approx$ queen). It uses a sliding context window.

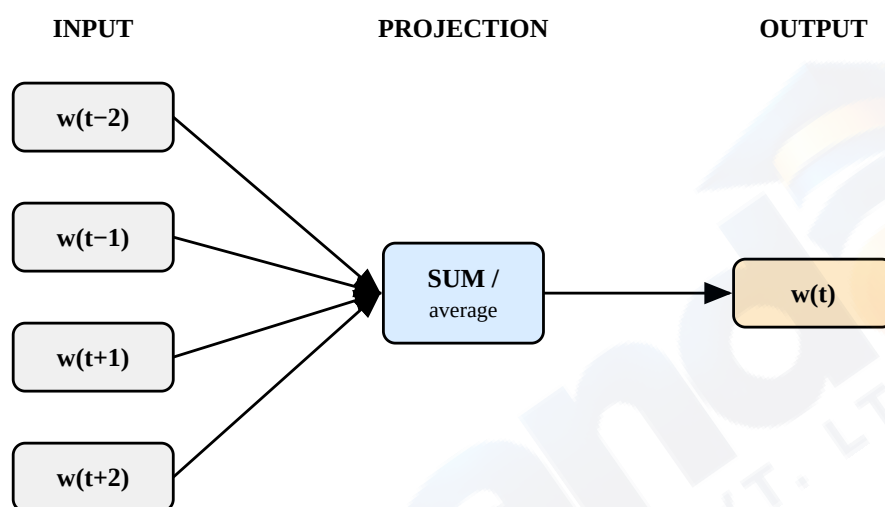


Fig.: CBOW — predicts the centre word from its context words

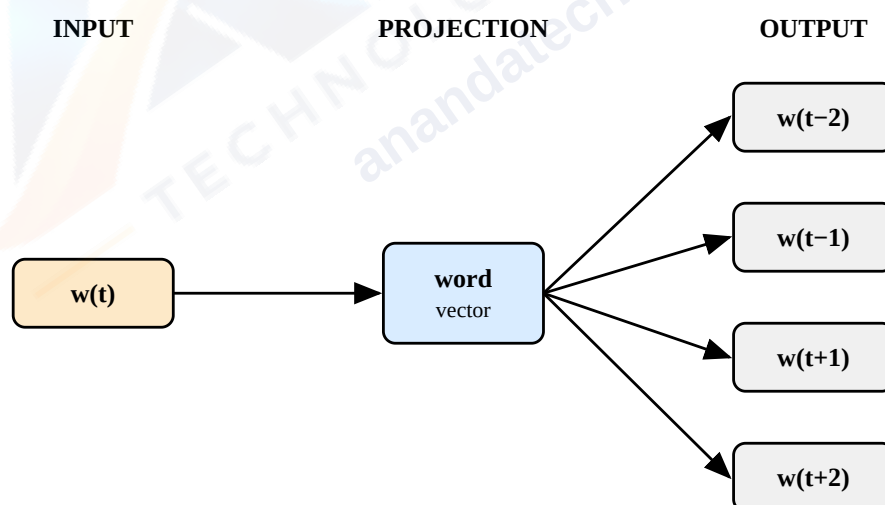


Fig.: Skip-gram — predicts the context words from the centre word

- **CBOW (Continuous Bag of Words):** input = context words around a position; their vectors are averaged in the projection layer; output = prediction of the **centre (target) word**. Faster, better

for frequent words.

- **Skip-gram:** input = the centre word; output = prediction of each **context word** in the window.
Slower but works better for small data and rare words.

Skip-gram pairs (window = 2) and a small Word2Vec model with gensim:

```
from gensim.models import Word2Vec

sentence = "students learn natural language processing".split()
window = 2
print("Skip-gram (centre -> context) pairs, window =", window)
for i, centre in enumerate(sentence):
    lo, hi = max(0, i - window), min(len(sentence), i + window + 1)
    ctx = [sentence[j] for j in range(lo, hi) if j != i]
    print(f" {centre:<11} -> {ctx}")

corpus = [s.split() for s in ["students learn natural language processing",
                              "students learn machine learning",
                              "computers process natural language text"]]
model = Word2Vec(corpus, vector_size=10, window=2, min_count=1,
                 sg=1, seed=1, workers=1, epochs=50)      # sg=1 -> skip-gram
print("Vocabulary size:", len(model.wv))
print("Vector for 'language' (10 numbers):")
print(model.wv['language'].round(3))
```

Output:

```
Skip-gram (centre -> context) pairs, window = 2
students    -> ['learn', 'natural']
learn       -> ['students', 'natural', 'language']
natural     -> ['students', 'learn', 'language', 'processing']
language    -> ['learn', 'natural', 'processing']
processing  -> ['natural', 'language']
Vocabulary size: 10
Vector for 'language' (10 numbers):
[-0.005  0.002  0.051  0.09  -0.093 -0.071  0.065  0.09  -0.05  -0.038]
```

(sg=1 selects skip-gram, sg=0 selects CBOW. Vector values depend on random initialization; on such a tiny corpus they are only illustrative.)

(Word2Vec 1 M, CBOW 2 M, skip-gram 2 M, pairs 1 M)

Note: This is a **model question paper prepared by Ananda Technologies** only for practice and exam preparation, based on the pattern of previous MSBTE question papers and the K-scheme syllabus. It is **not an official MSBTE paper**, and it does **not** mean that these questions will come in the exam. Answers follow standard textbook points; marks shown are indicative. Free for students — anandatechnologies.in