

Natural Language Processing (315329)

Model Question Paper — Set B · Model Answers

Q.1

a) Need of NLP; areas of linguistics.

2 M

Need: Most of the world's information (e-mails, social media, documents, speech) is unstructured natural language. NLP is needed so that computers can understand and process it — for human–computer interaction in natural language (chatbots, voice assistants), automatic translation, information extraction and analysis of huge text data.

Areas of linguistics (any two): phonology (sounds), morphology (word formation), syntax (sentence structure), semantics (meaning), pragmatics (meaning in context), discourse.

(Need 1 M, two areas 1 M)

b) Uses of regex; date pattern.

2 M

Uses: text cleaning (removing punctuation, HTML tags, URLs), tokenization, pattern extraction (e-mail IDs, phone numbers, dates, hashtags), input validation, search-and-replace.

Regex for dd-mm-yyyy: $^([0-9]|12)[0-9]|3([01]) - ([0-9]|1[0-2]) - [0-9]{4}$$
— day 01–31, month 01–12, 4-digit year. (28-10-2026 ✓, 32-01-2020 ✗; checked with Python.)

(Two uses 1 M, regex 1 M)

c) Tokenization.

2 M

Tokenization is the process of splitting a text into smaller units called **tokens** — sentences, words, sub-words or characters. It is usually the first step of NLP preprocessing.

Example (NLTK word_tokenize): “I can't come today.” → ['I', 'ca', 'n't', 'come', 'today', '.']

(Definition 1 M, example 1 M)

d) One-hot encoding.

2 M

One-hot encoding represents each word of a vocabulary of size V as a binary vector of length V that has **1 at the index of that word and 0 everywhere else**.

Example: vocabulary [cat, dog, rat] → cat = [1, 0, 0], dog = [0, 1, 0], rat = [0, 0, 1].

(Definition 1 M, example 1 M)

e) Polysemy and hyponymy.

2 M

Polysemy: one word with several related meanings, e.g. *head* — head of the body / head of a department / head of a coin.

Hyponymy: ‘is-a-kind-of’ relation between a specific word (hyponym) and a general word (hypernym), e.g. *mango* is a hyponym of *fruit*.

(Each 1 M)

f) Challenges of Transformers.

2 M

- **High computational cost:** self-attention cost grows with the square of the sequence length ($O(n^2)$), so long documents need a lot of memory and time.

- **Large data and resources:** pre-training needs huge corpora, GPUs and energy; models are large and slow to deploy.
- Others: limited context length, lack of interpretability (black box), bias learnt from training data.

(Any two challenges 1 M each)

g) Chunking.

2 M

Chunking (shallow or partial parsing) is the process of grouping POS-tagged words into short, non-overlapping phrases such as noun phrases (NP), verb phrases (VP) and prepositional phrases (PP), without building a complete parse tree.

Example: [NP The black cat] [VP sat] [PP on [NP the mat]].

(Definition 1 M, example 1 M)

Q.2

a) Text corpora; Brown and Penn Treebank.

4 M

A **text corpus** is a large, structured collection of texts used in NLP. **Role:** (i) training data for statistical / ML models (taggers, parsers, classifiers), (ii) testing and benchmarking, (iii) finding word frequencies, collocations and usage, (iv) annotated corpora provide gold-standard POS tags, parse trees and entities.

- **Brown Corpus:** first million-word electronic corpus of American English (compiled at Brown University in the 1960s from texts printed in 1961); 500 samples of text in 15 genres (news, fiction, religion, humor ...); POS-tagged. Used for studying genre differences and training taggers.
- **Penn Treebank:** Wall Street Journal text annotated with POS tags (the 45-tag Penn tag set) and full **syntactic parse trees**; standard corpus for training parsers. NLTK includes a 10% sample.

```
from nltk.corpus import brown, treebank

# Brown corpus: 1.16 million words, 500 texts, 15 categories (genres), POS-tagged
print("Brown categories :", brown.categories()[:6], "...")
print("Total words      :", len(brown.words()))
print("News words       :", brown.words(categories='news')[:8])
print("Tagged words      :", brown.tagged_words()[:4])

# Penn Treebank sample (Wall Street Journal): POS tags + full parse trees
print("Treebank files   :", len(treebank.fileids()))
print("Tagged sentence  :", treebank.tagged_sents()[0][:6])
print("Parse tree of a short sentence:")
print(treebank.parsed_sents('wsj_0001.mrg')[1])
```

Output:

```
Brown categories : ['adventure', 'belles_lettres', 'editorial', 'fiction', 'government',
'hobbies'] ...
Total words      : 1161192
News words       : ['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an']
Tagged words     : [('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-
TL')]
Treebank files   : 199
Tagged sentence  : [('Pierre', 'NNP'), ('Vinken', 'NNP'), ('.', '.'), ('61', 'CD'), ('years',
'NNS'), ('old', 'JJ')]
Parse tree of a short sentence:
(S
  (NP-SBJ (NP Mr.) (NP Vinken))
  (VP
    (VBZ is)
    (NP-PRD
```

```
(NP (NN chairman))
  (PP
    (IN of)
    (NP
      (NP (NNP Elsevier) (NNP N.V.))
      (, ,)
      (NP (DT the) (NNP Dutch) (VBG publishing) (NN group))))))
(. .))
```

(Role 1 M, each corpus 1½ M)

b) Porter stemmer with examples.

4 M

The **Porter stemmer** (Martin Porter, 1980) removes English suffixes using about 60 rules applied in **5 sequential steps**. A word is seen as $[C](VC)^m[V]$ (C = consonant group, V = vowel group); the **measure m** decides whether a rule can be applied (e.g. a suffix is removed only if the remaining stem has $m > 0$).

1. **Step 1a — plurals:** SSES → SS (caresses → caress), IES → I (ponies → poni), S → ‘ (cats → cat).
2. **Step 1b — -ed / -ing:** remove ED or ING if the stem has a vowel (connected → connect, hopping → hopp); then clean up: double consonant → single (hopp → hop), AT → ATE, BL → BLE.
3. **Step 1c / 2 — derivational suffixes (m > 0):** Y → I; ATIONAL → ATE, TIONAL → TION, OUSLI → OUS, IZATION → IZE ...
4. **Step 3 / 4:** ICATE → IC, FUL → ‘, NESS → ‘; then remove AL, ANCE, ENCE, ER, IC, ABLE, IBLE, ANT, MENT, ENT, OUS ... when $m > 1$.
5. **Step 5:** remove final E ($m > 1$) and change LL → L.

Output (NLTK PorterStemmer):

```
from nltk.stem import PorterStemmer, SnowballStemmer

porter = PorterStemmer()
words = ["caresses", "ponies", "connected", "hopping",
        "relational", "generously", "happiness", "conditional"]
for w in words:
    print(f"{w:<12} -> {porter.stem(w)}")
```

Output:

```
caresses      -> caress
ponies        -> poni
connected     -> connect
hopping       -> hop
relational    -> relat
generously    -> gener
happiness     -> happi
conditional   -> condit
```

The stem need not be a dictionary word (poni, gener, happi) — it is only a common base used to group related word forms.

(Working 2 M, stemmed words 2 M)

c) Semantic network (birds).

4 M

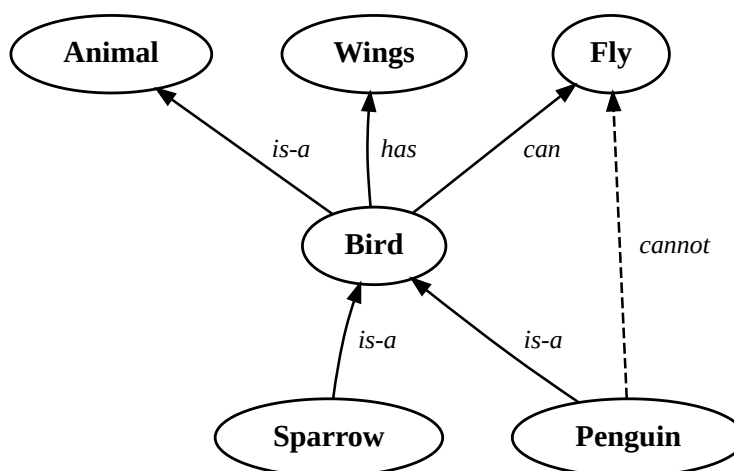


Fig.: Semantic network (dashed arc = exception stored at the lower node)

A **semantic network** stores knowledge as a labelled directed graph: nodes = concepts (Sparrow, Bird, Animal), arcs = relations (is-a, has, can).

Use — inheritance and reasoning: to answer “Can a sparrow fly?” the system follows *Sparrow* —is-a→ *Bird* —can→ *Fly* ⇒ yes; “Is a penguin an animal?” ⇒ yes through two is-a links. Properties stored at a general node are inherited by all lower nodes, unless an **exception** is stored at the lower node (Penguin —cannot→ Fly overrides the inherited property). Used in question answering, word-sense relations (WordNet) and knowledge graphs.

(Diagram 2 M, explanation 2 M)

d) Cosine similarity of two sentences (BoW).

4 M

Steps: build a common vocabulary → count each word in each sentence (BoW vector) → apply $\cos \theta = (S1 \cdot S2) / (|S1| |S2|)$.

```

import math
from sklearn.feature_extraction.text import CountVectorizer

s1 = "I like to play cricket and I like to watch cricket"
s2 = "I like to watch football"
vec = CountVectorizer(token_pattern=r"(?u)\b\w+\b") # keep one-letter word 'I'
X = vec.fit_transform([s1, s2]).toarray()
print("Vocabulary:", list(vec.get_feature_names_out()))
print("S1 vector :", X[0].tolist())
print("S2 vector :", X[1].tolist())

dot = sum(a * b for a, b in zip(X[0], X[1]))
n1 = math.sqrt(sum(a * a for a in X[0])); n2 = math.sqrt(sum(b * b for b in X[1]))
print(f"Dot product = {dot}, |S1| = {n1:.4f}, |S2| = {n2:.4f}")
print(f"Cosine similarity = {dot / (n1 * n2):.4f}")
  
```

Output:

```

Vocabulary: ['and', 'cricket', 'football', 'i', 'like', 'play', 'to', 'watch']
S1 vector : [1, 2, 0, 2, 2, 1, 2, 1]
S2 vector : [0, 0, 1, 1, 1, 0, 1, 1]
Dot product = 7, |S1| = 4.3589, |S2| = 2.2361
Cosine similarity = 0.7182
  
```

	and	cricket	football	i	like	play	to	watch
S1	1	2	0	2	2	1	2	1
S2	0	0	1	1	1	0	1	1

$S1 \cdot S2 = 0 + 0 + 0 + 2 + 2 + 0 + 2 + 1 = 7$; $|S1| = \sqrt{19} = 4.3589$; $|S2| = \sqrt{5} = 2.2361$; $\cos \theta = 7 / (4.3589 \times 2.2361) = \mathbf{0.7182}$. The sentences are fairly similar (they share 'I like to watch') but not identical in topic.

(*Vectors 2 M, calculation 2 M*)

Q.3

a) BPE algorithm with 4 merges.

4 M

Byte-Pair Encoding is a data-driven subword tokenization method. **Algorithm:**

1. Represent every word as a sequence of characters followed by an end-of-word symbol '₀'; the initial vocabulary is the set of characters.
2. Count the frequency of every pair of adjacent symbols in the corpus (each word's count is multiplied by its frequency).
3. Merge the most frequent pair into a new symbol and add it to the vocabulary.
4. Repeat steps 2–3 for k merges (k = hyper-parameter). New text is tokenized by applying the learnt merges in the same order.

Step 0 (initial split into characters, '₀' = end of word): new _ (6) · new er _ (3) · new est _ (4) · few _ (2)

Initial vocabulary = {n, e, w, _ , r, s, t, f} (8 symbols)

Merge	Most frequent pair	Freq.	New token	Corpus after merge
1	e + w	15	ew	n ew _ (6) · new er _ (3) · new est _ (4) · few _ (2)
2	n + ew	13	new	new _ (6) · new er _ (3) · new est _ (4) · few _ (2)
3	new + e	7	newe	new _ (6) · newe r _ (3) · newe st _ (4) · few _ (2)
4	new + _	6	new_	new_ (6) · newe r _ (3) · newe st _ (4) · few _ (2)

Final vocabulary (12 tokens): {n, e, w, _ , r, s, t, f, ew, new, newe, new_}

Using these merges, the unseen word *renew* is tokenized as r + e + new_ — no unknown token is needed.

(*Algorithm 2 M, merges 2 M*)

b) Dependency parsing.

4 M

Dependency parsing analyses the grammatical structure of a sentence in terms of **binary head–dependent relations** between words (no phrase nodes). Every word except the root depends on exactly one head word; each arc is labelled with a relation such as nsubj (nominal subject), dobj (direct object), amod (adjectival modifier), det (determiner). The main verb is the **root**.

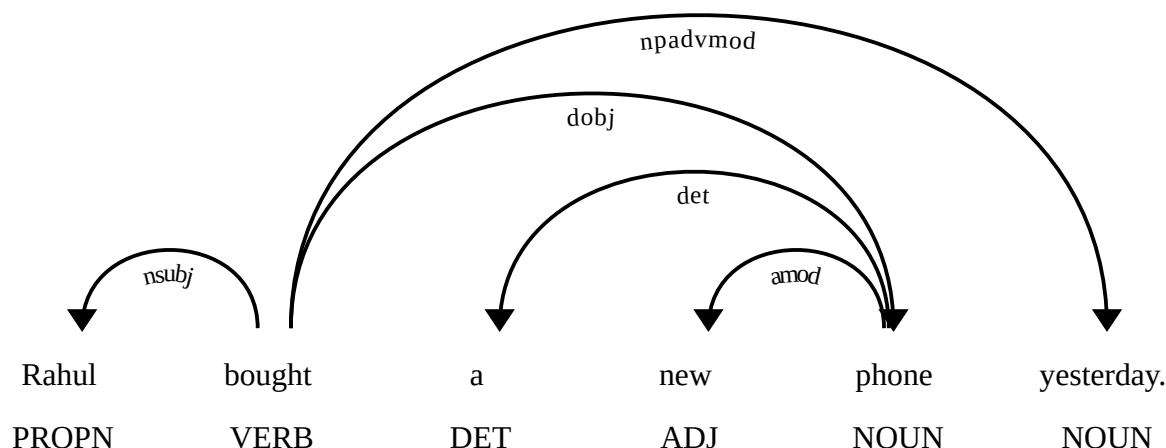


Fig.: Dependency structure produced by spaCy (arrow goes from head to dependent)

Dependent	Relation	Head
Rahul	nsubj	bought
bought	ROOT	bought
a	det	phone
new	amod	phone
phone	dobj	bought
yesterday	npadvmod	bought

Dependency parsing is useful for relation extraction (who did what to whom), question answering and for languages with free word order.

(Explanation 2 M, dependency structure 2 M)

c) Lemmatization vs stemming.

4 M

Lemmatization reduces a word to its **lemma** — the dictionary (base) form — using a vocabulary and morphological analysis and the part of speech of the word. E.g. *children* → *child*, *were* → *be*, *mice* → *mouse*, *running* → *run*.

Sr.	Point	Stemming	Lemmatization
1	Method	Removes suffixes by rules.	Uses dictionary + POS + morphology.
2	Result	May not be a real word (cheeses → chees).	Always a valid word (cheeses → cheese).
3	Irregular forms	Cannot handle (mice → mice).	Handles (mice → mouse, were → be).
4	Speed / accuracy	Fast, less accurate.	Slower, more accurate.

Example (spaCy lemmatizer vs NLTK Porter stemmer):

```
import spacy
from nltk.stem import PorterStemmer

nlp = spacy.load("en_core_web_sm")
ps = PorterStemmer()
```

```
doc = nlp("The children were running and the mice ate the cheeses")
print(f"{'Token':<10}{'POS':<7}{'Lemma (spaCy)':<15}{'Stem (Porter)'}")
for t in doc:
    print(f"{'t.text':<10}{'t.pos_':<7}{'t.lemma_':<15}{'ps.stem(t.text)'}")
```

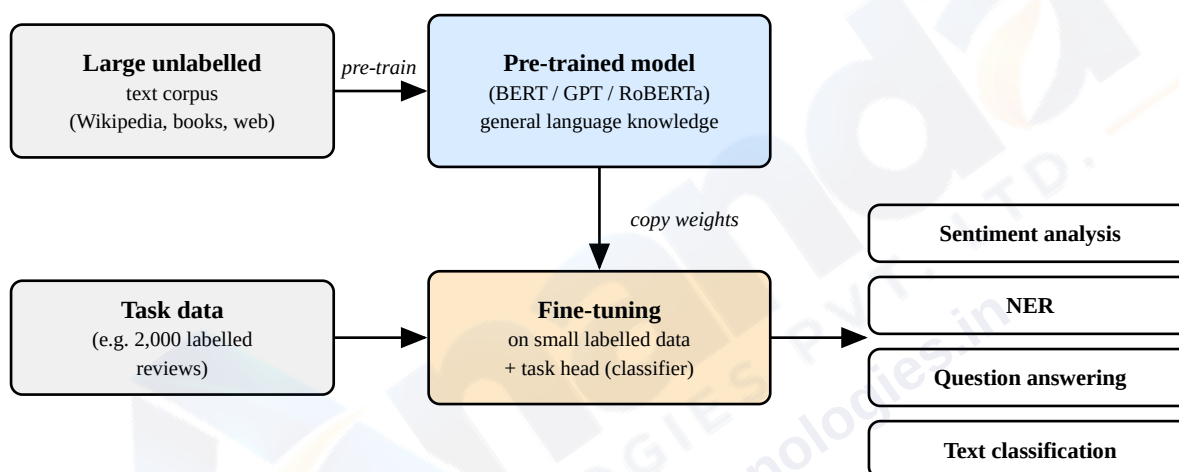
Output:

Token	POS	Lemma (spaCy)	Stem (Porter)
The	DET	the	the
children	NOUN	child	children
were	AUX	be	were
running	VERB	run	run
and	CCONJ	and	and
the	DET	the	the
mice	NOUN	mouse	mice
ate	VERB	eat	ate
the	DET	the	the
cheeses	NOUN	cheese	chees

(Lemmatization 1 M, differences 2 M, example 1 M)

d) Pre-training and fine-tuning; benefits.

4 M



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

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

- **Pre-training:** a large model (BERT, GPT) is trained once on a huge unlabelled corpus with self-supervised objectives — *masked language modelling* (predict hidden words, BERT) or *next-word prediction* (GPT). It learns general knowledge of grammar, meaning and context.
- **Fine-tuning:** the pre-trained weights are loaded, a small task-specific output layer is added (e.g. a 2-class classifier) and the whole model is trained for a few epochs on a small labelled dataset of the target task (sentiment, NER, QA).

Benefits: (1) good accuracy with little labelled data, (2) much less training time and cost, (3) reuse of one model for many tasks, (4) better generalization; ready models are available on the Hugging Face Hub.

(Pre-training 1 M, fine-tuning 1 M, benefits 2 M)

Q.4

a) Word order typology.

4 M

Word order typology classifies languages by the usual order of **Subject (S), Verb (V) and Object (O)** in a simple declarative sentence. Of the six possible orders, SOV and SVO are the most common in the world's languages.

Type	Order	Languages	Example
SVO	Subject – Verb – Object	English, Chinese, French	<i>Ram (S) eats (V) a mango (O).</i>
SOV	Subject – Object – Verb	Hindi, Marathi, Japanese, Korean	राम (S) आंबा (O) खातो (V) — “Ram mango eats”.
VSO	Verb – Subject – Object	Classical Arabic, Irish, Welsh	Arabic <i>kataba (V) al-waladu (S) ad-darsa (O)</i> — “wrote the boy the lesson”.
VOS / OVS / OSV	rare orders	Malagasy (VOS), Hixkaryana (OVS)	—

Languages with a **fixed** order (English) use position to show grammatical roles, while **free** word-order languages (Sanskrit, Hindi to some extent) use case markers. NLP systems (parsers, machine translation) must handle these differences — e.g. English → Hindi translation must move the verb to the end.

(Explanation 1 M, three types with examples 3 M)

b) BIO tagging in NER.

4 M

In NER, entities can span several tokens. The **BIO (IOB) format** labels every token: **B-TYPE** = Beginning of an entity, **I-TYPE** = Inside (continuation) of the same entity, **O** = Outside any entity. This converts NER into a token-level sequence-labelling task and keeps the boundaries of adjacent entities clear.

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Narendra Modi visited New York in September")
print("Entities:", [(e.text, e.label_) for e in doc.ents])
print("Token-level BIO tags:")
for t in doc:
    tag = t.ent_iob_ if t.ent_iob_ == "0" else f"{t.ent_iob_}-{t.ent_type_}"
    print(f" {t.text:<10} {tag}")
```

Output:

```
Entities: [('Narendra Modi', 'PERSON'), ('New York', 'GPE'), ('September', 'DATE')]
Token-level BIO tags:
Narendra B-PERSON
Modi I-PERSON
visited O
New B-GPE
York I-GPE
in O
September B-DATE
```

‘Narendra Modi’ is one PERSON entity of two tokens (B-PERSON, I-PERSON); ‘New York’ is a GPE (geo-political entity: country/city/state).

(BIO explanation 2 M, tagged example 2 M)

c) Parse tree for “A boy kicked the red ball.”

4 M

Grammar (CFG) rules used:

$S \rightarrow NP VP$

$NP \rightarrow Det N \mid Det Adj N$

$VP \rightarrow V NP$

$Det \rightarrow A \mid the \quad Adj \rightarrow red \quad N \rightarrow boy \mid ball \quad V \rightarrow kicked$

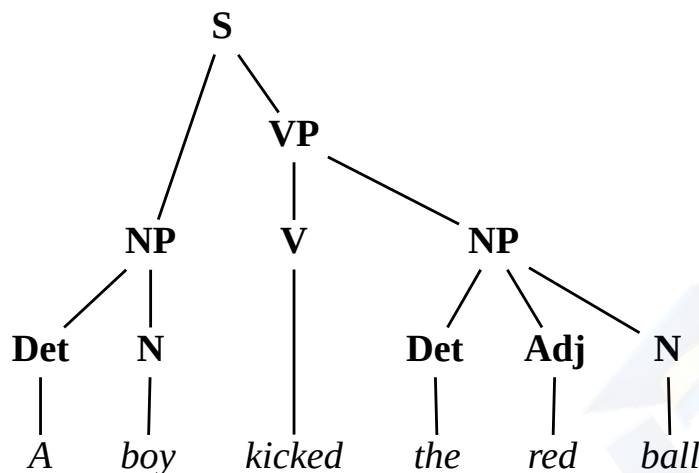


Fig.: Parse tree (NLTK ChartParser)

Bracketed form: [S [NP [Det A] [N boy]] [VP [V kicked] [NP [Det the] [Adj red] [N ball]]]]. The subject NP ‘A boy’ and the VP ‘kicked the red ball’ are the two main constituents of S; the object NP is inside the VP.

(Grammar 1 M, tree 3 M)

d) Bag-of-Words matrix.

4 M

The **Bag-of-Words** model represents each document by the **frequency of each vocabulary word**, ignoring order and grammar. Steps: tokenize all documents → build vocabulary of unique words → count each word in each document → document-term matrix.

```

from sklearn.feature_extraction.text import CountVectorizer

docs = ["the cat sat on the mat",
        "the dog sat on the log",
        "the cat chased the dog"]
vec = CountVectorizer()
X = vec.fit_transform(docs)
print("Vocabulary:", list(vec.get_feature_names_out()))
print("Bag-of-Words matrix (rows = documents):")
print(X.toarray())
  
```

Output:

```

Vocabulary: ['cat', 'chased', 'dog', 'log', 'mat', 'on', 'sat', 'the']
Bag-of-Words matrix (rows = documents):
[[1 0 0 0 1 1 1 2]
 [0 0 1 1 0 1 1 2]
 [1 1 1 0 0 0 0 2]]
  
```

Doc	cat	chased	dog	log	mat	on	sat	the
D1	1	0	0	0	1	1	1	2
D2	0	0	1	1	0	1	1	2
D3	1	1	1	0	0	0	0	2

Limitations: word order and meaning are lost ('dog bites man' = 'man bites dog'), vectors are large and sparse.

(Explanation 1 M, matrix 3 M)

e) NLTK program — NP, PP and VP chunks.

4 M

```
import nltk

sentence = "My elder brother bought a red car in Pune"
tagged = nltk.pos_tag(nltk.word_tokenize(sentence))
print(tagged)

grammar = r"""
NP: {<PRP\$>?<DT>?<JJ.*>*<NN.*>+}    # noun phrase
PP: {<IN><NP>}                          # preposition + noun phrase
VP: {<VB.*><NP|PP>+}                    # verb + its objects
"""
chunker = nltk.RegexpParser(grammar)
tree = chunker.parse(tagged)
print(tree)
for label in ("NP", "PP", "VP"):
    chunks = tree.subtrees(lambda s: s.label() == label)
    print(label, "chunks:", [" ".join(w for w, t in st.leaves()) for st in chunks])
```

Output:

```
[('My', 'PRP$'), ('elder', 'NN'), ('brother', 'NN'), ('bought', 'VBD'), ('a', 'DT'), ('red', 'JJ'), ('car', 'NN'), ('in', 'IN'), ('Pune', 'NNP')]
(S
  (NP My/PRP$ elder/NN brother/NN)
  (VP bought/VBD (NP a/DT red/JJ car/NN) (PP in/IN (NP Pune/NNP))))
NP chunks: ['My elder brother', 'a red car', 'Pune']
PP chunks: ['in Pune']
VP chunks: ['bought a red car in Pune']
```

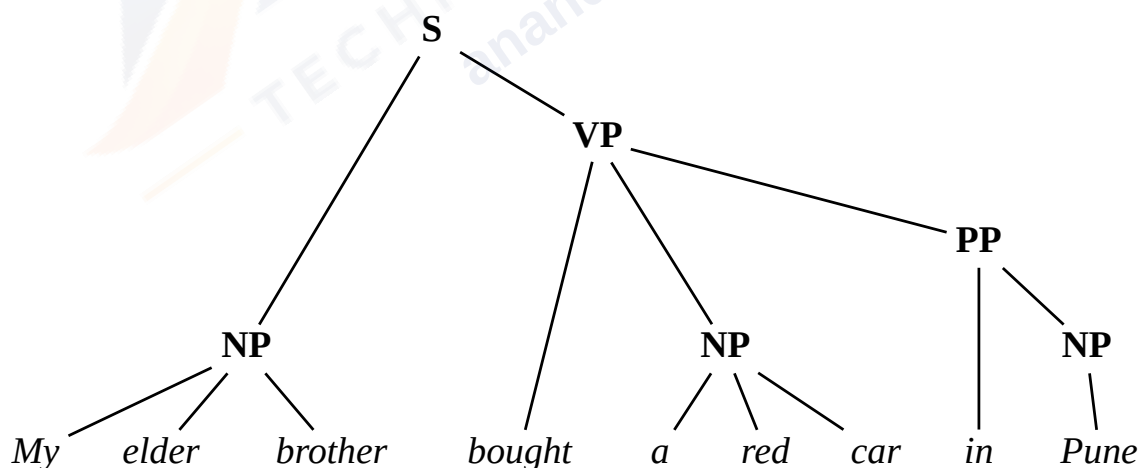


Fig.: Chunk tree

The chunk grammar is applied in stages: NP first, then PP (preposition + NP) and VP (verb + following NP/PP). (Note: the tagger tagged 'elder' as NN; it is still correctly grouped in the NP.)

Q.5

a) n-gram language model; bigram probabilities.

6 M

An **n-gram model** predicts the probability of a word from the previous $n - 1$ words (Markov assumption): $P(w_i | w_1 \dots w_{i-1}) \approx P(w_i | w_{i-n+1} \dots w_{i-1})$. Unigram $n = 1$, bigram $n = 2$, trigram $n = 3$.

Probabilities are estimated by **Maximum Likelihood Estimation** from counts: $P(w_i | w_{i-1}) = C(w_{i-1} w_i) / C(w_{i-1})$. Sentence probability = product of the bigram probabilities. Uses: next-word prediction, spelling correction, speech recognition, machine translation.

Corpus with start/end markers (lower-cased):

```
<s> i love nlp </s>
<s> i love python </s>
<s> students love nlp </s>
<s> i study nlp </s>
```

Unigram counts: $C(<s>) = 4$, $C(i) = 3$, $C(\text{love}) = 3$, $C(\text{nlp}) = 3$, $C(</s>) = 4$, $C(\text{python}) = 1$, $C(\text{students}) = 1$, $C(\text{study}) = 1$

Bigram probability	Calculation	Value
$P(i <s>)$	$C(<s> i) / C(<s>) = 3/4$	0.7500
$P(\text{students} <s>)$	$C(<s> \text{students}) / C(<s>) = 1/4$	0.2500
$P(\text{love} i)$	$C(i \text{ love}) / C(i) = 2/3$	0.6667
$P(\text{study} i)$	$C(i \text{ study}) / C(i) = 1/3$	0.3333
$P(\text{nlp} \text{love})$	$C(\text{love nlp}) / C(\text{love}) = 2/3$	0.6667
$P(\text{python} \text{love})$	$C(\text{love python}) / C(\text{love}) = 1/3$	0.3333
$P(</s> \text{nlp})$	$C(\text{nlp } </s>) / C(\text{nlp}) = 3/3$	1.0000
$P(</s> \text{python})$	$C(\text{python } </s>) / C(\text{python}) = 1/1$	1.0000
$P(\text{love} \text{students})$	$C(\text{students love}) / C(\text{students}) = 1/1$	1.0000
$P(\text{nlp} \text{study})$	$C(\text{study nlp}) / C(\text{study}) = 1/1$	1.0000

$P(<s> i \text{ love nlp } </s>) = P(i | <s>) \times P(\text{love} | i) \times P(\text{nlp} | \text{love}) \times P(</s> | \text{nlp}) = 3/4 \times 2/3 \times 2/3 \times 1 = 1/3 = 0.3333$

A sentence containing an unseen bigram (e.g. 'students study') gets probability 0; **smoothing** (e.g. add-one / Laplace) is used to avoid this.

(Explanation 2 M, bigram table 3 M, sentence probability 1 M (counts and products computed with Python))

b) Transformer architecture.

6 M

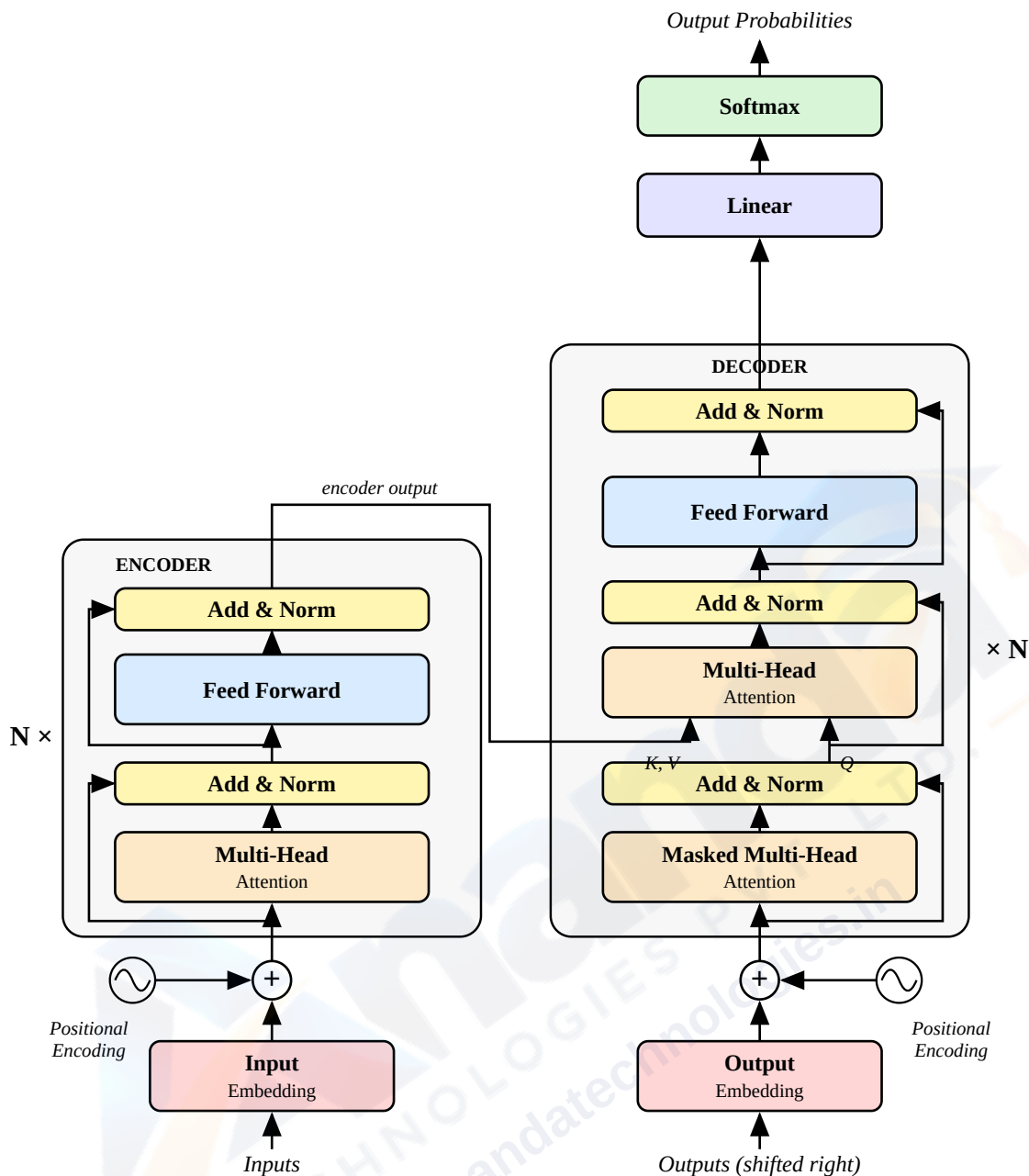


Fig.: Transformer architecture (encoder on the left, decoder on the right)

The **Transformer** is an encoder–decoder model built entirely on **self-attention**; it processes all tokens in parallel and captures long-range dependencies better than RNN/LSTM.

1. **Embedding and positional encoding:** tokens $\rightarrow$ vectors ($d = 512$ in the original model); positional encodings (sine/cosine) are added because attention itself has no notion of order.
2. **Encoder** ($N = 6$ layers): multi-head self-attention + feed-forward network, each with residual connection and layer normalization (Add & Norm).
3. **Multi-head attention:** $\text{Attention}(Q, K, V) = \text{softmax}(QK^T/\sqrt{d_k})V$ computed by 8 heads in parallel, outputs concatenated and projected.
4. **Decoder** ($N = 6$ layers): masked self-attention (cannot see future tokens), encoder–decoder attention (uses encoder output as K and V), feed-forward network.
5. **Output:** linear layer + softmax gives the probability of each vocabulary word; generation is auto-regressive (one token at a time).

How polysemy is solved: in “deposited cash in the bank” the self-attention layers mix in information from *deposited*, *cash*, while in “bank of the river” they mix in *river*; the two vectors of ‘bank’ end up in different regions of the vector space.

Example — BERT vectors of ‘bank’ in three sentences (cosine similarity):

```
import torch
from transformers import AutoTokenizer, AutoModel

tok = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")
model.eval()

def vector_of(sentence, word):
    enc = tok(sentence, return_tensors="pt")
    with torch.no_grad():
        hidden = model(**enc).last_hidden_state[0] # one 768-d vector per token
    idx = enc["input_ids"][0].tolist().index(tok.convert_tokens_to_ids(word))
    return hidden[idx]

s1 = "I deposited cash in the bank"
s2 = "The bank approved my home loan"
s3 = "We sat on the bank of the river"
v1, v2, v3 = (vector_of(s, "bank") for s in (s1, s2, s3))
cos = torch.nn.functional.cosine_similarity
print("Vector size of 'bank':", v1.shape[0])
print(f"bank(money) vs bank(loan) : {cos(v1, v2, dim=0):.3f}")
print(f"bank(money) vs bank(river): {cos(v1, v3, dim=0):.3f}")
print(f"bank(loan) vs bank(river): {cos(v2, v3, dim=0):.3f}")
```

Output:

```
Vector size of 'bank': 768
bank(money) vs bank(loan) : 0.576
bank(money) vs bank(river): 0.327
bank(loan) vs bank(river): 0.424
```

The two ‘financial’ uses are more similar to each other (higher cosine) than to the ‘river’ use — the model separates the senses automatically.

Use cases: word-sense disambiguation, question answering, NER, sentiment analysis, semantic search and chatbots, machine translation, text summarization.

(Concept 2 M, polysemy with example 2 M, use cases 2 M)

b) Fine-tuning Hugging Face Transformers on a custom dataset.

6 M

Steps:

1. **Prepare the custom dataset** with the 🤗 **Datasets** library — `Dataset.from_dict()` / `load_dataset('csv', ...)`, columns *text* and *label*; split into train/test.
2. **Tokenizer:** `AutoTokenizer.from_pretrained(name)` converts text into *input_ids* and *attention_mask* with the same vocabulary as the pre-trained model; `dataset.map()` tokenizes all rows with padding/truncation.
3. **Model:** `AutoModelForSequenceClassification.from_pretrained(name, num_labels=2)` loads the pre-trained body and adds a new classification head.
4. **Trainer API:** `TrainingArguments` (epochs, learning rate, batch size, output dir) + `Trainer(model, args, train_dataset, eval_dataset, compute_metrics)`; `trainer.train()` runs the training loop (batches, loss, back-propagation, optimizer), `trainer.evaluate()` reports the metrics.

5. **Save / use:** `trainer.save_model()` or `push_to_hub()`; predict new text.

Program (runs on CPU with a tiny 2-layer BERT; with a GPU, `distilbert-base-uncased` and a real dataset such as IMDB are used the same way):

```
from datasets import Dataset
from transformers import (AutoTokenizer, AutoModelForSequenceClassification,
                          TrainingArguments, Trainer)

import numpy as np

# 1. custom dataset (text + label), 1 = positive, 0 = negative
pos = ["excellent movie, loved it", "great acting and a good story",
       "wonderful music, very happy", "a brilliant and fun film",
       "superb direction, must watch", "very good and enjoyable",
       "loved the songs and the acting", "a great film for the family",
       "happy ending, excellent story", "good fun, wonderful cast",
       "brilliant story and great music", "enjoyable and very good acting"]
neg = ["boring movie, hated it", "poor acting and a weak story",
       "terrible music, very sad", "a dull and slow film",
       "worst direction, waste of time", "very bad and annoying",
       "hated the songs and the acting", "a poor film, too long",
       "sad ending, boring story", "bad and dull cast",
       "terrible story and poor music", "annoying and very bad acting"]
texts = pos + neg
labels = [1] * len(pos) + [0] * len(neg)
data = Dataset.from_dict({"text": texts, "label": labels})
data = data.train_test_split(test_size=0.25, seed=42)
print(data)

# 2. tokenizer of the pre-trained model
name = "google/bert_uncased_L-2_H-128_A-2" # tiny 2-layer BERT for a CPU demo
tok = AutoTokenizer.from_pretrained(name)
def encode(batch):
    return tok(batch["text"], padding="max_length", truncation=True, max_length=16)
data = data.map(encode, batched=True)

# 3. pre-trained model + new classification head (2 labels)
model = AutoModelForSequenceClassification.from_pretrained(name, num_labels=2)

def accuracy(p):
    pred = np.argmax(p.predictions, axis=1)
    return {"accuracy": float((pred == p.label_ids).mean())}

# 4. Trainer API: training arguments + train + evaluate
args = TrainingArguments(output_dir="out", num_train_epochs=30, learning_rate=5e-4,
                        per_device_train_batch_size=8, seed=42, report_to=[],
                        logging_strategy="no", save_strategy="no", disable_tqdm=True)
trainer = Trainer(model=model, args=args, train_dataset=data["train"],
                  eval_dataset=data["test"], compute_metrics=accuracy)
trainer.train()
res = trainer.evaluate()
print("Evaluation:", {k: round(res[k], 3) for k in ("eval_loss", "eval_accuracy")})

# 5. use the fine-tuned model
import torch
for s in ["what a wonderful story", "slow and boring film"]:
    logits = model(**tok(s, return_tensors="pt")).logits
    print(f"{s[:26]} -> [{'negative', 'positive'}[int(torch.argmax(logits))]]")
```

Output:

```
DatasetDict({
  train: Dataset({
    features: ['text', 'label'],
    num_rows: 18
  })
  test: Dataset({
```

```

        features: ['text', 'label'],
        num_rows: 6
    })
})
{'train_runtime': '1.508', 'train_samples_per_second': '358.1', 'train_steps_per_second':
'59.69', 'train_loss': '0.1875', 'epoch': '30'}
{'eval_loss': '0.4844', 'eval_accuracy': '0.8333', 'eval_runtime': '0.0047',
'eval_samples_per_second': '1275', 'eval_steps_per_second': '212.5', 'epoch': '30'}
Evaluation: {'eval_loss': 0.484, 'eval_accuracy': 0.833}
'what a wonderful story'    -> positive
'slow and boring film'     -> negative

```

(Steps with roles of tokenizer, Datasets and Trainer 4 M, code 2 M)

c) Text classification pipeline; evaluation metrics.

6 M

NLP application pipeline for text classification (e.g. spam detection, sentiment analysis):

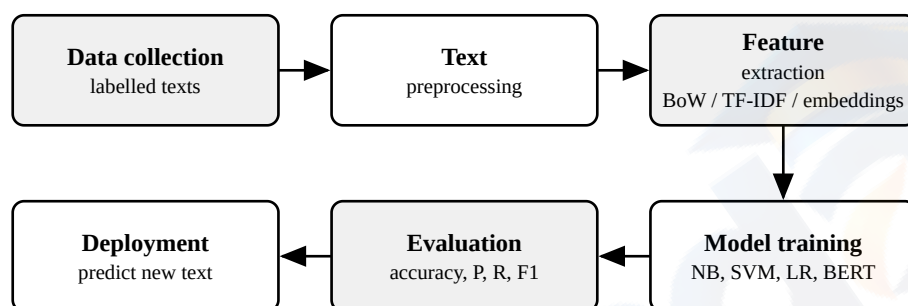


Fig.: Text classification pipeline

Evaluating a classifier — example: 12 test e-mails, 5 actually spam:

```

from sklearn.metrics import (confusion_matrix, accuracy_score, precision_score,
                             recall_score, f1_score)

# actual and predicted labels of 12 test e-mails (1 = spam, 0 = not spam)
y_true = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
y_pred = [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1]

tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
print("TP =", tp, " FP =", fp, " FN =", fn, " TN =", tn)
print("Accuracy =", round(accuracy_score(y_true, y_pred), 4))
print("Precision =", round(precision_score(y_true, y_pred), 4))
print("Recall   =", round(recall_score(y_true, y_pred), 4))
print("F1-score =", round(f1_score(y_true, y_pred), 4))

```

Output:

```

TP = 4  FP = 2  FN = 1  TN = 5
Accuracy = 0.75
Precision = 0.6667
Recall   = 0.8
F1-score = 0.7273

```

	Predicted Spam	Predicted Not spam
Actual Spam	TP = 4	FN = 1
Actual Not spam	FP = 2	TN = 5

- **Accuracy** = (TP + TN) / total = (4 + 5) / 12 = 0.75 — fraction of all predictions that are correct.

- **Precision** = $TP / (TP + FP) = 4 / 6 = 0.6667$ — of the mails predicted spam, how many are really spam.
- **Recall** = $TP / (TP + FN) = 4 / 5 = 0.80$ — of the actual spam mails, how many were caught.
- **F1-score** = $2PR / (P + R) = 2 \times 0.6667 \times 0.8 / 1.4667 = 0.7273$ — harmonic mean of precision and recall, useful for imbalanced classes.

(Pipeline 2 M, confusion matrix and four metrics 4 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

