

## Natural Language Processing (315329)

Model Question Paper — Set C · Model Answers

### Q.1

#### a) Define NLP; levels of linguistic analysis.

2 M

**NLP** is the field of AI concerned with enabling computers to understand, analyse and generate human language (text and speech).

**Levels:** (1) Phonology — sounds, (2) Morphology — structure of words, (3) Syntax — sentence structure / grammar, (4) Semantics — meaning, (5) Pragmatics — meaning in context, (6) Discourse — meaning across sentences.

*(Definition 1 M, any four levels 1 M)*

#### b) Regex for hashtags.

2 M

**Regex:** `#\w+` — ‘#’ followed by one or more word characters (letters, digits, underscore).

`re.findall(r'#\w+', tweet) → ['#IPL2026', '#CSK', '#Cricket_Fever']`

*(Correct regex 1 M, output 1 M)*

#### c) Lemmatization.

2 M

**Lemmatization** converts a word to its **lemma** (dictionary base form) using a vocabulary and morphological analysis, considering the part of speech. The result is always a valid word.

Examples (WordNet lemmatizer): geese → goose, ran (verb) → run, better (adjective) → good.

*(Definition 1 M, two examples 1 M)*

#### d) Disadvantages of BoW / TF-IDF.

2 M

- **No word order / context:** ‘not good, bad’ and ‘not bad, good’ get the same vector; meaning and semantics are lost.
- **High-dimensional, sparse vectors:** vector length = vocabulary size, mostly zeros — memory and computation heavy.
- Cannot capture synonyms (car / automobile are unrelated dimensions); unseen (out-of-vocabulary) words are ignored.

*(Any two 1 M each)*

#### e) Top-down and bottom-up parsing.

2 M

**Top-down parsing:** starts from the start symbol S and applies grammar rules to expand it downward until the words (leaves) of the sentence are produced — goal-driven (e.g. recursive-descent parser).

**Bottom-up parsing:** starts from the words of the sentence and combines them into larger constituents using the rules in reverse until S is reached — data-driven (e.g. shift-reduce parser).

*(Each 1 M)*

### f) Hugging Face dataset.

2 M

A **Hugging Face dataset** is a ready-to-use dataset hosted on the Hugging Face Hub (or created locally) that is handled by the 🧠 **Datasets** library — fast (Apache Arrow), with train/validation/test splits and methods like `map()`, `filter()`, `train_test_split()`.

```
from datasets import load_dataset
ds = load_dataset("cornell-movie-review-data/rotten_tomatoes",
split="train")
```

*(Meaning 1 M, statement 1 M)*

### g) Stop words.

2 M

**Stop words** are very common words of a language (articles, prepositions, pronouns, auxiliary verbs) that carry little meaning for tasks like search or classification, so they are often removed during preprocessing.

Examples: is, the, a, an, in, of, and, we. (NLTK's English list has 198 stop words.)

*(Definition 1 M, four examples 1 M)*

## Q.2

### a) Four real-world applications of NLP.

4 M

1. **Machine translation:** automatic translation between languages, e.g. Google Translate translating English to Marathi; uses encoder–decoder transformers.
2. **Chatbots and virtual assistants:** Siri, Alexa, customer-care bots understand user queries (intent detection, NER) and reply in natural language.
3. **Sentiment analysis:** classifies reviews, tweets and feedback as positive / negative / neutral — used by companies for brand monitoring and product feedback.
4. **Spam detection / e-mail filtering:** text classification of e-mails into spam and not spam; Gmail also uses smart reply and smart compose.
5. Others: text summarization, speech recognition (voice typing), information extraction from resumes / invoices, question answering, grammar checking (Grammarly), search engines.

*(Any four applications with explanation 1 M each)*

### b) Ambiguity and its types.

4 M

**Ambiguity** means that a language unit has more than one possible interpretation. Humans resolve it with context and world knowledge, but it is one of the main difficulties in NLP.

1. **Lexical ambiguity:** a single word has several meanings or POS. “She sat near the *bank*” (river bank / financial bank); “*book*” (noun / verb).
2. **Syntactic (structural) ambiguity:** a sentence has more than one parse tree. “I saw the man *with the telescope*” — who has the telescope?
3. **Semantic ambiguity:** the sentence structure is clear but the meaning can be interpreted in different ways. “Every student read a book” — the same book or different books?
4. **Pragmatic ambiguity:** meaning depends on the situation / speaker's intention. “Can you open the window?” — a question about ability or a request.

5. (Also **referential / anaphoric** ambiguity: “Ravi told Amit that *he* passed” — who passed?)

(Definition 1 M, types with examples 3 M)

### c) Regex: disjunction, grouping, precedence, character sets, counters.

4 M

- **Character sets / disjunction of characters [ ]**: `[Cc]` matches C or c, `[0-9]` any digit, `[^a-z]` any character except a–z (negation).
- **Disjunction (alternation) |**: `Rs|Call` matches either string; `cat|dog`.
- **Grouping ( )**: treats several characters as one unit and captures them: `(\d{3})-(\d{8})` captures STD code and number separately; `gupp(y|ies)`.
- **Precedence**: operators bind in the order parentheses → counters (\*, +, ?, { }) → sequence → disjunction |. So `ab+` = a followed by one or more b, while `(ab)+` repeats ‘ab’.
- **Counters (quantifiers)**: \* zero or more, + one or more, ? zero or one (colour?), {n} exactly n, {n,m} n to m times. Anchors ^ and \$ mark start and end.

```
import re

text = ("Colour or color? Call 9876543210 or 020-24567890. "
        "Order #A12 costs Rs.450, #B7 costs Rs.99.")
print("Disjunction [Cc]olou?r      :", re.findall(r"[Cc]olou?r", text))
print("Alternation  Rs|Call        :", re.findall(r"Rs|Call", text))
print("Char set      [0-9]{10}      :", re.findall(r"\b[0-9]{10}\b", text))
print("Grouping      (\d{3})-(\d{8}) :", re.findall(r"(\d{3})-(\d{8})", text))
print("Counter +      #[A-Z]\d+      :", re.findall(r"#[A-Z]\d+", text))
print("Counter *      Rs\\.\\d*       :", re.findall(r"Rs\\.\\d*", text))
print("Precedence     ab+ vs (ab)+    :", re.findall(r"ab+", "abbb abab"),
      re.findall(r"(?:ab)+", "abbb abab"))
```

### Output:

```
Disjunction [Cc]olou?r      : ['Colour', 'color']
Alternation  Rs|Call        : ['Call', 'Rs', 'Rs']
Char set      [0-9]{10}      : ['9876543210']
Grouping      (\d{3})-(\d{8}) : [('020', '24567890')]
Counter +      #[A-Z]\d+      : ['#A12', '#B7']
Counter *      Rs\\.\\d*       : ['Rs.450', 'Rs.99']
Precedence     ab+ vs (ab)+    : ['abbb', 'ab', 'ab'] ['ab', 'abab']
```

(Each concept with example ~1 M)

### d) Semantic network (Rohan).

4 M

A **semantic network** represents knowledge as a graph of **concepts (nodes)** joined by **labelled relations (arcs)** such as is-a, has, part-of. Properties of a general concept are inherited by its sub-concepts and instances.

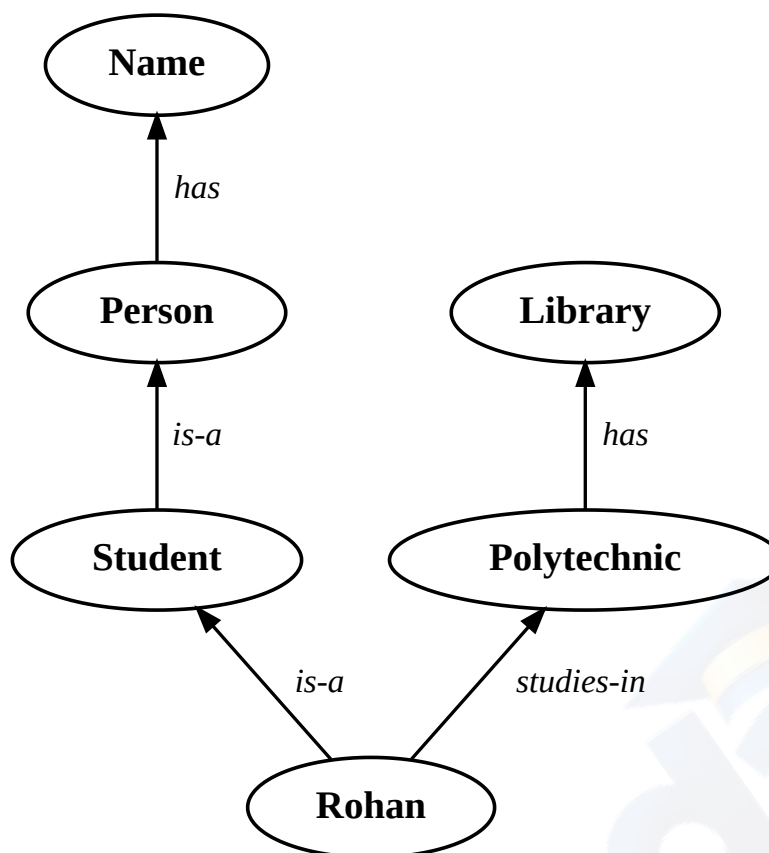


Fig.: Semantic network for the given statements

**Inference:** from Rohan —is-a→ Student —is-a→ Person —has→ Name, the system concludes “Rohan is a person and has a name” although it is not stated directly. Semantic networks are the basis of WordNet and knowledge graphs.

(Explanation 2 M, diagram 2 M)

### Q.3

#### a) Rule-based tokenization using spaCy.

4 M

spaCy's tokenizer is **rule-based** (not learnt). It first splits the text on whitespace, then processes each substring from left to right:

1. **Tokenizer exceptions (special cases):** checks whether the substring matches a special-case rule, e.g. “Let's” → “Let” + “s”, “U.K.” kept as one token, “don't” → “do” + “n't”.
2. **Prefixes:** splits off leading characters such as \$, (, “ — “\$50” → “\$” + “50”.
3. **Suffixes:** splits off trailing characters such as , . ! ) ’ ’ 's.
4. **Infexes:** splits inside a word at hyphens, “/”, “...”, e.g. “hello-world”.
5. Remaining text becomes a token; URLs, e-mails and numbers like 1,200 are kept intact.

New rules can be added with `nlp.tokenizer.add_special_case()`;

`nlp.tokenizer.explain()` shows which rule produced each token.

```

import spacy
from spacy.symbols import ORTH

nlp = spacy.load("en_core_web_sm")
text = "Let's visit the U.K. in 2026, it costs $1,200!"
  
```

```
doc = nlp(text)
print("Tokens:", [t.text for t in doc])

# which rule produced each token?
for rule, token_text in nlp.tokenizer.explain("Let's pay $50!"):
    print(f" {token_text:<5} <- {rule}")

# add a special-case rule: split 'gimme' into two tokens
nlp.tokenizer.add_special_case("gimme", [{ORTH: "gim"}, {ORTH: "me"}])
print("Special case:", [t.text for t in nlp("gimme the notes")])
```

## Output:

```
Tokens: ['Let', "'s", 'visit', 'the', 'U.K.', 'in', '2026', ',', 'it', 'costs', '$',
'1,200', '!']
Let    <- SPECIAL-1
's     <- SPECIAL-2
pay    <- TOKEN
$      <- PREFIX
50     <- TOKEN
!      <- SUFFIX
Special case: ['gim', 'me', 'the', 'notes']
```

(Explanation 2 M, example with output 2 M)

## b) Byte-Pair Encoding with 5 merges.

4 M

**BPE** builds a subword vocabulary by starting from single characters and repeatedly **merging the most frequent pair of adjacent symbols**. Frequent words become one token, rare words are broken into meaningful subwords (no out-of-vocabulary words).

**Step 0 (initial split into characters, '\_' = end of word):** h o p e \_ (8) · h o p e d \_ (4) · h o p i n g \_ (6) · r o p e \_ (7)

Initial vocabulary = {h, o, p, e, \_, d, i, n, g, r} (10 symbols)

| Merge | Most frequent pair | Freq. | New token | Corpus after merge                                                  |
|-------|--------------------|-------|-----------|---------------------------------------------------------------------|
| 1     | o + p              | 25    | op        | h o p e _ (8) · h o p e d _ (4) · h o p i n g _ (6) · r o p e _ (7) |
| 2     | op + e             | 19    | ope       | h o p e _ (8) · h o p e d _ (4) · h o p i n g _ (6) · r o p e _ (7) |
| 3     | ope + _            | 15    | ope_      | h o p e _ (8) · h o p e d _ (4) · h o p i n g _ (6) · r o p e _ (7) |
| 4     | h + ope_           | 8     | hope_     | h o p e _ (8) · h o p e d _ (4) · h o p i n g _ (6) · r o p e _ (7) |
| 5     | r + ope_           | 7     | rope_     | h o p e _ (8) · h o p e d _ (4) · h o p i n g _ (6) · r o p e _ (7) |

**Final vocabulary** (15 tokens): {h, o, p, e, \_, d, i, n, g, r, op, ope, ope\_, hope\_, rope\_}

New word: *roping* → r + op + i + n + g + \_.

**BPE in real models** — GPT-2 tokenizer (byte-level BPE, 'Ġ' marks a leading space):

```
from transformers import AutoTokenizer
```

```
tok = AutoTokenizer.from_pretrained("gpt2") # byte-level BPE tokenizer
for text in ["tokenization", "unhappiness", "ChatGPT is amazing"]:
    print(f"{text!r:22} -> {tok.tokenize(text)}")
print("Vocabulary size:", tok.vocab_size)
```

### Output:

```
'tokenization'      -> ['token', 'ization']
'unhappiness'       -> ['un', 'h', 'appiness']
'ChatGPT is amazing' -> ['Chat', 'G', 'PT', 'Gis', 'Gamazoning']
Vocabulary size: 50257
```

(Explanation 2 M, merges 2 M)

### c) Bag-of-Words vs Word2Vec.

4 M

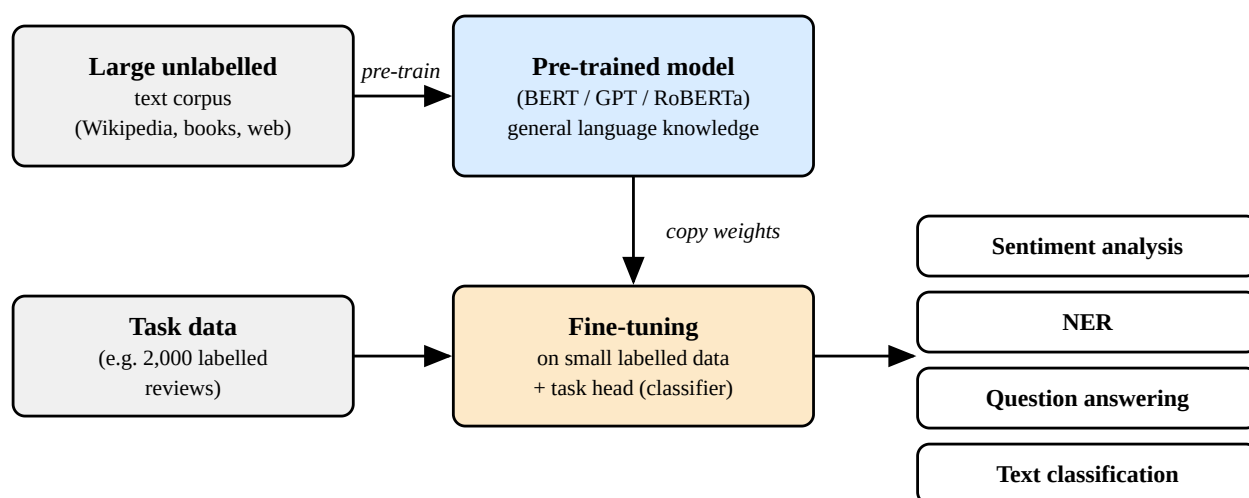
| Sr. | Point                | Bag-of-Words                                                     | Word2Vec                                                                      |
|-----|----------------------|------------------------------------------------------------------|-------------------------------------------------------------------------------|
| 1   | Type                 | Count-based, sparse representation of a document.                | Prediction-based (neural network) dense embedding of each word.               |
| 2   | Vector size          | Equal to vocabulary size (thousands), mostly zeros.              | Small fixed size (e.g. 100–300), all values non-zero.                         |
| 3   | Meaning / similarity | No semantic meaning; 'good' and 'nice' are unrelated dimensions. | Similar words get close vectors (cosine similarity high); captures analogies. |
| 4   | Context and order    | Ignores word order and context.                                  | Learns from neighbouring words in a context window (CBOW / Skip-gram).        |
| 5   | Training             | No training — just counting.                                     | Needs training on a large corpus.                                             |
| 6   | Use                  | Simple text classification, spam filtering.                      | Input features for deep learning, similarity, clustering.                     |

(Any four points 1 M each)

### d) Transfer learning; reusing BERT.

4 M

**Transfer learning** means using the knowledge learnt by a model on a large source task to improve learning on a different target task with less data.



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

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

**Reusing BERT for a new task (e.g. classifying complaints):** (1) load pre-trained BERT weights (from\_pretrained('bert-base-uncased')), (2) add a small task head — e.g. a linear + softmax layer on the [CLS] token output, (3) fine-tune on the labelled complaint data for 2–4 epochs with a small learning rate, (4) evaluate and deploy. Alternatively BERT can be frozen and used only as a feature extractor.

**Benefits:** less labelled data and training time, higher accuracy, reuse across tasks.

(Concept 1 M, diagram 1 M, steps 2 M)

## Q.4

### a) n-gram vectors and types.

4 M

An **n-gram** is a contiguous sequence of  $n$  items (words or characters) from a text. **n-gram vectors** represent a document by the counts (or TF-IDF) of its  $n$ -grams, so some local word order is kept ('not good' becomes one feature).

- **Unigram ( $n = 1$ ):** single words.
- **Bigram ( $n = 2$ ):** pairs of consecutive words.
- **Trigram ( $n = 3$ ):** three consecutive words. A text of  $T$  tokens has  $T - n + 1$   $n$ -grams.

```

from nltk import ngrams

text = "Pune is the cultural capital of Maharashtra"
tokens = text.split()
T = len(tokens)
for n in (1, 2, 3):
    grams = list(ngrams(tokens, n))
    print(f"{n}-grams (T-n+1 = {T}-{n}+1 = {len(grams)}):")
    print(" ", [" ".join(g) for g in grams])
  
```

### Output:

```

1-grams (T-n+1 = 7-1+1 = 7):
['Pune', 'is', 'the', 'cultural', 'capital', 'of', 'Maharashtra']
  
```

2-grams ( $T-n+1 = 7-2+1 = 6$ ):

['Pune is', 'is the', 'the cultural', 'cultural capital', 'capital of', 'of Maharashtra']

3-grams ( $T-n+1 = 7-3+1 = 5$ ):

['Pune is the', 'is the cultural', 'the cultural capital', 'cultural capital of', 'capital of Maharashtra']

In scikit-learn, `CountVectorizer(ngram_range=(1, 2))` builds unigram + bigram vectors.

(Explanation and types 2 M, generated n-grams 2 M)

## b) Minimum edit distance.

4 M

**Minimum edit distance (Levenshtein distance)** is the minimum number of insertions, deletions and substitutions needed to change one string into another. It is computed by dynamic programming:

$$D(i, 0) = i, D(0, j) = j; D(i, j) = \min \{ D(i-1, j) + 1 \text{ (delete)}, D(i, j-1) + 1 \text{ (insert)}, D(i-1, j-1) + (0 \text{ if } s_i = t_j \text{ else } 1) \text{ (substitute)} \}$$

Source  $s = \text{algorithm}$  (rows), target  $t = \text{logarithm}$  (columns):

|   | # | l | o | g | a | r | i | t | h | m |
|---|---|---|---|---|---|---|---|---|---|---|
| # | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| a | 1 | 1 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |
| l | 2 | 1 | 2 | 3 | 4 | 4 | 5 | 6 | 7 | 8 |
| g | 3 | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| o | 4 | 3 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |
| r | 5 | 4 | 3 | 3 | 4 | 3 | 4 | 5 | 6 | 7 |
| i | 6 | 5 | 4 | 4 | 4 | 4 | 3 | 4 | 5 | 6 |
| t | 7 | 6 | 5 | 5 | 5 | 5 | 4 | 3 | 4 | 5 |
| h | 8 | 7 | 6 | 6 | 6 | 6 | 5 | 4 | 3 | 4 |
| m | 9 | 8 | 7 | 7 | 7 | 7 | 6 | 5 | 4 | 3 |

**Minimum edit distance = 3** (bottom-right cell; verified with `nltk.edit_distance`).

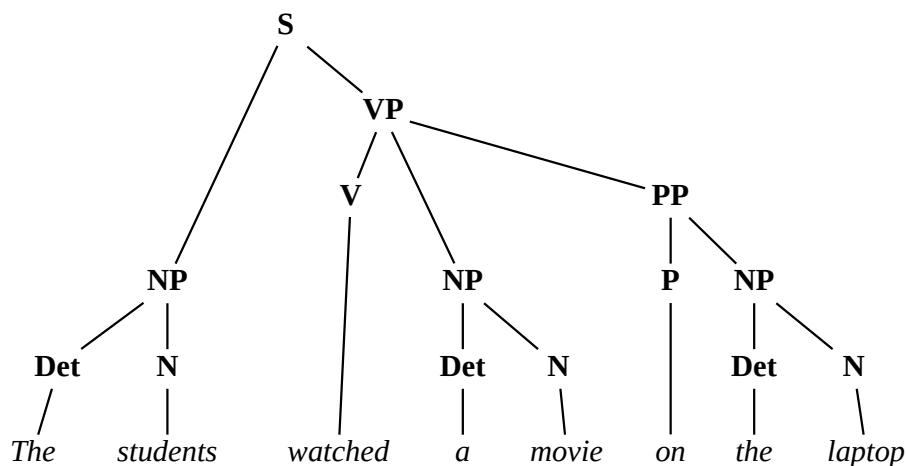
Alignment from the backtrace: substitute  $a \rightarrow l$ , substitute  $l \rightarrow o$ , substitute  $o \rightarrow a$  — the remaining letters 'g' and 'rithm' match. Used in spelling correction and string matching.

(Method 1 M, table 2 M, answer 1 M)

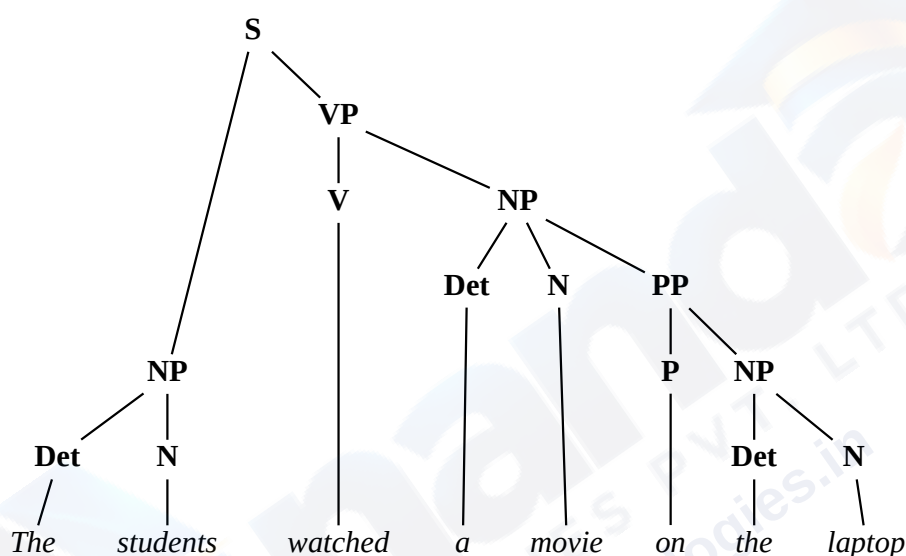
## c) Syntax trees — ambiguous sentence.

4 M

Grammar:  $S \rightarrow NP VP$ ;  $NP \rightarrow Det N \mid Det N PP$ ;  $VP \rightarrow V NP \mid V NP PP$ ;  $PP \rightarrow P NP$ . The NLTK chart parser finds **two** parse trees, so the sentence is **structurally (syntactically) ambiguous** — the PP “on the laptop” can attach to the verb or to the noun:



(a) PP attached to VP: they used the laptop to watch



(b) PP attached to NP: the movie that was on the laptop

Meaning (a) is the natural one; a parser needs probabilities (PCFG) or semantic knowledge to choose it. This is called the **PP-attachment problem**.

*(Two trees 3 M, explanation of ambiguity 1 M)*

#### d) One-hot encoding; limitations.

4 M

**One-hot encoding** builds a vocabulary of  $V$  unique words and represents each word as a  $V$ -length vector with a single 1 at its own index and 0 elsewhere. A sentence is represented as the sequence (matrix) of its word vectors.

```
import numpy as np

sentence = "I like NLP and I like AI"
words = sentence.lower().split()
vocab = sorted(set(words))
index = {w: i for i, w in enumerate(vocab)}
print("Vocabulary:", vocab)

onehot = np.zeros((len(words), len(vocab)), dtype=int)
for pos, w in enumerate(words):
    onehot[pos, index[w]] = 1
```

```
for w, row in zip(words, onehot):
    print(f"{w:<5} {row}")
```

## Output:

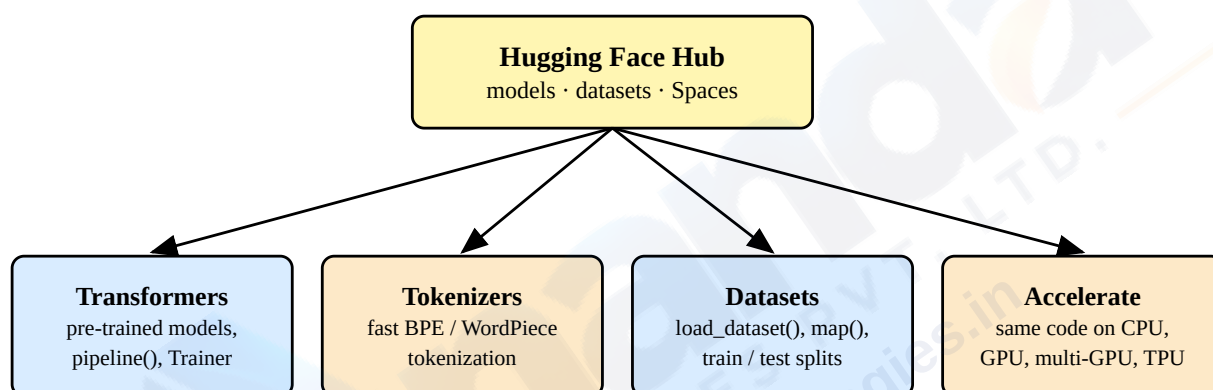
```
Vocabulary: ['ai', 'and', 'i', 'like', 'nlp']
i      [0 0 1 0 0]
like   [0 0 0 1 0]
nlp    [0 0 0 0 1]
and    [0 1 0 0 0]
i      [0 0 1 0 0]
like   [0 0 0 1 0]
ai     [1 0 0 0 0]
```

**Limitations:** (1) vector length grows with vocabulary (sparse, memory-heavy); (2) all words are equally distant — no similarity ('like' and 'love' are orthogonal); (3) no meaning or context; (4) cannot represent new (OOV) words.

(Explanation 1 M, vectors 2 M, limitations 1 M)

## e) Hugging Face ecosystem; pipeline().

4 M



*from\_pretrained() downloads from the Hub · push\_to\_hub() shares back*

Fig.: Hugging Face ecosystem

- **Transformers:** library of thousands of pre-trained models (BERT, GPT-2, T5) with `pipeline()`, Auto classes and the Trainer API.
- **Hub:** online repository of models, datasets and Spaces (demo apps), with model cards and versioning.
- **Tokenizers:** very fast (Rust) implementations of BPE, WordPiece, Unigram tokenizers.
- **Datasets:** load and process datasets efficiently (`load_dataset`, `map`).
- **Accelerate:** runs the same PyTorch training code on CPU, single/multi-GPU or TPU, with mixed precision, with very small code changes.

```
from transformers import pipeline
from datasets import load_dataset

# pipeline(): pre-trained model + tokenizer downloaded from the Hub in one line
clf = pipeline("sentiment-analysis",
               model="distilbert-base-uncased-finetuned-sst-2-english")
for r in clf(["The new metro has made travel so easy!",
             "The exam hall was hot and noisy."]):
```

```
print(r["label"], round(r["score"], 4))

# Datasets: load a ready dataset from the Hub
ds = load_dataset("cornell-movie-review-data/rotten_tomatoes", split="train")
print(ds)
print(ds[0])
```

## Output:

```
POSITIVE 0.9993
NEGATIVE 0.9986
Dataset({
  features: ['text', 'label'],
  num_rows: 8530
})
{'text': 'the rock is destined to be the 21st century\'s new " conan " and that he\'s going to make a splash even greater than arnold schwarzenegger , jean-claud van damme or steven segal .', 'label': 1}
```

(Components 2 M, example 2 M)

## Q.5

### a) Stemming; Porter vs Snowball vs Lancaster.

6 M

**Stemming** is the process of reducing inflected / derived words to a common **stem** by removing affixes with rules, so that related forms (connect, connected, connecting) are treated as one feature. The stem need not be a dictionary word.

| Sr. | Point          | Porter                                 | Snowball (Porter2)                          | Lancaster (Paice–Husk)                        |
|-----|----------------|----------------------------------------|---------------------------------------------|-----------------------------------------------|
| 1   | Developed      | Martin Porter, 1980                    | Martin Porter, 2001 (improved Porter)       | Chris Paice, Lancaster University, 1990       |
| 2   | Rules          | 5 steps of suffix rules with measure m | Refined rules; framework for many languages | About 120 iterative rules, applied repeatedly |
| 3   | Aggressiveness | Moderate                               | Slightly less aggressive, more accurate     | Very aggressive (over-stemming)               |
| 4   | Languages      | English only                           | English + ~15 languages                     | English                                       |
| 5   | Speed / use    | Most widely used, standard             | Recommended over Porter                     | Fast, short stems, less readable              |

### Comparison of output (NLTK):

```
from nltk.stem import PorterStemmer, SnowballStemmer, LancasterStemmer

porter, snow, lanc = PorterStemmer(), SnowballStemmer("english"), LancasterStemmer()
words = ["running", "happily", "generously", "fairly", "studies", "organization", "cats"]
print(f"{'Word':<14}{'Porter':<12}{'Snowball':<12}{'Lancaster'}")
for w in words:
    print(f"{w:<14}{porter.stem(w):<12}{snow.stem(w):<12}{lanc.stem(w)}")
```

## Output:

| Word    | Porter  | Snowball | Lancaster |
|---------|---------|----------|-----------|
| running | run     | run      | run       |
| happily | happili | happili  | happy     |

|              |        |          |       |
|--------------|--------|----------|-------|
| generously   | gener  | generous | gen   |
| fairly       | fairli | fair     | fair  |
| studies      | studi  | studi    | study |
| organization | organ  | organ    | org   |
| cats         | cat    | cat      | cat   |

Lancaster over-stems (organization → org, generously → gen); Snowball fixes some Porter results (generously → generous, fairly → fair).

(Definition 1 M, comparison 3 M, example 2 M)

## b) TF-IDF and cosine similarity.

6 M

**TF-IDF(t, d) = TF(t, d) × IDF(t)**, where  $TF(t, d) = (\text{count of } t \text{ in } d) / (\text{number of words in } d)$  and  $IDF(t) = \log_{10}(N / df(t))$ ,  $N$  = number of documents,  $df(t)$  = number of documents containing  $t$ . Common words in many documents get low weight; rare, document-specific words get high weight.

### Documents after lower-casing and tokenizing:

| Doc | Tokens                   | No. of words |
|-----|--------------------------|--------------|
| D1  | ai is the future         | 4            |
| D2  | ai and ml are the future | 6            |
| D3  | ml needs data            | 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      |
|--------|---------------|---------------|---------------|
| ai     | $1/4 = 0.250$ | $1/6 = 0.167$ | 0             |
| is     | $1/4 = 0.250$ | 0             | 0             |
| the    | $1/4 = 0.250$ | $1/6 = 0.167$ | 0             |
| future | $1/4 = 0.250$ | $1/6 = 0.167$ | 0             |
| and    | 0             | $1/6 = 0.167$ | 0             |
| ml     | 0             | $1/6 = 0.167$ | $1/3 = 0.333$ |
| are    | 0             | $1/6 = 0.167$ | 0             |
| needs  | 0             | 0             | $1/3 = 0.333$ |
| data   | 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 = $\log_{10}(N/df)$ |
|--------|-------|--------|-------------------------|
| ai     | 2     | 1.5    | 0.1761                  |
| is     | 1     | 3      | 0.4771                  |
| the    | 2     | 1.5    | 0.1761                  |
| future | 2     | 1.5    | 0.1761                  |
| and    | 1     | 3      | 0.4771                  |
| ml     | 2     | 1.5    | 0.1761                  |
| are    | 1     | 3      | 0.4771                  |
| needs  | 1     | 3      | 0.4771                  |
| data   | 1     | 3      | 0.4771                  |

### Step 3 — TF-IDF(t, d) = TF × IDF

| Term   | D1     | D2     | D3     |
|--------|--------|--------|--------|
| ai     | 0.0440 | 0.0293 | 0.0000 |
| is     | 0.1193 | 0.0000 | 0.0000 |
| the    | 0.0440 | 0.0293 | 0.0000 |
| future | 0.0440 | 0.0293 | 0.0000 |
| and    | 0.0000 | 0.0795 | 0.0000 |
| ml     | 0.0000 | 0.0293 | 0.0587 |
| are    | 0.0000 | 0.0795 | 0.0000 |
| needs  | 0.0000 | 0.0000 | 0.1590 |
| data   | 0.0000 | 0.0000 | 0.1590 |

**Cosine similarity between D1 and D2** (TF-IDF vectors over the terms in the order above, rounded to 4 decimals):

| i | D1 <sub>i</sub> | D2 <sub>i</sub> | D1 <sub>i</sub> × D2 <sub>i</sub> | D1 <sub>i</sub> <sup>2</sup> | D2 <sub>i</sub> <sup>2</sup> |
|---|-----------------|-----------------|-----------------------------------|------------------------------|------------------------------|
| 1 | 0.044           | 0.0293          | 0.001289                          | 0.001936                     | 0.000858                     |
| 2 | 0.1193          | 0               | 0                                 | 0.014232                     | 0                            |
| 3 | 0.044           | 0.0293          | 0.001289                          | 0.001936                     | 0.000858                     |
| 4 | 0.044           | 0.0293          | 0.001289                          | 0.001936                     | 0.000858                     |
| 5 | 0               | 0.0795          | 0                                 | 0                            | 0.00632                      |
| 6 | 0               | 0.0293          | 0                                 | 0                            | 0.000858                     |
| 7 | 0               | 0.0795          | 0                                 | 0                            | 0.00632                      |
| 8 | 0               | 0               | 0                                 | 0                            | 0                            |
| 9 | 0               | 0               | 0                                 | 0                            | 0                            |
| Σ |                 |                 | 0.003868                          | 0.02004                      | 0.016074                     |

$D1 \cdot D2 = (0.044 \times 0.0293) + (0.044 \times 0.0293) + (0.044 \times 0.0293) = \mathbf{0.003868}$  (terms with a zero are omitted)

$|D1| = \sqrt{0.02004} = 0.1416$  ;  $|D2| = \sqrt{0.016074} = 0.1268$

$\cos \theta = (D1 \cdot D2) / (|D1| \times |D2|) = 0.003868 / (0.1416 \times 0.1268) = 0.003868 / 0.0179 = \mathbf{0.2155} \Rightarrow \theta = \cos^{-1}(0.2155) \approx 77.6^\circ$

Similarity = **0.2155**: D1 and D2 share 'ai', 'the', 'future', but these are in 2 of 3 documents and get low IDF, while the unique words (is / and, ml, are) dominate — so the documents are only moderately similar.

**Disadvantages of TF-IDF**: ignores word order and meaning, sparse high-dimensional vectors, cannot handle synonyms or new words.

*(Formula 1 M, TF-IDF table 3 M, cosine 2 M (computed with Python))*

### c) NER with spaCy program.

6 M

**Named Entity Recognition (NER)** is an information-extraction task that finds and classifies **named entities** in text into categories such as PERSON, ORG (organization), GPE (country, city, state), LOC, DATE, TIME, MONEY, PERCENT. It is treated as a **sequence-labelling** problem: each token gets a tag in **BIO** format — B-X (beginning of entity X), I-X (inside), O (outside).

**Approaches**: rule-based (gazetteers, regex), statistical (HMM, CRF), and deep learning (BiLSTM-CRF, transformers — spaCy, BERT). **Uses**: resume parsing, news categorization, chatbots, search, extracting data from invoices.

```
import spacy

nlp = spacy.load("en_core_web_sm")
text = ("Sundar Pichai is the CEO of Google. "
        "Lata Mangeshkar was born in Indore on 28 September 1929.")
doc = nlp(text)
for ent in doc.ents:
    print(f"{ent.text:<20} {ent.label_:<8} {spacy.explain(ent.label_)}")
print([(t.text, t.ent_iob_ + ('-' + t.ent_type_ if t.ent_type_ else '')) for t in doc[:8]])
```

### Output:

|                                                                                                                                         |        |                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------|--------|-----------------------------------------|
| Sundar Pichai                                                                                                                           | PERSON | People, including fictional             |
| Google                                                                                                                                  | ORG    | Companies, agencies, institutions, etc. |
| Lata Mangeshkar                                                                                                                         | PERSON | People, including fictional             |
| 28 September 1929                                                                                                                       | DATE   | Absolute or relative dates or periods   |
| [('Sundar', 'B-PERSON'), ('Pichai', 'I-PERSON'), ('is', 'O'), ('the', 'O'), ('CEO', 'O'), ('of', 'O'), ('Google', 'B-ORG'), ('.', 'O')] |        |                                         |

Note: the small spaCy model missed 'Indore' (should be GPE) — statistical NER models can miss entities; larger models (en\_core\_web\_trf) or fine-tuning on Indian names/places improve this.

*(Explanation 2 M, program 3 M, output 1 M)*

## Q.6

a) Transformer architecture; multi-head attention; positional encoding.

6 M

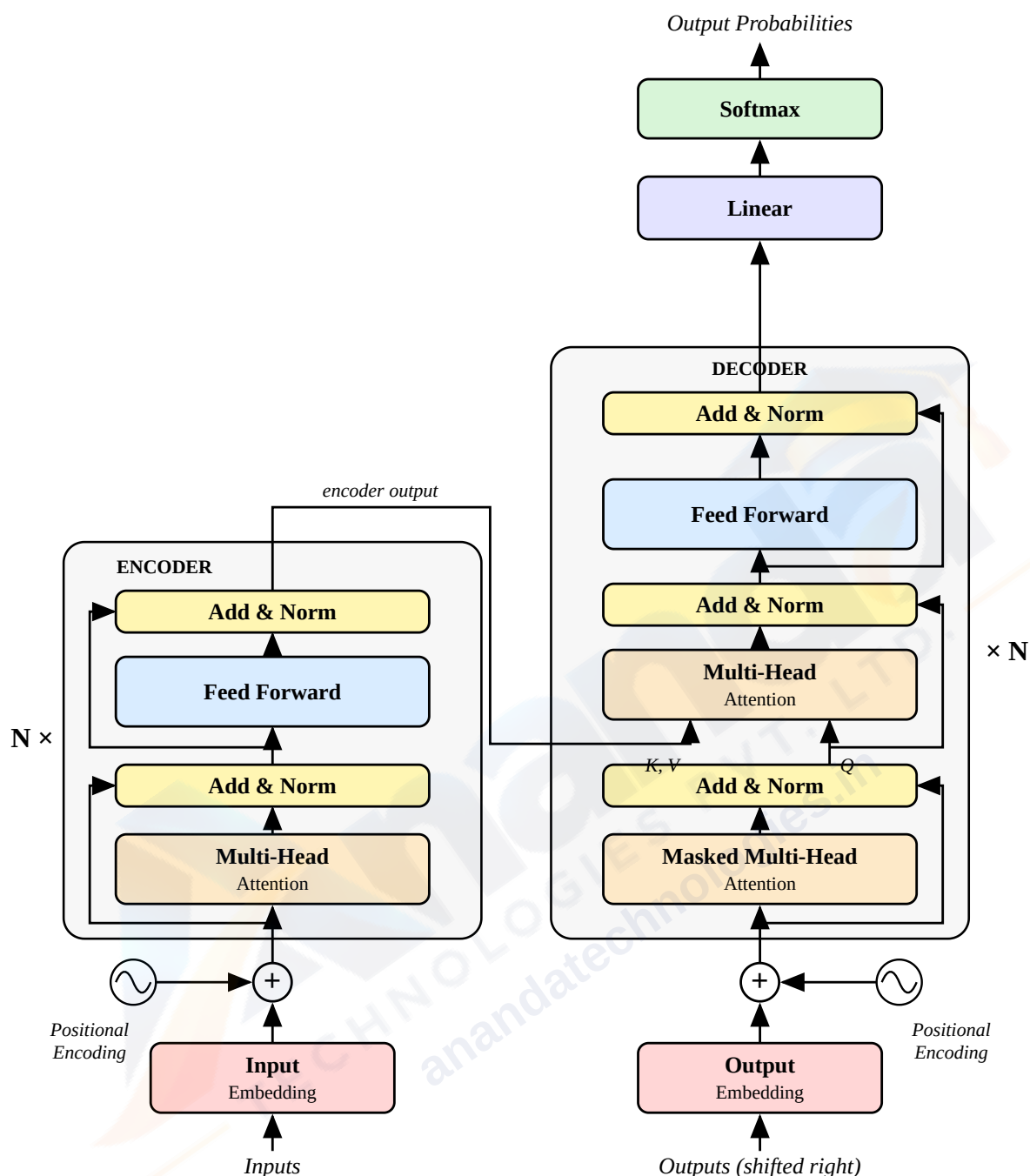


Fig.: Transformer architecture (encoder on the left, decoder on the right)

**Architecture:** encoder stack ( $N = 6$  layers of multi-head self-attention + feed-forward, each with Add & Norm) and decoder stack (masked self-attention, encoder–decoder attention, feed-forward), followed by Linear + Softmax. All tokens are processed in parallel.

**Multi-head attention:** instead of one attention function, the  $Q, K, V$  vectors are projected  $h$  times ( $h = 8$ ) into smaller sub-spaces; each head computes  $\text{softmax}(QK^T/\sqrt{d_k})V$  separately, the  $h$  outputs are concatenated and multiplied by a weight matrix  $W^O$ . Different heads learn different relations (subject–verb, pronoun–noun, nearby words).

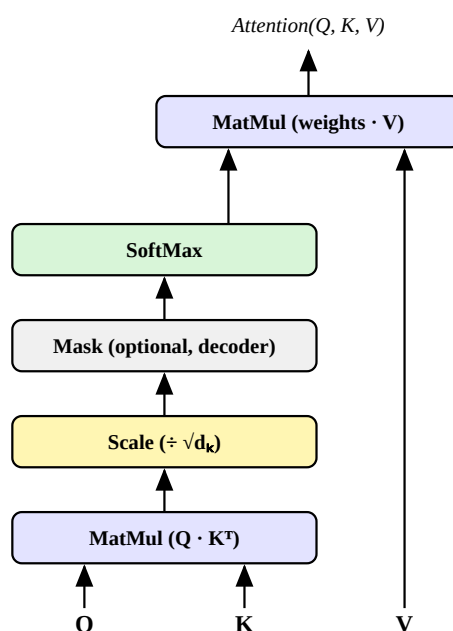


Fig.: Scaled dot-product attention used inside each head

**Positional encoding:** attention has no idea of word order, so a position vector is added to each word embedding:  $PE(pos, 2i) = \sin(pos / 10000^{2i/d})$ ,  $PE(pos, 2i+1) = \cos(pos / 10000^{2i/d})$ . Each position gets a unique pattern and relative positions can be learnt.

(Diagram 2 M, architecture 2 M, multi-head attention 1 M, positional encoding 1 M)

## b) HMM POS tagging — Viterbi.

6 M

In **stochastic (HMM) POS tagging**, tags are hidden states and words are observations. The tagger chooses the tag sequence  $t_1 \dots t_n$  that maximizes  $\prod P(w_i | t_i) \times P(t_i | t_{i-1})$  — emission probability  $\times$  transition probability, both estimated from a tagged corpus by counting. The **Viterbi algorithm** (dynamic programming) finds this best sequence efficiently:  $v_i(t) = \max$  over previous tags  $p$  of  $[v_{i-1}(p) \times P(t | p)] \times P(w_i | t)$ , with a back-pointer to the best  $p$ .

| Word | Tag | Viterbi calculation                                                                                    | v      |
|------|-----|--------------------------------------------------------------------------------------------------------|--------|
| fish | N   | $P(\text{start} \rightarrow N) \times P(\text{fish} N) = 0.7 \times 0.6$                               | 0.4200 |
| fish | V   | $P(\text{start} \rightarrow V) \times P(\text{fish} V) = 0.3 \times 0.1$                               | 0.0300 |
| can  | N   | $\max(0.4200 \times 0.3, 0.0300 \times 0.6) \times P(\text{can} N) = \max(0.1260, 0.0180) \times 0.2$  | 0.0252 |
| can  | V   | $\max(0.4200 \times 0.7, 0.0300 \times 0.4) \times P(\text{can} V) = \max(0.2940, 0.0120) \times 0.5$  | 0.1470 |
| swim | N   | $\max(0.0252 \times 0.3, 0.1470 \times 0.6) \times P(\text{swim} N) = \max(0.0076, 0.0882) \times 0.2$ | 0.0176 |
| swim | V   | $\max(0.0252 \times 0.7, 0.1470 \times 0.4) \times P(\text{swim} V) = \max(0.0176, 0.0588) \times 0.4$ | 0.0235 |

**Backtrace:** the largest value in the last column is  $v(\text{swim}, V) = 0.0235$ ; following the back-pointers gives the best sequence **fish/N** → **can/V** → **swim/V** (checked against all 8 possible sequences by brute force).

So ‘fish’ is a noun, ‘can’ and ‘swim’ are verbs — the context (transition probabilities) decides the tag even though ‘fish’ and ‘swim’ can be both nouns and verbs.

(HMM explanation 2 M, Viterbi table 3 M, answer 1 M (computed with Python))

### c) NLTK program — NP, VP and ADJP chunks.

Chunking uses a **regular-expression grammar over POS tags**. Rules are applied in order, so ADJP chunks are built first and then used inside NP chunks.

```
import nltk

sentence = "The very tall boy quickly climbed the old mango tree"
tagged = nltk.pos_tag(nltk.word_tokenize(sentence))
print(tagged)

grammar = r"""
ADJP: {<RB.*>?<JJ.*>}          # adjective phrase: optional adverb + adjective
NP:   {<DT>?<ADJP>*<NN.*>+}      # noun phrase
VP:   {<RB.*>?<VB.*>}          # verb phrase: optional adverb + verb
"""

tree = nltk.RegexpParser(grammar).parse(tagged)
print(tree)
for label in ("NP", "VP", "ADJP"):
    found = tree.subtrees(lambda s: s.label() == label)
    print(label, ":", [" ".join(w for w, t in st.leaves()) for st in found])
```

### Output:

```
[('The', 'DT'), ('very', 'RB'), ('tall', 'JJ'), ('boy', 'NN'), ('quickly', 'RB'),
('climbed', 'VBD'), ('the', 'DT'), ('old', 'JJ'), ('mango', 'NN'), ('tree', 'NN')]
(S
  (NP The/DT (ADJP very/RB tall/JJ) boy/NN)
  (VP quickly/RB climbed/VBD)
  (NP the/DT (ADJP old/JJ) mango/NN tree/NN))
NP : ['The very tall boy', 'the old mango tree']
VP : ['quickly climbed']
ADJP : ['very tall', 'old']
```

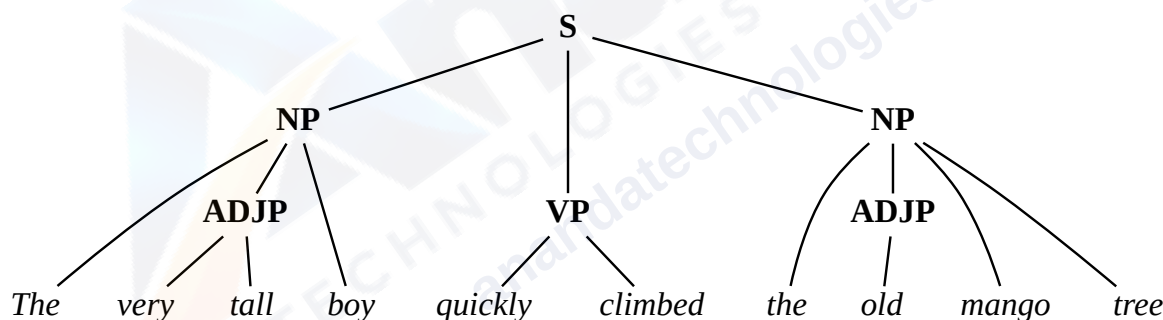


Fig.: Chunk tree

(Grammar 2 M, program 2 M, output/tree 2 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](http://anandatechnologies.in)