Assignment 1 — Solutions

NLP Basics & Text Preprocessing

Part A — Foundations · 25 questions with answers

Q1. What is NLP? Give 3 real-world applications you have seen in production. ★

Answer

Natural Language Processing (NLP) is the branch of AI that enables computers to understand, interpret and generate human language (text and speech). It combines linguistics, machine learning and deep learning.

Real-world applications:

- Chatbots and virtual assistants (customer support bots, ChatGPT-style assistants)
- Sentiment analysis of product reviews / social media
- Machine translation (Google Translate)
- Spam and fraud-message detection, email classification
- Document search, question answering and summarization (RAG systems)

Q2. What are the stages in the lifecycle of an NLP project (data → deployment → monitoring)?

Answer

1. **Problem definition** – business goal, success metric (accuracy, F1, latency).
2. **Data collection** – logs, documents, APIs, web scraping, labelled datasets.
3. **Data cleaning & preprocessing** – remove noise, normalize, tokenize.
4. **EDA** – word frequencies, class balance, text lengths.
5. **Feature engineering / representation** – BoW, TF-IDF, embeddings.
6. **Modelling** – classical ML (Naive Bayes, LR) or deep learning / Transformers / LLMs.
7. **Evaluation** – precision, recall, F1, BLEU/ROUGE, human evaluation.
8. **Deployment** – API (FastAPI), batch jobs, containers.
9. **Monitoring & maintenance** – data drift, model drift, feedback loop, retraining.

Q3. What are the main challenges in NLP? ★

Answer

- **Ambiguity** – the same word/sentence can have many meanings.
- **Context and long-range dependencies** – meaning depends on earlier text.
- **Sarcasm, irony and idioms** – "Great, another Monday!" is negative.
- **Spelling mistakes, slang, emojis, code-mixing** (Hinglish).
- **Out-of-vocabulary words** and new terminology.
- **Low-resource languages** – little labelled data.
- **Domain shift** – a model trained on news fails on medical text.
- **Bias** in training data, and **hallucination** in generative models.

Q4. What is ambiguity in NLP? Explain lexical, syntactic, semantic and pragmatic ambiguity with examples.

Answer

Ambiguity means a word, phrase or sentence can be interpreted in more than one way.

- **Lexical ambiguity** – a single word has several meanings: "bank" (river bank / money bank).
- **Syntactic (structural) ambiguity** – the sentence can be parsed in different ways: "I saw the man with the telescope" (who has the telescope?).
- **Semantic ambiguity** – the meaning of the full sentence is unclear even after parsing: "The chicken is ready to eat" (chicken will eat, or chicken will be eaten).
- **Pragmatic ambiguity** – meaning depends on context/intent: "Can you pass the salt?" is a request, not a question about ability.
- (Also **anaphoric/referential** ambiguity: "John told Mark he won" – who is "he"?)

Q5. Define corpus, document, vocabulary and token.

Answer

- **Corpus** – a large collection of text documents used for analysis or training (e.g. all Wikipedia articles).
- **Document** – a single unit of text in the corpus (one review, one email, one article).
- **Vocabulary** – the set of unique tokens/words in the corpus.
- **Token** – the smallest unit after tokenization (a word, subword or character).

Example: corpus = 1,000 reviews; each review = document; "good", "bad"... = vocabulary; "I", "love", "it" = tokens.

Q6. What are the common preprocessing steps in NLP? Which of them should you SKIP when using a Transformer/LLM, and why?

Answer

Common steps: lowercasing, removing punctuation/special characters, removing HTML/URLs, tokenization, stopword removal, stemming/lemmatization, handling contractions, spelling correction, removing numbers, handling emojis.

Skip for Transformers/LLMs: stopword removal, stemming/lemmatization, heavy punctuation removal and (often) lowercasing.

Why: Transformers learn from the full context. Words like "not", "no" are stopwords but change meaning ("not good"). Pre-trained models expect natural text with their own tokenizer, so heavy cleaning makes the input look different from the pre-training data and usually reduces accuracy. Only do light cleaning (remove HTML, extra whitespace, PII).

Q7. What is text normalization? List common normalization methods.

Answer

Text normalization converts text into a standard, consistent form so that different variants of the same word are treated alike.

Methods:

- Lowercasing ("Apple" → "apple")
- Removing punctuation, special characters, extra whitespace
- Expanding contractions ("don't" → "do not")
- Stemming and lemmatization
- Spelling correction

- Unicode normalization (removing accents: "café" → "cafe")
- Converting numbers/dates to a standard format, or replacing them with tokens like <NUM>
- Handling slang/abbreviations ("u" → "you")

Q8. What is tokenization? Compare word-level, character-level and subword tokenization. ★

Answer

Tokenization splits text into smaller units called tokens, which are then mapped to IDs for the model.

- **Word-level:** "I love NLP" → ["I", "love", "NLP"]. Simple, but huge vocabulary and OOV problem.
- **Character-level:** ["I", " ", "l", "o", ...]. Tiny vocabulary, no OOV, but very long sequences and weak meaning per token.
- **Subword-level (BPE, WordPiece, SentencePiece):** "unhappiness" → ["un", "happi", "ness"]. Balanced vocabulary size (30k–200k), rare words are split into known pieces, so no OOV. **All modern Transformers/LLMs use subword tokenization.**

Q9. What is stemming and how is it different from lemmatization? When would you prefer each? ★ · Asked: NLP interview rounds

Answer

Both reduce a word to a base form.

- **Stemming** chops suffixes with rules (Porter, Snowball). Fast but crude; output may not be a real word: "studies" → "studi", "running" → "run".
- **Lemmatization** uses vocabulary + grammar (POS) to return the dictionary form (lemma): "studies" → "study", "better" → "good" (adjective), "was" → "be".

Prefer stemming for speed on large data (search indexing, simple classifiers). **Prefer lemmatization** when meaning/readability matters (chatbots, topic modelling, information extraction).

Q10. Explain over-stemming vs under-stemming with examples.

Answer

- **Over-stemming:** two words with different meanings are reduced to the same stem (false positive). Example: "university" and "universe" → "univers"; "news" → "new".
- **Under-stemming:** words with the same meaning are reduced to different stems (false negative). Example: "data" → "dat" but "datum" → "datu"; "alumnus" / "alumni" give different stems.

Lemmatization reduces both problems because it uses a dictionary.

Q11. Why does WordNetLemmatizer need POS tags? What happens if you don't pass them?

Answer

WordNetLemmatizer **assumes every word is a noun by default**. The lemma depends on the part of speech, so without a POS tag verbs and adjectives are not reduced correctly.

```
from nltk.stem import WordNetLemmatizer
lem = WordNetLemmatizer()
lem.lemmatize("running")          # 'running'  (treated as noun)
lem.lemmatize("running", pos="v") # 'run'
lem.lemmatize("better", pos="a")  # 'good'
```

In practice, get POS tags using nltk.pos_tag() and map them to WordNet tags (n, v, a, r).

Q12. What are contractions (e.g. "don't", "I'm") and how do you handle them?

Answer

Contractions are shortened forms of words: "don't" = "do not", "I'm" = "I am", "won't" = "will not".

Why handle them: tokenizers may split them oddly ("don", "'", "t"), and the negation "not" can be lost, which hurts sentiment analysis.

How: expand them using a dictionary or the `contractions` library:

```
import contractions
contractions.fix("I don't think we'll make it")
# 'I do not think we will make it'
```

For Transformers, this is usually not required because their tokenizers already handle contractions.

Q13. How do you handle emojis, URLs, hashtags and HTML tags in text?

Answer

- **HTML tags:** remove with BeautifulSoup `get_text()` or regex `<.*?>`.
- **URLs:** remove with regex `https?://\S+` or replace with a token `<URL>`.
- **Hashtags / mentions:** remove '#' but keep the word ("#MachineLearning" → "Machine Learning"); replace @user with `<USER>`.
- **Emojis:** depends on task – for sentiment, **convert to text** with `emoji.demojize("😀")` → ":grinning_face:" because they carry emotion; otherwise remove them. Modern LLM tokenizers can process emojis directly.

```
import re, emoji
text = re.sub(r"<.*?>", "", text)
text = re.sub(r"https?://\S+", "<URL>", text)
text = emoji.demojize(text)
```

Q14. How do you handle spelling mistakes in user input?

Answer

- Spell-check libraries: **TextBlob** (`TextBlob(t).correct()`), **pyspellchecker**, **SymSpell** (very fast, edit-distance based).
- Edit distance (Levenshtein) to find the closest dictionary word.
- Context-aware correction using language models (e.g. a T5/BERT-based corrector, or ask an LLM to fix spelling).
- Use **subword tokenization or character-level models** (FastText) so misspelled words still get meaningful vectors.
- For search, use **fuzzy matching**.

Be careful: do not "correct" domain terms, names or product codes – keep a custom whitelist.

Q15. What are subword tokenization methods? Explain BPE, WordPiece and SentencePiece briefly. ★

Answer

Subword tokenization splits rare words into frequent smaller units, so the vocabulary stays small and there are no OOV words.

- **BPE (Byte Pair Encoding)** – starts with characters and repeatedly **merges the most frequent pair** of symbols until the vocabulary size is reached. Used by GPT-2/3/4, Llama 3, RoBERTa (byte-level BPE).
- **WordPiece** – similar to BPE but merges the pair that **maximizes the likelihood** of the training data (score = freq(ab) / (freq(a)·freq(b))). Continuation pieces are prefixed with "##". Used by BERT, DistilBERT.
- **SentencePiece** – a library that treats text as a raw stream (spaces become "▁"), so it is **language-independent** and needs no pre-tokenization. Implements BPE or **Unigram LM** (starts with a big vocabulary and prunes tokens that least reduce likelihood). Used by T5, ALBERT, XLNet, Llama 1/2, Mistral.

Q16. Explain step by step how the WordPiece tokenizer builds its vocabulary and tokenizes a new word.

Answer

Training (building vocabulary):

1. Pre-tokenize text into words (split on whitespace/punctuation).
2. Start with a vocabulary of all single characters; non-initial characters get "##" (e.g. "h", "##u", "##g").
3. For every adjacent pair, compute score = freq(pair) / (freq(first) × freq(second)).
4. Merge the pair with the highest score and add it to the vocabulary.
5. Repeat until the target vocabulary size (e.g. 30,522 for BERT) is reached.

Tokenizing a new word – greedy longest-match-first:

1. Find the longest prefix of the word that is in the vocabulary.
2. Take it as a token, then repeat on the remaining part, looking for "##" pieces.
3. If no piece matches, the whole word becomes [UNK].

Example (bert-base-uncased): "tokenization" → ["token", "##ization"]; "unaffable" → ["una", "##ffa", "##ble"]. The exact split depends on the learned vocabulary.

Q17. Which tokenizers are used by GPT-family models and by BERT? ★

Answer

- **GPT-2 / GPT-3:** byte-level **BPE** (~50k tokens).
- **GPT-3.5 / GPT-4 / GPT-4o:** byte-level BPE via OpenAI's **tiktoken** library (cl100k_base ~100k tokens, o200k_base ~200k tokens).
- **BERT: WordPiece** (vocabulary 30,522 for bert-base-uncased).

(For comparison: T5 uses SentencePiece-Unigram; Llama 2 uses SentencePiece-BPE; Llama 3 uses tiktoken-style BPE.)

Q18. Why does an LLM sometimes struggle to count letters in a word or do arithmetic? (Hint: tokenization.)

Answer

LLMs do not see letters or digits – they see **tokens**. "strawberry" may be one or two tokens, so the model never directly sees the individual letters "r", "r", "r" and must have memorized spelling. Similarly, numbers are split inconsistently ("12345" might be "123" + "45"), so digit-by-digit arithmetic is hard.

Fixes: ask the model to spell the word out first (chain-of-thought), use tools (a Python/calculator tool) for counting and maths, or use models with digit-level tokenization.

Q19. Write code for word and sentence tokenization (NLTK / spaCy). [CODE]

Answer

```
import nltk
nltk.download("punkt"); nltk.download("punkt_tab")
from nltk.tokenize import word_tokenize, sent_tokenize

text = "NLP is fun. Transformers changed everything! Don't you agree?"
print(sent_tokenize(text))
# ['NLP is fun.', 'Transformers changed everything!', "Don't you agree?"]
print(word_tokenize(text))
# ['NLP', 'is', 'fun', '.', 'Transformers', ..., 'Do', "n't", 'you', 'agree', '?']

# spaCy
import spacy
nlp = spacy.load("en_core_web_sm")      # python -m spacy download en_core_web_sm
doc = nlp(text)
print([t.text for t in doc])
print([s.text for s in doc.sents])

# Subword tokenization (Hugging Face)
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("bert-base-uncased")
print(tok.tokenize("Tokenization is unbelievable"))
# ['token', '##ization', 'is', 'unbelievable']
```

Q20. Write code for stemming and lemmatization and compare the outputs on the same sentence. [CODE]

Answer

```
import nltk
nltk.download("wordnet"); nltk.download("averaged_perceptron_tagger_eng")
from nltk.stem import PorterStemmer, SnowballStemmer, WordNetLemmatizer
from nltk.corpus import wordnet
from nltk import pos_tag, word_tokenize

text = "The children were running and studies showed better results"
words = word_tokenize(text)

porter = PorterStemmer()
lem = WordNetLemmatizer()

def wn_pos(tag):
    return {"J": wordnet.ADJ, "V": wordnet.VERB, "R": wordnet.ADV}.get(tag[0], wordnet.NOUN)

for w, t in pos_tag(words):
    print(f"{w:10} stem={porter.stem(w):10} lemma={lem.lemmatize(w, wn_pos(t))}")
# children   stem=children   lemma=child
# were       stem=were       lemma=be
# running    stem=run        lemma=run
# studies    stem=studi      lemma=study
# better     stem=better     lemma=good
```

Observation: the stemmer produces non-words ("studi") and misses irregular forms ("were"), while the lemmatizer returns real dictionary words.

Q21. Write code for stopword removal. When can removing stopwords hurt a model? [CODE]

Answer

```
import nltk
nltk.download("stopwords")
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

stop = set(stopwords.words("english"))
text = "This movie is not good and I would not watch it again"
tokens = [w for w in word_tokenize(text.lower()) if w.isalpha() and w not in stop]
print(tokens)   # ['movie', 'good', 'would', 'watch']   <- 'not' was removed!

# Keep negations
stop_custom = stop - {"not", "no", "nor", "never"}
print([w for w in word_tokenize(text.lower()) if w not in stop_custom])
# ['movie', 'not', 'good', 'would', 'not', 'watch']
```

When removing stopwords hurts: sentiment analysis (negations), question answering ("who", "what"), machine translation, text generation, and **any Transformer/LLM**, because they rely on full grammatical context.

Q22. In real-time data users type "Lunch", "lunch" and "LUNCH". Why do we lowercase text, and when should you NOT lowercase? ★ · Asked: NLP interview rounds

Answer

Why lowercase: so that different surface forms of the same word map to **one token/feature**. Without it, "Lunch", "lunch" and "LUNCH" become three different vocabulary entries → bigger, sparser vocabulary, fewer examples per word, worse generalization for BoW/TF-IDF/Word2Vec models. Users at inference time type in any case, so training and inference must be normalized the **same way**.

Do NOT lowercase when case carries meaning:

- **NER:** "Apple" (company) vs "apple" (fruit); "US" vs "us".
- Acronyms, product codes, tickers, IDs ("SBI", "PAN", "IT" department).
- Sentiment/intent where shouting matters ("I am ANGRY").
- **Transformer/LLM pipelines with cased models** – the tokenizer and model were trained on cased text; just pass raw text.

Rule: whatever you do in training, do exactly the same in the inference pipeline.

Q23. You removed all numeric values from the text. Could numbers contain useful information? How would you handle numbers and punctuation instead? · Asked: NLP interview rounds

Answer

Yes – numbers often carry the key signal:

- Prices and amounts ("₹1,999", "20% off"), dates, quantities ("3 days late"), ratings ("5/5"), versions ("iPhone 15"), error codes ("E-4021"), account/transaction amounts in fraud text.

Better approaches than deleting them:

- **Replace with placeholder tokens** that keep the type: `<NUM>`, `<PRICE>`, `<DATE>`, `<PERCENT>` (regex or NER).

- **Bucket** values ("delay_1_3_days", "amount_high").
- **Extract as structured features** (amount, date) for the model alongside the text.
- Keep them as-is for Transformers/LLMs, which handle numbers in context.

Punctuation: don't blindly drop it – "!!!" signals emotion, "?" signals a question, "not good." vs "not, good" can matter; ":)" is sentiment. Remove only noise for classical models; keep it for Transformers. Decide based on the task and validate with metrics.

Q24. Users will type stopwords at inference time. If you remove them during training, what happens? When should you remove stopwords and when not? Give practical applications. ★ · Asked: NLP interview rounds

Answer

Preprocessing must be **identical at training and inference** – if training removed stopwords, the inference pipeline must remove them too, otherwise the input distribution differs and accuracy drops. The bigger risk is **losing meaning**: "not", "no", "never", "without" are stopwords in NLTK.

Remove stopwords when: using classical BoW/TF-IDF models for **topic-level tasks** – document classification by topic, keyword extraction, topic modelling (LDA), search indexing/keyword search – where they add noise and dimensionality.

Keep stopwords when:

- **Sentiment analysis** ("not good" ≠ "good").
- **Question answering, chatbots, intent detection** ("where", "how", "can I").
- Machine translation, summarization, text generation, NER.
- **Any Transformer/LLM/RAG pipeline** (they need full grammar and context).

Use a **custom stopword list** that keeps negations if you must remove them.

Q25. Live-coding: clean this messy real-world review end-to-end and explain why the ORDER of steps matters: [CODE] · Asked: NLP interview rounds

```
raw = "Heyyyy!!! I justtt bought this from @Amazon □. TBH it's sooo goodddd!!! Price was Rs.1,999 &
    I got 20% OFF. Plz check ur order b4 buying #WorthIt <br> https://amzn.in/x"
```

Answer

Order matters: remove HTML/URLs/mentions/hashtags **before** removing punctuation (otherwise "https://..." leaves junk words); expand contractions and slang **before** removing punctuation (the apostrophe in "it's" is needed); lowercase **before** dictionary lookups; normalize spaces **last**.

```
import re, emoji

SLANG = {"tbh": "to be honest", "plz": "please", "ur": "your", "b4": "before",
         "idk": "i do not know", "btw": "by the way", "gr8": "great", "asap": "as soon as possible"}
CONTRACTIONS = {"it's": "it is", "can't": "cannot", "i'd": "i would", "don't": "do not",
                "i'm": "i am", "won't": "will not"}

def clean(text):
    text = re.sub(r"<.*?>", " ", text)                # 1. HTML tags
    text = re.sub(r"https?://\S+|www\.\S+", " ", text) # 2. URLs
    text = re.sub(r"[@#]\w+", " ", text)              # 3. mentions + hashtags
    text = emoji.replace_emoji(text, replace=" ")     # 4. emojis
    text = text.lower()                               # 5. lowercase
    for k, v in CONTRACTIONS.items():                 # 6. contractions (needs apostrophe)
        text = text.replace(k, v)
    text = re.sub(r"(\w)\1{2,}", r"\1", text)         # 7. "sooo"->"so", "goodddd"->"good"
    words = [SLANG.get(w, w) for w in text.split()]   # 8. chat slang (before punctuation removal)
    text = " ".join(words)
    text = re.sub(r"[^a-z\s]", " ", text)             # 9. punctuation, numbers, symbols
```

```
    return re.sub(r"\s+", " ", text).strip()           # 10. extra spaces

print(clean(raw))
# hey i just bought this from to be honest it is so good price was rs i got off
# please check your order before buying
print(clean(raw).split())   # tokens
```

Follow-ups to expect: "goodddd" → "god" if you collapse to one letter blindly (that's why the rule keeps one letter only for 3+ repeats, and some pipelines use a spell-checker afterwards); numbers like "20%" and the price were dropped – mention you would keep them as <PERCENT>/<PRICE> if they matter; for a Transformer you would do only steps 1–4.