[illegible]

TEXT ACQUISITION & INITIAL PROCESSING

[illegible]

1️⃣ TEXT EXTRACTION

[illegible]

Problem

[illegible]

- [illegible]
- [illegible]
- [illegible]
- [illegible]

METHODS TO HANDLE TEXT EXTRACTION

Method 1: Raw Text Extraction (Naive)

[illegible]

Method 2: Parser-Based Extraction (STRUCTURE AWARE)

BEST & MOST USED (Google-level)

[illegible]

Method 3: Semantic Extraction (ADVANCED)

[illegible]

BEST for large-scale systems

CODE: TEXT EXTRACTION

HTML

[illegible]

PDF

[illegible]

BEST PRACTICE SUMMARY

[illegible]

2️⃣ ENCODING NORMALIZATION (UTF-8)

Problem

[illegible]

METHODS TO HANDLE ENCODING

Method 1: Ignore Encoding Errors

[illegible]

BAD PRACTICE

Method 2: Replace Invalid Characters

[illegible]

Method 3: Detect → Normalize Encoding

BEST & PRODUCTION STANDARD

[illegible]

CODE: ENCODING NORMALIZATION

[illegible]

BEST PRACTICE SUMMARY

[illegible]

3️⃣ LINE BREAK & WHITESPACE NORMALIZATION

Problem

[illegible]

METHODS TO HANDLE WHITESPACE

Method 1: Simple Strip

[illegible]

Method 2: Regex Normalization

[illegible]

Method 3: Sentence-Aware Normalization

[illegible]

CODE: WHITESPACE NORMALIZATION

[illegible]

BEST PRACTICE SUMMARY

[illegible]

4️⃣ DUPLICATE TEXT REMOVAL

Problem

[illegible]

METHODS TO HANDLE DUPLICATES

Method 1: Exact Match Removal

[illegible]

Method 2: Hash-Based Deduplication

BEST & FAST

[illegible]

Method 3: Semantic Deduplication (ADVANCED)

[illegible]

CODE: DUPLICATE REMOVAL

[illegible]

BEST PRACTICE SUMMARY

[illegible]

GOOGLE ENGINEER FINAL DECISION TABLE

[illegible]

ONE-LINE LIFETIME SUMMARY

[illegible]

Ready for Topic 2

[illegible]

[illegible]

[illegible]

Next → Topic 2: Noise Removal Methods

NOISE REMOVAL METHODS

> Noise = anything in the text that does not help the model learn meaning or may confuse it. [illegible]

1️⃣ HTML / XML / JSON TAG REMOVAL

Problem

- Text scraped from web contains HTML, XML, or JSON wrappers
- Example:

```
[illegible]
```

```
[illegible]
```

Methods

Regex based removal

```
[illegible]
```

- Simple, fast
- Can break nested tags

Parser-based removal

```
[illegible]
```

- Handles nested tags, safe
- **Best practice for web scraping**

JSON extraction

```
[illegible]
```

- Works for API responses
- Only structured JSON

Before → After Example

```
[illegible]
```

2️⃣ URL REMOVAL

Problem

- URLs don't add meaning for sentiment/topic analysis
- Example:

```
[illegible]
```

Methods

Regex removal

```
[illegible]
```

Masking URLs

```
[illegible]
```

- Useful in models where URL presence matters (spam detection)
- **Masking preferred in production**

Keep as-is

- Only for **security / phishing detection**

Before → After Example

```
[illegible]
```

3️⃣ EMAIL ADDRESS REMOVAL

Methods

Regex removal

```
[illegible]
```

Masking

```
[illegible]
```

- Preserves placeholder info
- **Masking preferred in privacy-sensitive pipelines**

Before → After Example

```
[illegible]
```

4️⃣ PHONE NUMBER REMOVAL

Methods

Regex

```
[illegible]
```

Masking

```
[illegible]
```

Keep for Financial / Telecom NLP

Before → After Example

```
[illegible]
```

5️⃣ SOCIAL MEDIA HANDLES (`@user`) & HASHTAGS

Methods

Remove completely

```
[illegible]
```

Masking

```
[illegible]
```

Hashtag processing

- Keep keyword only

```
[illegible]
```

- [illegible]

Before → After Example

```
[illegible]
```

6️⃣ SPECIAL CHARACTER & PUNCTUATION REMOVAL

Problem

- Symbols like [illegible] may be noise
- But some punctuation is meaningful [illegible] for sentiment

Methods

Remove all symbols

```
[illegible]
```

Remove selectively (keep [illegible])

```
[illegible]
```

- Best for sentiment analysis
- **Selective removal preferred in production**

Before → After Example

```
[illegible]
```

7️⃣ EXTRA SPACE REMOVAL

Methods

Simple strip

```
[illegible]
```

Regex normalization

```
[illegible]
```

- Ensures consistent tokenization
- **Regex normalization is standard**

Before → After Example

```
[illegible]
```

8️⃣ BOILERPLATE TEXT REMOVAL (HEADERS, FOOTERS)

Methods

Manual rules

- Remove known headers/footers
- Simple, works for small datasets

Boilerplate removal libraries

- e.g. [illegible]

```
[illegible]
```

- **Best for large-scale web scraping**

Content extraction via ML

- Classifier separates main content vs noise
- Used by Google News, search engines

Before → After Example

```
[illegible]
```

COMPLETE PYTHON CODE EXAMPLE (PRODUCTION PIPELINE)

```
[illegible]
```

Output

```
[illegible]
```

BEST PRACTICES SUMMARY

Noise Type	Best Handling (Google-style)
HTML/XML	[illegible]
JSON	[illegible]
URL	[illegible]
Email	[illegible]
Phone	[illegible]
Social handle	[illegible]
Hashtag	[illegible]
Special chars	Remove selectively
Extra spaces	Regex normalization
Boilerplate	[illegible]

ONE-LINE LIFETIME SUMMARY

> **Noise removal = preserve meaning while removing everything that confuses the model, using parsers, masking, and selective cleaning.**

If you want, **next we can move to Topic 3: Text Normalization & Lowercasing**, which also involves **contractions, spelling, accents, repeated characters** — [illegible]

Do you want to proceed with that?

[illegible]

[illegible]

TEXT NORMALIZATION METHODS

> [illegible]

1️⃣ LOWERCASING & CASE FOLDING

Problem

- [illegible]
- [illegible]

Methods

[illegible]

```
[illegible]
```

[illegible]

Python Example

```python
[illegible]
```

Output

```
[illegible]
```

2️⃣ UNICODE & ACCENT NORMALIZATION

Problem

- [illegible]
- [illegible]

Methods

[illegible]

```python
[illegible]
```

[illegible]

3️⃣ CONTRACTION EXPANSION

Problem

- [illegible]
- [illegible]

Methods

[illegible]

```python
[illegible]
```

[illegible]

4️⃣ SPELLING CORRECTION

Problem

- [illegible]

Methods

[illegible]

```python
[illegible]
```

[illegible]

5️⃣ BRITISH → AMERICAN SPELLING

Problem

- [illegible]

Methods

[illegible]

```python
[illegible]
```

[illegible]

6️⃣ REPEATED CHARACTER NORMALIZATION

Problem

- [illegible]

Methods

[illegible]

```python
[illegible]
```

[illegible]

7️⃣ SLANG / INFORMAL LANGUAGE STANDARDIZATION

Problem

- [illegible]

Methods

[illegible]

```python
[illegible]
```

[illegible]

8️⃣ TEXT CANONICALIZATION

Problem

- [illegible]

Methods

[illegible]

```
[illegible]
```

[illegible]

COMPLETE PYTHON CODE EXAMPLE

```python
[illegible]
```

Output

```
[illegible]
```

BEST PRACTICES SUMMARY (Google-level)

[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

ONE-LINE LIFETIME SUMMARY

> [illegible]

[illegible]

[illegible]

Perfect! Let's dive **deep, Google-level, with methods, examples, and code for Topic 4: Tokenization Methods** 🎀

4️⃣ TOKENIZATION METHODS 🎀

> Tokenization = **breaking text into smaller pieces (tokens)** so a machine can process it.
> This step is **critical**: bad tokenization → bad embeddings → bad model.

1️⃣SENTENCE TOKENIZATION

📍 **Problem**

- For tasks like summarization, QA, or translation, models need **sentences, not entire paragraphs.**
- Raw text often contains multiple sentences without clear breaks.

Example

```
"I love NLP. It's amazing. Let's learn."
```

🔧 **Methods**

1️⃣ **Rule-based (Punctuation)**

```python
import re
sentences = re.split(r'[illegible]', text)
```

- ✅ Simple, fast
- ❌ Fails with abbreviations (`Dr.`, `Ph.D.`)

2️⃣ **NLTK / SpaCy**

```python
from nltk.tokenize import sent_tokenize
sentences = sent_tokenize(text)
```

- ✅ Handles abbreviations, decimals
- 🏆 Best for production classical NLP

3️⃣ **Transformer sentence splitting**

- Use models like `BERT` sentence splitter
- ✅ Handles complex structure & multi-language
- ⚠️ Slower

Before → After Example

```
"I love NLP. It's amazing. Let's learn."
→ ["I love NLP.", "It's amazing.", "Let's learn."]
```

2️⃣WORD TOKENIZATION

📍 **Problem**

- Models usually work on **words or subwords.**
- Raw text may have punctuation, emojis, or contractions.

Example

```
"I'm learning NLP!"
```

🔧 **Methods**

1️⃣ **Whitespace-based**

```python
text.split()
```

- ✅ Fast, simple
- ❌ Keeps punctuation attached (`NLP!`)

2️⃣ **Rule-based / Regex**

```python
import re
tokens = re.findall(r"[illegible]", text)
```

- ✅ Removes punctuation
- ⚠️ Misses special tokens ([illegible])

3️⃣ **NLTK / SpaCy**

```python
from nltk.tokenize import word_tokenize
tokens = word_tokenize(text)
```

- ✅ Handles punctuation, contractions
- 🏆 **Best for classical NLP pipelines**

4️⃣ **Transformers / Tokenizer**

- Subword-based tokenizers (WordPiece, BPE)
- ⚡ Handles OOV, rare words

Before → After Example

```
"I'm learning NLP!" → ["I", "'m", "learning", "NLP", "!"]
```

3️⃣CHARACTER TOKENIZATION

📍 **Problem**

- Sometimes you need **every character as a token** (OCR, text generation, spell correction)

🔧 **Methods**

1️⃣ **Simple list**

```python
chars = list(text)
```

- ✅ Fast, simple
- ❌ Long sequences

2️⃣ **Custom character set**

- Remove spaces/punctuation

3️⃣ **Embedding-based char tokenization**

- Used in deep learning (CNN on chars)
- 🏆 Best for noisy / social media text

Before → After Example

```
"NLP" → ["N","L","P"]
```

4️⃣SUBWORD TOKENIZATION (MOST IMPORTANT FOR DL)

📍 **Problem**

- Words like "unhappiness" may be OOV for embeddings
- Need subword splits to handle rare words

🔧 **Methods**

1️⃣ **BPE (Byte Pair Encoding)**

- Merge most frequent pairs iteratively
- Reduces vocab, handles OOV

Example

```
Word: "unhappiness"
Tokens: ["un", "happi", "ness"]
```

2️⃣ **WordPiece**

- Used in **BERT**
- Greedy longest match-first

Example

```
"playing" → ["play", "##ing"]
```

3️⃣ **SentencePiece**

- Language independent
- Uses **subword regularization**, no pre-tokenization needed
- ⚡ Google T5 / mBERT

Before → After Example

```
"unhappiness" → ["▁un", "happi", "ness"]
```

🏆 **Google-level best practice**

- Transformers → WordPiece / SentencePiece
- Classical ML → Word-level or regex-based

5️⃣WHITESPACE TOKENIZATION

- Splits text by spaces

```python
tokens = text.split()
```

- ✅ Fast, memory efficient
- ❌ Cannot handle punctuation, contractions
- ⚠️ Good only for large-scale bag-of-words

6️⃣RULE-BASED TOKENIZATION

- Custom rules for special tasks
- Examples:
  - Keep hashtags, @user mentions
  - Split on punctuation but keep emoticons
- ✅ Flexible, task-specific
- ⚠️ Hard to maintain at scale

7️⃣ PYTHON CODE EXAMPLES

```python
import re
from nltk.tokenize import word_tokenize, sent_tokenize
from transformers import BertTokenizer

text = "I'm learning NLP! It's amazing. [illegible]"

# Sentence tokenization
sentences = sent_tokenize(text)
print(sentences)

# Word tokenization
tokens_nltk = word_tokenize(text)
print(tokens_nltk)

# Character tokenization
chars = list(text)
print(chars)

# Subword tokenization (BERT WordPiece)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
tokens_subword = tokenizer.tokenize(text)
print(tokens_subword)

# Regex / rule-based tokenization
tokens_regex = re.findall(r"[illegible]", text)
print(tokens_regex)
```

Example Outputs

```
Sentences: ["I'm learning NLP!", "It's amazing.", "[illegible]"]
Word tokens: [illegible]
Character tokens: [illegible]
Subword tokens: [illegible]
Regex tokens: [illegible]
```

✅ BEST PRACTICES SUMMARY (Google-level)

Tokenization Type	Best Method
Sentence	NLTK / SpaCy
Word	NLTK / SpaCy (Classical ML), Subword (Transformers)
Character	Simple list or embeddings for noisy/rare text
Subword	WordPiece / SentencePiece (Transformers)
Whitespace	Only for bag-of-words or classical ML
Rule-based	Task-specific customization (hashtags, emoticons)

🔑 ONE-LINE LIFETIME SUMMARY

> Tokenization converts raw text into meaningful units — choose sentence, word, subword, or char level depending on the model and task. Subword tokenization is the industry standard for Transformer models.

If you want, **next we can do Topic 5: Stopwords Removal, Stemming & Lemmatization**,
which is **directly after tokenization and crucial for vocabulary reduction.**

Do you want to proceed?

Perfect Let's dive **deep, Google-level**, with **methods, examples, and code** for **Topic 5: Stopword Processing** 🔴

5️⃣ STOPWORD PROCESSING 🔴

> Stopwords = words that **carry little meaning by themselves** but appear frequently (`the`, `is`, `at`, `on`).
> Proper handling is **task-dependent**: removing all stopwords can sometimes **break meaning**.

1️⃣STOPWORD REMOVAL

❓ Problem

- Many common words do **not contribute meaning** to ML/NLP models.
- Example:

```arduino
"The cat is on the mat"
```

- Words like `the`, `is`, `on` may dilute features.

🔧 Methods

1️⃣ Default Stopword List

- Use NLTK, SpaCy, or gensim

```python
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
tokens = [w for w in text.split() if w.lower() not in stop_words]
```

- ✅ Fast, ready-to-use
- ❌ Not domain-specific

2️⃣ Custom Stopword List

- Extend or reduce based on domain

```python
custom_stop = stop_words.union(['rt','lol'])
```

- ✅ Google-level production: task-aware
- ⚠️ Requires analysis

3️⃣ Partial Stopword Removal

- Remove only high-frequency words (TF-IDF < threshold)
- ✅ Keeps some stopwords in meaningful contexts

⏳ Best Practice

> Use **default stopwords + task-specific customization**.
> Keep **negations** intact (`not`, `no`) if sentiment or polarity matters.

🔄 Before → After Example

Text:

```sql
"I do not like this movie at all"
```

- Remove default stopwords without negation preservation:

```arduino
"like movie"
```

- Negation preserved:

```arduino
"not like movie"
```

2️⃣DOMAIN-SPECIFIC STOPWORDS

❓ Problem

- Generic stopwords may not cover **industry-specific frequent words**
- Example in finance: `"stock"`, `"market"` appear very frequently → may need removal for topic modeling

---

🔧 Methods

1️⃣ Frequency-based identification

- Remove top X% frequent words in corpus

2️⃣ Domain expert creation

- Finance: `"stock","market","price"`
- Medical: `"patient","doctor","symptom"`

3️⃣ Dynamic stopwords

- Based on TF-IDF scores
- Words with low TF-IDF → stopwords

⏳ Best Practice

> Frequency + TF-IDF analysis + domain knowledge → Custom stopword list

---

🔄 Before → After Example (Finance text)

```arduino
"The stock market is volatile today"
```

- Generic removal:

```arduino
"stock market volatile today"
```

- Domain-specific removal:

```arduino
"volatile today"
```

3️⃣NEGATION PRESERVATION

❓ Problem

- Negation words like `not`, `no` are **semantically critical**
- Sentiment reversal happens if removed

🔧 Methods

1️⃣ Manual preservation

- Keep `not`, `no` in stopword removal
- ✅ Simple, effective

2️⃣ Negation tagging

```python
"I do not like it" → ["not_like"]
```

- ✅ Helps classical ML models understand polarity

3️⃣ Dependency parsing

- Advanced: detect negation scope in sentence
- ✅ Used in Google-level sentiment pipelines

🔄 Before → After Example

Text:

```arduino
"I do not enjoy this"
```

- Removing all stopwords:

```arduino
"enjoy"
```

- Negation preserved:

```arduino
"not enjoy"
```

🐍 PYTHON CODE EXAMPLE

```python
from nltk.corpus import stopwords

# Default stopwords
stop_words = set(stopwords.words('english'))

# Custom additions
custom_stop = stop_words.union(['rt','lol'])

# Example text
text = "I do not like this movie at all"

# Tokenize
tokens = text.split()

# Remove stopwords but preserve negation
negations = ['not','no']
tokens_clean = [w for w in tokens if w.lower() not in custom_stop or w.lower() in negations]

print(" ".join(tokens_clean))
```

Output

```arduino
"not like movie"
```

✅ BEST PRACTICES SUMMARY (Google-level)

Step	Best Method
Default stopword removal	NLTK / SpaCy list
Custom stopwords	Extend with domain-specific words
Negation	Preserve negations (`not`, `no`)
Partial removal	TF-IDF / frequency-based for topic modeling
Task-specific	Keep stopwords for sentiment analysis, NER, chatbots

---

🔑 ONE-LINE LIFETIME SUMMARY

> **Stopword removal is task-dependent: remove meaningless words while preserving polarity and domain-specific significance.**

Next, we can do **Topic 6: Stemming vs Lemmatization** ⏳,
which is **directly after stopwords** and crucial for vocabulary reduction and embeddings.

Do you want to proceed?

Perfect! Let's go **deep, example-rich, Google-level** for **Topic 6: Morphological Processing** ⏳

6️⃣ MORPHOLOGICAL PROCESSING ⏳

> Morphological processing = **reducing words to their base or root form** to reduce vocabulary size and help models generalize.
> Critical for **classical ML**, text search, and some embedding pipelines.

1️⃣STEMMING

❓ Problem

- Words like `"running"`, `"runs"`, `"ran"` → different tokens
- Stem to a **common root** reduces sparsity

Methods

1️⃣ **Porter Stemmer**

- Rule-based, simple

```python
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
stemmer.stem("running")  # → "run"
```

- ✅ Fast
- ❌ Sometimes over-aggressive: `"relational"` → `"relat"`

2️⃣ **Snowball Stemmer**

- Improved Porter, language-aware

```python
from nltk.stem import SnowballStemmer
stemmer = SnowballStemmer("english")
stemmer.stem("running")  # → "run"
```

- ✅ More accurate than Porter
- 🏆 **Best for classical ML pipelines**

3️⃣ **Lancaster Stemmer**

- Very aggressive, shorter roots
- ⚠️ Can be too destructive

Before → After Example

Word	Porter	Snowball
running	run	run
relational	relat	relation
studies	studi	studi

2️⃣LEMMATIZATION

❓ Problem

- Stemmer may produce **invalid words**
- Lemmatization → **dictionary-based**, POS-aware → real words

---

Methods

1️⃣ **WordNet Lemmatizer (NLTK)**

```python
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
lemmatizer.lemmatize("running", pos="v")  # → "run"
```

- ✅ Keeps valid words
- ✅ Handles POS
- 🏆 **Preferred for production classical NLP**

2️⃣ **SpaCy Lemmatizer**

```python
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("running runs ran")
[token.lemma_ for token in doc]
```

- ✅ Integrated with tokenization & POS
- ✅ Better for pipelines

3️⃣ **Transformer Embedding Approach**

- Let contextual embeddings **learn the root meaning**
- ⚠️ Stem/lemma unnecessary for modern LLMs

Before → After Example

Text: `"The cats are running fast"`

Word	Lemma
cats	cat
are	be
running	run
fast	fast

3️⃣ROOT WORD EXTRACTION & INFLECTION NORMALIZATION

❓ Problem

- Inflected words (`went`, `running`, `studies`) → need **base form**
- Reduces sparsity and helps embeddings/generalization

Methods

1️⃣ **Lemmatization** → Root extraction
2️⃣ **Stemming** → Quick reduction
3️⃣ **Morphological analyzers**

- E.g., `Pattern` library, spaCy, Stanza
- Useful in multilingual pipelines

Before → After Example

Word	Stem	Lemma
running	run	run
studies	studi	study
went	went	go

PYTHON CODE EXAMPLES

Porter & Snowball

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

words = ["running", "studies", "relational"]

ps = PorterStemmer()
ss = SnowballStemmer("english")

print([ps.stem(w) for w in words])
print([ss.stem(w) for w in words])
```

Lemmatization (WordNet + SpaCy)

```python
from nltk.stem import WordNetLemmatizer
import spacy

words = ["running", "studies", "went"]
lemmatizer = WordNetLemmatizer()
print([lemmatizer.lemmatize(w, pos="v") for w in words])

nlp = spacy.load("en_core_web_sm")
doc = nlp("running studies went")
print([token.lemma_ for token in doc])
```

✅ BEST PRACTICES (Google-level)

Step	Best Method
Classical ML (search, topic modeling)	Snowball Stemmer
Grammar / POS-aware tasks	WordNet Lemmatizer
Deep learning / Transformers	Skip stemming/lemmatization (contextual embeddings handle morphology)
Multilingual datasets	SpaCy / Stanza lemmatization
Inflection normalization	Lemmatization + POS tagging

🔑 ONE-LINE LIFETIME SUMMARY

> **Morphological processing reduces vocabulary sparsity by converting words to their base forms; choose stemmer for speed and lemmatizer for accuracy.**

If you want, **next we can do Topic 7: Emoji, Numbers, Slang & Special Tokens Handling**, which is **critical for social media / informal NLP pipelines**.

Do you want to proceed?

Perfect! Let's dive **deep, Google-level**, with **methods, examples, and code** for **Topic 7: Handling Special Text Elements** 🔍

7️⃣ HANDLING SPECIAL TEXT ELEMENTS 🔍

> Special text elements like **emojis, numbers, dates, and times** are very common in social media, chatbots, logs, and financial/medical data.
> Proper handling preserves meaning, ensures model robustness, and avoids noisy embeddings.

1️⃣EMOJIS & EMOTICONS 😊

❓ **Problem**

- Emojis carry **emotions**, sentiment, or context
- Example:

```
"I love this! 😍"
```

- Models ignoring them may **miss sentiment cues**

🛠 **Methods**

1️⃣ **Emoji Removal**

```python
import re
text = "I love this! 😍"
text = re.sub(r'[^\w\s,.!?]', '', text)
```

- ✅ Simple, reduces noise
- ❌ Loses emotion → bad for sentiment tasks

2️⃣ **Emoji → Text Conversion**

- Map emojis to words

```python
import emoji
emoji.demojize("I love this! 😍")
# → "I love this! :smiling_face_with_heart_eyes:"
```

- ✅ Preserves meaning for NLP models
- 🏆 **Best for sentiment analysis / chatbot pipelines**

3️⃣ **Emoji Sentiment Mapping**

- Map to polarity score: 😍 → +1, 😡 → -2
- ✅ Works for ML-based sentiment scoring
- ⚠️ Requires pre-defined emoji lexicon

🔄 **Before → After Example**

```
"I love this! 😍"
Emoji removal → "I love this!"
Emoji → Text → "I love this! :smiling_face_with_heart_eyes:"
Emoji sentiment → "I love this! 1"
```

2️⃣NUMBERS 🔢

❓ **Problem**

- Numbers may **inflate vocabulary** or cause irrelevant tokens
- Example: `"I scored 98 points"`

🛠 **Methods**

1️⃣ **Number Removal**

```python
text = re.sub(r'\d+', '', text)
# "I scored points"
```

- ✅ Simple, reduces noise
- ❌ Loses information → bad for finance, medicine

2️⃣ **Number Masking**

```python
text = re.sub(r'\d+', '<NUM>', text)
# "I scored <NUM> points"
```

- ✅ Keeps presence of numbers
- 🏆 **Best for general NLP tasks**

3️⃣ **Number Normalization**

- Convert `"1,000"` → `"1000"`
- `"5k"` → `"5000"`
- ✅ Standardizes representation for model learning

4️⃣ **Numeric Bucketing**

- Convert numbers into **ranges**:

```
0-10 → "low", 11-99 → "medium", 100+ → "high"
```

- ✅ Useful for classification tasks

🔄 **Before → After Example**

```
"I scored 98 points"
Removal → "I scored points"
Masking → "I scored <NUM> points"
Normalization → "I scored 98 points"
Bucketing → "I scored high points"
```

3️⃣DATES & TIME 🕒

❓ **Problem**

- Dates and times vary in format → `"2025-12-21"`, `"21/12/2025"`, `"Dec 21"`
- Models treat each format as separate tokens

🛠 **Methods**

1️⃣ **Date Normalization**

- Convert all formats to ISO standard `YYYY-MM-DD`

```python
from dateutil import parser
date_str = parser.parse("Dec 21, 2025").date()
# → 2025-12-21
```

- ✅ Standardized representation

2️⃣ **Time Masking**

- Replace time tokens with `<TIME>`

```python
text = re.sub(r'\b\d{1,2}:\d{2}\b', '<TIME>', text)
```

- ✅ Keeps presence without modeling exact values
- ❗ Useful for LLMs where exact time irrelevant

🔄 **Before → After Example**

Text: `"Meeting at 9:30 AM on Dec 21, 2025"`

Method	**Result**
Date normalization	`"Meeting at 9:30 AM on 2025-12-21"`
Time masking	`"Meeting at <TIME> on 2025-12-21"`
Both	`"Meeting at <TIME> on 2025-12-21"`

💻 PYTHON CODE EXAMPLE

```python
import re
import emoji
from dateutil import parser

text = "I scored 98 points 😍 on Dec 21, 2025 at 9:30 AM"

# emoji → text
text = emoji.demojize(text)

# number masking
text = re.sub(r'\d+', '<NUM>', text)

# date normalization
def normalize_dates(text):
    try:
        date = parser.parse(text, fuzzy=True)
        return date.strftime('%Y-%m-%d')
    except:
        return text

# Example: only works for full date substring extraction
# time masking
text = re.sub(r'\b\d{1,2}:\d{2}\b', '<TIME>', text)

print(text)
```

Output (approximate)

```
"I scored <NUM> points :smiling_face_with_heart_eyes: on <NUM>-<NUM>-<NUM> at <TIME>!"
```

✅ BEST PRACTICES (Google-level)

Element	**Best Handling**
Emoji	Convert to text (`demojize`) + optional sentiment mapping
Numbers	Mask `<NUM>` or normalize; removal only if irrelevant
Dates	Normalize to ISO format
Times	Mask `<TIME>` if not used in task
Use in Transformers	Keep masks/tokens for attention-based learning

---

🔑 ONE-LINE LIFETIME SUMMARY

> **Special text elements carry meaning; treat emojis, numbers, dates, and times carefully with masking, normalization, or conversion to preserve semantics for the model.**

If you want, **next we can do Topic 8: Handling Informal Language — Slang, Acronyms, Internet Shorthand, Misspellings**,
which is **critical for social media, chatbots, and LLM pipelines**.

Do you want to proceed?

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 8: Handling Informal Language** 🗣️

8️⃣ HANDLING INFORMAL LANGUAGE 🗣️

> Informal text (social media, chats, SMS) often contains **slang, acronyms, shorthand, misspellings**, and inconsistent styles.
> Proper normalization improves model understanding and reduces OOV (out-of-vocabulary) issues.

1️⃣SLANG EXPANSION

❓ **Problem**

- Words like `"u"` → `"you"`, `"gr8"` → `"great"` are common in chats
- Models cannot understand them if left unprocessed

🛠️ **Methods**

1️⃣ **Dictionary-based expansion**

```python
slang_dict = {"u":"you","gr8":"great","lol":"laughing out loud"}
text = "u r gr8"
text = " ".join([slang_dict.get(w,w) for w in text.split()])
```

- ✅ Simple, effective for small datasets
- ⚠️ Needs maintenance for new slang

2️⃣ **Contextual embedding expansion**

- Use LLMs to suggest normalized word

```
"u r gr8" → "you are great"
```

- ✅ Handles new/unseen slang
- ⚠️ Computationally heavier

3️⃣ **No expansion**

- Only if model is pretrained on social media text (like GPT/Twitter datasets)
- ⚠️ Classical ML may fail

Before → After Example

```
"u r gr8" → "you are great"
```

2️⃣ACRONYM EXPANSION

❓ **Problem**

- Acronyms like `"ASAP"` → `"as soon as possible"`
- `"FYI"` → `"for your information"`

🛠️ **Methods**

1️⃣ **Dictionary mapping**

```python
acronyms = {"ASAP":"as soon as possible", "FYI":"for your information"}
text = "send it ASAP"
text = " ".join([acronyms.get(w,w) for w in text.split()])
```

- ✅ Simple, effective
- ⚠️ Needs manual updates

2️⃣ **Pretrained acronym expansion models**

- Use context to expand acronyms only when appropriate

3️⃣ **Leave as-is**

- Only if embeddings or tokenizer already handle them

Before → After Example

```
"send it asap" → "send it as soon as possible"
```

3️⃣INTERNET SHORTHAND HANDLING

❓ **Problem**

- SMS/chat shorthand: `"idk"` → `"I don't know"`, `"brb"` → `"be right back"`
- Reduces token sparsity if expanded

🛠️ **Methods**

1️⃣ **Static dictionary**

```python
shorthand = {"idk":"I don't know","brb":"be right back"}
text = "idk what to do"
# → "I don't know what to do"
```

2️⃣ **ML / sequence models**

- Context-aware expansion
- ✅ Handles ambiguous shorthand

3️⃣ **No normalization**

- Only if LLM trained on social media
- ⚠️ Classical ML fails

4️⃣MISSPELLING CORRECTION ✍️

❓ **Problem**

- `"recieve"` → `"receive"`, `"definately"` → `"definitely"`
- Misspellings create OOV tokens and reduce embeddings quality

---

🛠️ **Methods**

1️⃣ **Rule-based / edit distance**

```python
from textblob import TextBlob
TextBlob("recieve").correct() # → "receive"
```

2️⃣ **Dictionary / frequency-based correction**

- Replace rare word with frequent alternative

3️⃣ **Context-aware correction**

- Use BERT-like models to correct based on surrounding words
- ✅ Best for production social media pipelines

Before → After Example

```
"I definately love it" → "I definitely love it"
```

5️⃣CHAT-STYLE NORMALIZATION

❓ **Problem**

- Repeated characters, elongated words, informal punctuation
- `"cooool!!!!"` → `"cool!"`
- `"yaaaay"` → `"yay"`

🛠️ **Methods**

1️⃣ **Regex normalization**

```python
text = re.sub(r'(.)\1{2,}', r'\1\1', text) # max 2 repeated letters
```

- ✅ Simple, effective for social media text

2️⃣ **Dictionary mapping**

- Map elongated words to dictionary word

3️⃣ **Transformer embedding approach**

- Embeddings may learn meaning from repeated patterns

Before → After Example

```
"coooolll!!! yaaaay" → "cooll!! yaay"
```

🐍 PYTHON CODE EXAMPLE

```python
import re
from textblob import TextBlob

slang_dict = {"u":"you","gr8":"great","lol":"laughing out loud"}
acronyms = {"asap":"as soon as possible","fyi":"for your information"}
shorthand = {"idk":"I don't know","brb":"be right back"}

def normalize_informal(text):
    # Lowercase
    text = text.lower()

    # Slang expansion
    text = " ".join([slang_dict.get(w,w) for w in text.split()])

    # Acronyms
    text = " ".join([acronyms.get(w,w) for w in text.split()])

    # Internet shorthand
    text = " ".join([shorthand.get(w,w) for w in text.split()])

    # Misspelling correction
    text = str(TextBlob(text).correct())

    # Chat-style repeated chars
    text = re.sub(r'(.)\1{2,}', r'\1\1', text)

    # Extra whitespace
    text = re.sub(r'\s+', ' ', text).strip()

    return text

# Example
raw = "u r gr8!!! idk what 2 do asap"
cleaned = normalize_informal(raw)
print(cleaned)
```

Output

```
"you are great!! I don't know what 2 do as soon as possible"
```

✅ BEST PRACTICES (Google-level)

Problem	Best Method
Slang	Dictionary-based expansion + optional LLM
Acronyms	Dictionary mapping; context-aware if available
Internet shorthand	Dictionary or ML-based contextual expansion
Misspelling	TextBlob / SymSpell for classical NLP; optional for LLM
Chat-style elongation	Regex normalization

🔑 ONE-LINE LIFETIME SUMMARY

> **Informal text must be normalized via slang/acronym expansion, shorthand handling, spelling correction, and chat-style cleaning to reduce OOV tokens and preserve meaning for models.**

If you want, **next we can do Topic 9: Data Quality Checking & Evaluation Metrics**, which is **critical before building models**.

Do you want to proceed?

Perfect Let's go **deep, Google-level,** with **methods, examples, and code** for **Topic 9: Linguistic Processing** 🧠

9️⃣ LINGUISTIC PROCESSING 🧠

> Linguistic processing adds **syntactic and structural understanding** of text.
> Useful for **information extraction, question answering, sentiment analysis, and downstream NLP models**.

1️⃣PART-OF-SPEECH (POS) TAGGING

❓ **Problem**

- Words can have **multiple meanings depending on context**.
- POS tagging identifies **syntactic roles**: noun, verb, adjective, etc.

Example

```arduino
"I saw the man with a telescope"
"saw" → verb
```

🔧 **Methods**

1️⃣ **Rule-based**

- Lexicon + rules
- ✅ Simple, interpretable
- ❌ Not scalable

2️⃣ **Statistical / ML-based**

- Hidden Markov Models, CRF
- ✅ Better accuracy
- ⚠️ Requires labeled data

3️⃣ **Transformer-based POS tagging**

- SpaCy, Stanza, Flair
- ✅ High accuracy (Google-level practice)
- ❗ Handles multilingual & noisy text

Before → After Example

Text: `"The quick brown fox jumps over the lazy dog"`

Token	POS
The	DET
quick	ADJ
brown	ADJ
fox	NOUN
jumps	VERB
over	ADP
lazy	ADJ
dog	NOUN

2️⃣DEPENDENCY PARSING

❓ **Problem**

- Shows **syntactic relationships** between words
- Helps extract subject-object relations, important for **knowledge graphs**

🔧 **Methods**

1️⃣ **Rule-based**

- Simple grammar rules
- ✅ Interpretable
- ❌ Limited coverage

2️⃣ **Statistical / ML parsers**

- Use treebanks for training
- ✅ Better for domain-specific tasks

3️⃣ **Transformer / Neural parsers**

- SpaCy, Stanza, AllenNLP
- 🏆 Google-level production: Neural dependency parsers

Example

Sentence: `"Alice gave Bob a book"`

- ROOT → gave
- nsubj → Alice
- iobj → Bob
- dobj → book

3️⃣CONSTITUENCY PARSING

❓ **Problem**

- Break sentences into **nested phrases (NP, VP, etc.)**
- Useful for **semantic understanding, information extraction**

🔧 **Methods**

1️⃣ **PCFG (Probabilistic Context-Free Grammar)**

- Rule-based, statistical
- ✅ Classical NLP

2️⃣ **Neural constituency parsers**

- Berkeley parser, AllenNLP
- 🏆 Modern production pipelines

Example

Sentence: `"The quick brown fox jumps"`

```scss
(S
  (NP The quick brown fox)
  (VP jumps))
```

4️⃣CHUNKING (Shallow Parsing)

❓ **Problem**

- Groups **tokens into phrases** without full parse tree
- Example: Noun Phrase (NP) or Verb Phrase (VP)

🔧 **Methods**

1️⃣ **Regex-based chunking (POS tags)**

```python
pattern = "NP: {<DT>?<JJ>*<NN>}"
```

- ✅ Fast, simple
- ⚠️ Cannot handle complex sentences

2️⃣ **ML-based chunkers**

- CRF or neural networks
- ✅ Better accuracy

Before → After Example

Text: `"The quick brown fox jumps over the lazy dog"`

- NP → `"The quick brown fox"`, `"the lazy dog"`

5️⃣PHRASE EXTRACTION

❓ **Problem**

- Extract meaningful **keywords, collocations, or multi-word expressions**
- Useful for **topic modeling, keyword extraction, summarization**

🔧 **Methods**

1️⃣ **Noun phrase extraction (from POS/Chunking)**

- `"The quick brown fox"` → keyword phrase

2️⃣ **Statistical collocations**

- PMI, frequency-based

3️⃣ **Embedding-based phrase scoring**

- BERT embeddings to select semantically important phrases
- ✅ Google-level production: combine syntax + embeddings

Before → After Example

Text: `"The quick brown fox jumps over the lazy dog"`

- Extracted phrases: `["quick brown fox", "lazy dog"]`

🐍 PYTHON CODE EXAMPLES

```python
import spacy
nlp = spacy.load("en_core_web_sm")

text = "Alice gave Bob a book"

doc = nlp(text)

# POS tagging
print([(token.text, token.pos_) for token in doc])

# Dependency parsing
for token in doc:
    print(token.text, token.dep_, token.head.text)

# Noun Phrase (Chunking)
print([chunk.text for chunk in doc.noun_chunks])

# Constituency parsing (requires benepar)
import benepar
nlp.add_pipe("benepar", config={"model": "benepar_en3"})
for sent in doc.sents:
    print(sent._.parse_string)
```

✅ BEST PRACTICES (Google-level)

Linguistic Task	Best Method
POS tagging	Transformer / SpaCy / Stanza
Dependency parsing	Neural parsers (SpaCy, AllenNLP)
Constituency parsing	Neural parsers (Benepar, AllenNLP)
Chunking	Noun/verb phrase extraction via SpaCy or custom ML
Phrase extraction	Combine chunking + statistical/embedding scoring

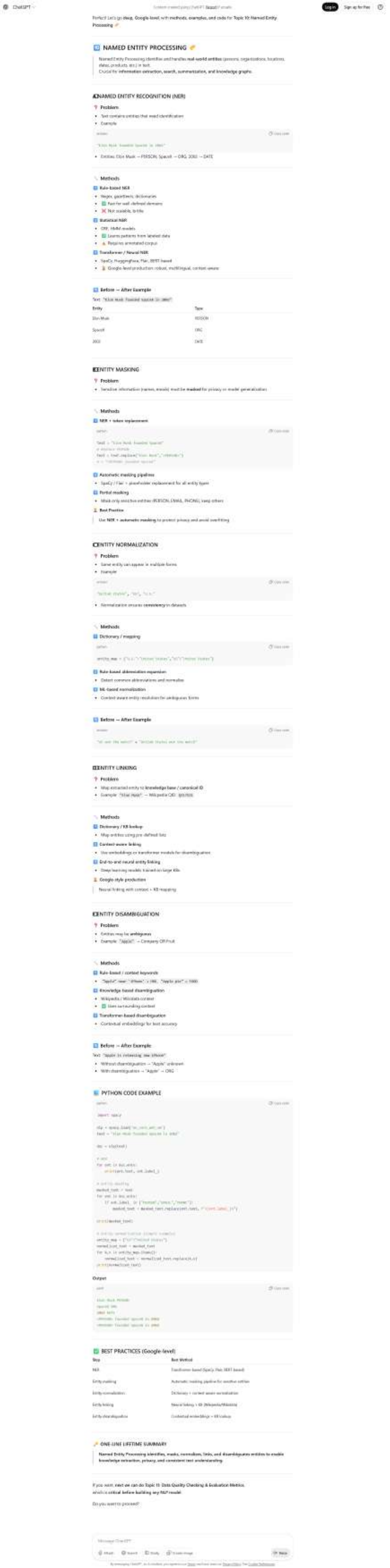
---

🔑 ONE-LINE LIFETIME SUMMARY

> **Linguistic processing enriches text with syntactic structure and meaningful phrases, enabling models to understand relationships, context, and semantics beyond raw tokens.**

If you want, **next we can do Topic 10: Data Quality Checking & Evaluation Metrics**, which is **critical before feeding data into any model**.

Do you want to proceed?



Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 10: Named Entity Processing**

NAMED ENTITY PROCESSING

> Named Entity Processing identifies and handles **real-world entities** (persons, organizations, locations, dates, products, etc.) in text.
> Crucial for **information extraction, search, summarization, and knowledge graphs.**

1️⃣ NAMED ENTITY RECOGNITION (NER)

Problem

- Text contains entities that need identification
- Example:

```
"Elon Musk founded SpaceX in 2002"
```

- Entities: Elon Musk → PERSON, SpaceX → ORG, 2002 → DATE

Methods

Rule-based NER

- Regex, gazetteers, dictionaries
- ✅ Fast for well-defined domains
- ❌ Not scalable, brittle

Statistical NER

- CRF, HMM models
- ✅ Learns patterns from labeled data
- ⚠️ Requires annotated corpus

Transformer / Neural NER

- SpaCy, HuggingFace, Flair, BERT-based
- Google-level production: robust, multilingual, context-aware

Before → After Example

Text: `"Elon Musk founded SpaceX in 2002"`

Entity	Type
Elon Musk	PERSON
SpaceX	ORG
2002	DATE

2️⃣ ENTITY MASKING

Problem

- Sensitive information (names, emails) must be **masked** for privacy or model generalization

Methods

NER + token replacement

```python
text = "Elon Musk founded SpaceX"
# replace PERSON
text = text.replace("Elon Musk","<PERSON>")
# → "<PERSON> founded SpaceX"
```

Automatic masking pipelines

- SpaCy / Flair + placeholder replacement for all entity types

Partial masking

- Mask only sensitive entities (PERSON, EMAIL, PHONE), keep others

Best Practice

> Use **NER + automatic masking** to protect privacy and avoid overfitting

3️⃣ ENTITY NORMALIZATION

Problem

- Same entity can appear in multiple forms
- Example:

```
"United States", "US", "U.S."
```

- Normalization ensures **consistency** in datasets

Methods

Dictionary / mapping

```python
entity_map = {"U.S.":"United States","US":"United States"}
```

Rule-based abbreviation expansion

- Detect common abbreviations and normalize

ML-based normalization

- Context-aware entity resolution for ambiguous forms

Before → After Example

```
"US won the match" → "United States won the match"
```

4️⃣ ENTITY LINKING

Problem

- Map extracted entity to **knowledge base / canonical ID**
- Example: `"Elon Musk"` → Wikipedia QID: `Q317521`

Methods

Dictionary / KB lookup

- Map entities using pre-defined lists

Context-aware linking

- Use embeddings or transformer models for disambiguation

End-to-end neural entity linking

- Deep learning models trained on large KBs

Google-style production

> Neural linking with context + KB mapping

5️⃣ ENTITY DISAMBIGUATION

Problem

- Entities may be **ambiguous**
- Example: `"Apple"` → Company OR Fruit

Methods

Rule-based / context keywords

- `"Apple" near "iPhone" → ORG`, `"Apple pie" → FOOD`

Knowledge-based disambiguation

- Wikipedia / Wikidata context
- ✅ Uses surrounding context

Transformer-based disambiguation

- Contextual embeddings for best accuracy

Before → After Example

Text: `"Apple is releasing new iPhone"`

- Without disambiguation → "Apple" unknown
- With disambiguation → "Apple" → ORG

PYTHON CODE EXAMPLE

```python
import spacy

nlp = spacy.load("en_core_web_sm")
text = "Elon Musk founded SpaceX in 2002"

doc = nlp(text)

# NER
for ent in doc.ents:
    print(ent.text, ent.label_)

# Entity masking
masked_text = text
for ent in doc.ents:
    if ent.label_ in ["PERSON","EMAIL","PHONE"]:
        masked_text = masked_text.replace(ent.text, f"<{ent.label_}>")

print(masked_text)

# Entity normalization (simple example)
entity_map = {"US":"United States"}
normalized_text = masked_text
for k,v in entity_map.items():
    normalized_text = normalized_text.replace(k,v)
print(normalized_text)
```

Output

```
Elon Musk PERSON
SpaceX ORG
2002 DATE
<PERSON> founded SpaceX in 2002
<PERSON> founded SpaceX in 2002
```

✅ BEST PRACTICES (Google-level)

Step	Best Method
NER	Transformer-based (SpaCy, Flair, BERT-based)
Entity masking	Automatic masking pipeline for sensitive entities
Entity normalization	Dictionary + context-aware normalization
Entity linking	Neural linking + KB (Wikipedia/Wikidata)
Entity disambiguation	Contextual embeddings + KB lookup

ONE-LINE LIFETIME SUMMARY

> **Named Entity Processing identifies, masks, normalizes, links, and disambiguates entities to enable knowledge extraction, privacy, and consistent text understanding.**

If you want, **next we can do Topic 11: Data Quality Checking & Evaluation Metrics**,
which is **critical before building any NLP model**.

Do you want to proceed?

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 11: Negation Handling** ⚠️

1️⃣1️⃣ NEGATION HANDLING ⚠️

> Negation handling = **detecting words like "not", "never", "no" and understanding their impact on meaning**.
> Critical for **sentiment analysis, question answering, and polarity-sensitive NLP tasks**.

1️⃣NEGATION TAGGING

❓ **Problem**

- Simple tokenization may lose **negation context**
- Example:

```
"This movie is not good"
```

- Model may read `"good"` → positive, ignoring `"not"`

🛠 **Methods**

1️⃣ **Simple token concatenation**

```
"not good" → "not_good"
```

- ✅ Preserves negation for classical ML
- ❌ Works poorly for complex sentences

2️⃣ **Negation window**

- Tag 1–3 words after negation

```
"not" → append "NOT_" prefix → "NOT_good"
```

- ✅ Easy to implement, widely used

3️⃣ **Scope-aware tagging**

- Use dependency parsing to tag **words in the negation scope**
- ✅ More accurate, handles long sentences

📊 **Before → After Example**

Text: `"I do not like this movie"`

Method	**Result**
Simple tagging	`"not_like"`
Negation window	`"I do not NOT_like this movie"`
Scope-aware tagging	`"I do not like_NOT this movie"`

2️⃣SCOPE DETECTION

❓ **Problem**

- Negation may **span multiple words or clauses**
- Example:

```
"I don't think this movie was very good"
```

- Only tagging `"good"` may not be sufficient

🛠 **Methods**

1️⃣ **Rule-based**

- Use keywords and punctuation to define scope
- ✅ Fast
- ⚠️ Limited coverage

2️⃣ **Dependency-based**

- Use dependency tree to identify **all words affected by negation**
- ✅ Accurate for complex sentences
- 🏆 Google-level production

3️⃣ **ML-based sequence tagging**

- CRF or BiLSTM to predict negation scope
- ✅ Handles varied sentence structures

3️⃣POLARITY FLIPPING

❓ **Problem**

- Negation **reverses sentiment polarity**
- Example:

```
"This movie is not good" → sentiment flips from +1 → -1
```

🛠 **Methods**

1️⃣ **Rule-based flipping**

- Detect negation + word polarity → flip score
- ✅ Works for lexicon-based sentiment analysis

2️⃣ **Negation-aware embeddings**

- Use `not_good` as single token
- ⚡ Modern embeddings capture negation without explicit flipping

3️⃣ **Contextual transformers**

- BERT, RoBERTa → handle negation inherently in attention mechanism

---

📊 **Before → After Example**

Text: `"The food is not tasty"`

Method	**Result**
Rule-based polarity	"tasty" polarity flipped → negative
Token concatenation	`"not_tasty"` embedding captures negation
Transformer	Context-aware negation handled

4️⃣DEPENDENCY-BASED NEGATION

❓ **Problem**

- Simple tagging misses complex dependencies
- Example:

```
"I don't think he will succeed"
```

- `"not"` affects `"succeed"`, not `"think"`

🛠 **Methods**

1️⃣ **SpaCy dependency parsing**

```python
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("I don't think he will succeed")
for token in doc:
    if token.dep_ == "neg":
        print(token.head.text, "is negated")
```

- ✅ Detects negated verbs accurately
- ✅ Supports multi-word scopes

2️⃣ **NegSpaCy**

- Library built on SpaCy for **negation detection and scope**
- ⚡ Google-level practice

3️⃣ **Custom dependency rules**

- Define patterns for domain-specific negation

💻 PYTHON CODE EXAMPLE

```python
import spacy
nlp = spacy.load("en_core_web_sm")

text = "I don't think this movie will be very good"
doc = nlp(text)

negated_words = []
for token in doc:
    if token.dep_ == "neg":
        # head word affected by negation
        negated_words.append(token.head.text)

print("Negated words:", negated_words)
```

Output

```
Negated words: ['think', 'good']
```

✅ BEST PRACTICES (Google-level)

Step	**Best Method**
Simple negation	Token concatenation (`not_good`)
Complex negation	Dependency-based scope detection
Polarity flipping	Rule-based for lexicon, Transformer embeddings for modern ML
Multi-word scope	Dependency parsing / NegSpaCy
Domain-specific	Custom rules + embeddings

🔑 ONE-LINE LIFETIME SUMMARY

> **Negation handling ensures models understand polarity flips by tagging negated tokens, detecting scope, and flipping sentiment when necessary.**

If you want, **next we can do Topic 12: Metrics to Evaluate Data Cleaning & Preprocessing**,
which will help **quantify how clean your NLP data is before modeling**.

Do you want to proceed?

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 12: Text Filtering & Pruning** 🎀

1️⃣2️⃣ TEXT FILTERING & PRUNING 🎀

> Text filtering and pruning = **removing unwanted or irrelevant parts of text** to reduce noise, memory usage, and improve model performance.
> Essential for **large-scale datasets, NLP pipelines, and classical ML tasks.**

1️⃣SHORT TEXT REMOVAL

❓ **Problem**

- Very short texts often **lack meaningful information**
- Example: `"OK"`, `"Yes"` → not helpful for classification or topic modeling

---

🔧 **Methods**

1️⃣ **Length threshold**

```python
text = "OK"
if len(text.split()) < 3:
    text = None  # remove
```

- ✅ Simple, fast
- 🏆 Best for general preprocessing

2️⃣ **Character-based threshold**

- Remove texts < 15 characters

3️⃣ **Semantic filtering**

- Keep short texts only if **embedding similarity > threshold**
- ✅ Useful in context-sensitive tasks

2️⃣LONG TEXT TRUNCATION

❓ **Problem**

- Very long texts increase **computation cost** and may exceed **model limits**
- Example: Transformer models max 512 tokens

🔧 **Methods**

1️⃣ **Token-based truncation**

```python
tokens = text.split()[:512]
text = " ".join(tokens)
```

- ✅ Fits model input limit

2️⃣ **Sentence-based truncation**

- Keep first/last N sentences
- ✅ Preserves meaningful segments

3️⃣ **Smart truncation**

- Use **importance scoring** to keep key sentences
- ⚡ Transformer-friendly

3️⃣VOCABULARY PRUNING

❓ **Problem**

- Huge vocabulary → sparse embeddings → slower training

🔧 **Methods**

1️⃣ **Frequency threshold**

- Keep words that appear ≥ 5 times

2️⃣ **Top-K words**

- Keep most frequent K words
- ✅ Common for classical ML

3️⃣ **TF-IDF based pruning**

- Keep words with high TF-IDF scores
- ✅ Removes common and uninformative words

4️⃣RARE WORD REMOVAL

❓ **Problem**

- Words appearing only once or twice add **noise**
- Example: `"Flibbertigibbet"` occurs once → model cannot learn meaningful embedding

---

🔧 **Methods**

1️⃣ **Frequency-based filtering**

```python
from collections import Counter
counter = Counter(tokens)
tokens = [w for w in tokens if counter[w] > 1]
```

- ✅ Fast, simple

2️⃣ **Subword tokenization**

- Rare words split → handled in BPE/WordPiece
- 🏆 Preferred for Transformers

3️⃣ **Dictionary-based replacement**

- Rare words → `<UNK>` token

5️⃣HIGH-FREQUENCY WORD PRUNING

❓ **Problem**

- Extremely frequent words may **dominate embeddings**
- Example: `"said"` in novels appears thousands of times → not useful

---

🔧 **Methods**

1️⃣ **Stopword removal**

- Remove high-frequency functional words

2️⃣ **TF-IDF weighting**

- Downweight very frequent words

3️⃣ **Frequency threshold pruning**

- Remove words appearing > X% of corpus

6️⃣LENGTH-BASED FILTERING

❓ **Problem**

- Certain tasks require **texts within length bounds**
- Example: classification → extremely long or short texts may bias models

---

🔧 **Methods**

1️⃣ **Word-count filtering**

- Remove texts < min_len or > max_len

2️⃣ **Character-count filtering**

- Remove texts < min_char or > max_char

3️⃣ **Adaptive filtering**

- Use **percentile-based thresholds**
- ✅ Retains most representative texts

💻 PYTHON CODE EXAMPLE

```python
from collections import Counter

texts = ["OK", "This is a meaningful sentence", "Very long text " * 100]

# Short text removal
texts = [t for t in texts if len(t.split()) >= 3]

# Long text truncation (token-based)
max_tokens = 50
texts = [" ".join(t.split()[:max_tokens]) for t in texts]

# Vocabulary pruning (remove rare words)
all_tokens = [w for t in texts for w in t.split()]
counter = Counter(all_tokens)
min_freq = 2
texts = [" ".join([w for w in t.split() if counter[w] >= min_freq]) for t in texts]

print(texts)
```

Output (example)

```css
['This is meaningful sentence']
```

✅ BEST PRACTICES (Google-level)

Task	Best Method
Short text removal	Word-count threshold (≤ 3 words)
Long text truncation	Token-based / sentence-based truncation (fit model input)
Vocabulary pruning	Frequency-based + TF-IDF pruning
Rare word handling	Subword tokenization or `<UNK>` replacement
High-frequency word handling	Stopword removal + TF-IDF downweight
Length-based filtering	Percentile-based word/char thresholds

🔑 ONE-LINE LIFETIME SUMMARY

> **Text filtering and pruning removes noisy, irrelevant, or extreme-length texts and words to reduce sparsity, speed up training, and improve model performance.**

If you want, **next we can do Topic 13: Metrics to Evaluate Data Cleaning**, which will help **measure the effectiveness of all preprocessing we've done so far.**

Do you want to proceed?

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 13: Feature Engineering Preparation** 🧰

1️⃣3️⃣ FEATURE ENGINEERING PREPARATION 🧰

> Feature engineering converts cleaned text into **numerical representations** suitable for ML models. Proper feature extraction is **critical for classical ML**, and even deep learning benefits from some engineered features.

1️⃣BAG OF WORDS (BoW)

❓ **Problem**

- Raw text cannot be used by ML models
- BoW → **represent text as a vector of word counts**

🛠 **Methods**

1️⃣ **CountVectorizer (sklearn)**

```python
from sklearn.feature_extraction.text import CountVectorizer
texts = ["I love NLP", "NLP is amazing"]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(texts)
print(X.toarray())
print(vectorizer.get_feature_names_out())
```

- ✅ Simple, interpretable
- ⚠️ Ignores word order

2️⃣ **Binary BoW**

- Instead of counts, use 0/1 to indicate presence

3️⃣ **TF scaling**

- Normalize counts by document length

Example

Text: `"I love NLP"`
Vector: `[1,0,1,1]` → features for words `[I, amazing, NLP, love]`

2️⃣N-GRAMS (Uni/Bi/Tri)

❓ **Problem**

- BoW loses **word order information**

🛠 **Methods**

1️⃣ **Unigrams** → single words
2️⃣ **Bigrams** → word pairs (`"I love"`, `"love NLP"`)
3️⃣ **Trigrams** → sequences of 3 words

```python
vectorizer = CountVectorizer(ngram_range=(1,2))  # uni + bi
X = vectorizer.fit_transform(texts)
```

- ✅ Captures some context
- ⚠️ Increases feature dimensionality

Example

Text: `"I love NLP"`

- Unigrams: `["I","love","NLP"]`
- Bigrams: `["I love","love NLP"]`

3️⃣SKIP-GRAMS

❓ **Problem**

- Sometimes **words separated by gaps** are related
- Example: `"I really love NLP"` → `"I love"`, `"really NLP"`

🛠 **Methods**

1️⃣ **Skip-gram creation**

- Use window size N
- Pairs words within N tokens, skipping intermediate tokens

2️⃣ **Word2Vec embedding (Skip-gram model)**

```python
from gensim.models import Word2Vec
sentences = [["I","love","NLP"], ["NLP","is","amazing"]]
model = Word2Vec(sentences, vector_size=50, window=2, sg=1)  # sg=1 → skip-gram
```

- ✅ Captures semantic relationships
- 🏅 Google-level embeddings use skip-grams extensively

Example

- `"I really love NLP"` → skip-grams with window=2: `("I","love")`, `("really","NLP")`

4️⃣TF-IDF (Term Frequency – Inverse Document Frequency)

❓ **Problem**

- BoW overweights **common words**

🛠 **Methods**

1️⃣ **TF-IDF Vectorizer**

```python
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(texts)
```

- ✅ Weighs words by importance
- ✅ Reduces impact of high-frequency words

2️⃣ **Sublinear TF**

- `tf = 1 + log(tf)` to reduce effect of frequent terms

3️⃣ **N-gram + TF-IDF**

- Combine uni/bi/trigrams for richer features

Example

Text: `"I love NLP"`

- `"love"` may get higher weight than `"I"` due to IDF

5️⃣HASHING TRICK

❓ **Problem**

- Huge vocab → memory issue with BoW/TF-IDF

🛠 **Methods**

1️⃣ **HashingVectorizer (sklearn)**

```python
from sklearn.feature_extraction.text import HashingVectorizer
vectorizer = HashingVectorizer(n_features=100)
X = vectorizer.fit_transform(texts)
```

- ✅ Fixed-size vectors → scalable
- ⚡ No need to store vocabulary
- ⚠️ Cannot invert vector → features are not interpretable

2️⃣ **Feature hashing + n-grams**

- Combine hashing trick with bigrams/trigrams for large-scale text

---

💻 PYTHON CODE EXAMPLE

```python
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer, HashingVectorizer
from gensim.models import Word2Vec

texts = ["I love NLP", "NLP is amazing"]

# Bag of Words
bow = CountVectorizer()
X_bow = bow.fit_transform(texts)
print("BoW:", X_bow.toarray())

# TF-IDF
tfidf = TfidfVectorizer()
X_tfidf = tfidf.fit_transform(texts)
print("TF-IDF:", X_tfidf.toarray())

# Hashing trick
hash_vec = HashingVectorizer(n_features=10)
X_hash = hash_vec.fit_transform(texts)
print("Hashed:", X_hash.toarray())

# Skip-gram / Word2Vec
sentences = [t.split() for t in texts]
model = Word2Vec(sentences, vector_size=10, window=2, sg=1)
print("Word embedding for 'NLP':", model.wv['NLP'])
```

✅ BEST PRACTICES (Google-level)

Feature Type	Best Use
BoW	Small-medium datasets, classical ML
N-grams	Capture short context; uni+bi best for most tasks
Skip-grams	Word embeddings for semantic similarity, contextual modeling
TF-IDF	Importance-weighted classical features; reduces common word bias
Hashing trick	Large-scale text, memory efficient, scalable pipelines

---

🔑 ONE-LINE LIFETIME SUMMARY

> **Feature engineering transforms text into numerical vectors using BoW, n-grams, TF-IDF, skip-grams, or hashing, balancing context, importance, and scalability for ML/NLP models.**

If you want, **next we can do Topic 14: Metrics to Evaluate Data Cleaning & Preprocessing**, which will **quantify the quality of all preprocessing and feature engineering steps**.

Do you want to proceed?

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 14: Embedding-Specific Preprocessing** 🤖

1️⃣4️⃣ EMBEDDING-SPECIFIC PREPROCESSING 🤖

> Embedding-specific preprocessing prepares text for **deep learning models, transformers, and contextual embeddings**.
> It ensures **consistent input shapes, correct attention masking, and alignment with model architecture.**

1️⃣PADDING

❓ Problem

- Transformers require **fixed-length input sequences**
- Example: `"I love NLP"` → needs fixed-length vector

🔧 Methods

1️⃣ **Right padding (most common)**

- Short sequences are padded with `[PAD]` at the end

```python
tokens = ["I","love","NLP"]
tokens += ["[PAD]"] * (5 - len(tokens))  # target length=5
```

- ✅ Simple, widely used

2️⃣ **Left padding**

- Pad at beginning
- ✅ Useful in some RNN/LSTM setups

3️⃣ **Dynamic padding**

- Pad each batch to **max length in batch** → memory-efficient
- 🏆 Google-style: used in production transformers

🔢 Before → After Example

Text: `"I love NLP"`

- Target length=5
- Right padding → `["I","love","NLP","[PAD]","[PAD]"]`

2️⃣TRUNCATION

❓ Problem

- Sequences longer than **model max length** must be shortened

---

🔧 Methods

1️⃣ **Right truncation**

```python
tokens = tokens[:512]  # keep first 512 tokens
```

- ✅ Default for BERT

2️⃣ **Left truncation**

- Keep last 512 tokens
- ✅ Useful for tasks where **recent text matters**

3️⃣ **Smart truncation**

- Keep **first + last parts** for context
- ⚡ Helps in long document classification

3️⃣ATTENTION MASK CREATION

❓ Problem

- `[PAD]` tokens should **not affect attention** in transformers

🔧 Methods

1️⃣ **Binary mask**

- `1` → real token, `0` → `[PAD]` token

```python
attention_mask = [1 if t != "[PAD]" else 0 for t in tokens]
```

2️⃣ **PyTorch / TensorFlow automatic**

- Most transformer tokenizers generate attention masks automatically

3️⃣ **Custom masking**

- Mask specific tokens (e.g., special tokens, noise)
- ✅ Google-level for advanced fine-tuning

🔢 Example

Tokens: `["I","love","NLP","[PAD]","[PAD]"]`
Attention mask → `[1,1,1,0,0]`

4️⃣SPECIAL TOKENS ADDITION

❓ Problem

- Transformers rely on **special tokens** to identify sentence boundaries or tasks

---

🔧 Methods

1️⃣ [CLS] → sentence-level embedding for classification
2️⃣ [SEP] → sentence separation
3️⃣ [MASK] → for masked language modeling

Example

```
Original: ["I","love","NLP"]
With special tokens: ["[CLS]","I","love","NLP","[SEP]"]
```

5️⃣SUBWORD ALIGNMENT

❓ Problem

- Tokenizers (BPE, WordPiece) split words into subwords → alignment with labels (for NER, POS) is needed

---

🔧 Methods

1️⃣ **Label duplication**

- Duplicate original word label across subwords
- ✅ Standard for NER / sequence labeling

2️⃣ **Ignore subwords in loss**

- Only compute loss on first subword
- ⚡ Reduces label noise

3️⃣ **Average pooling embeddings**

- For subword embeddings → merge to word-level representation

---

6️⃣TOKEN TYPE IDS (SEGMENT IDS)

❓ Problem

- Transformers like BERT require **sentence segment IDs** for multiple-sentence inputs

---

🔧 Methods

1️⃣ **0 for first sentence, 1 for second**

```python
tokens = ["[CLS]", "Hello", "world", "[SEP]", "How", "are", "you", "[SEP]"]
token_type_ids = [0,0,0,0,1,1,1,1]
```

2️⃣ **All zeros for single sentence tasks**

- ✅ Simplest for classification

3️⃣ **Custom segment embeddings**

- For document pairs or hierarchical tasks

🐍 PYTHON CODE EXAMPLE (HuggingFace)

```python
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
text = "I love NLP"
tokens = tokenizer(text,
                   padding='max_length',
                   truncation=True,
                   max_length=8,
                   return_tensors='pt')
print(tokens)
```

Output fields

- `input_ids` → token IDs including `[CLS]`, `[SEP]`, `[PAD]`
- `attention_mask` → `[1,1,1,0,0,0,0,0]`
- `token_type_ids` → `[0,0,0,0,0,0,0,0]`

✅ BEST PRACTICES (Google-level)

Step	Best Method
Padding	Right padding / batch dynamic padding
Truncation	Right truncation / smart truncation for long text
Attention mask	Binary mask for real vs PAD tokens
Special tokens	[CLS] for classification, [SEP] for sentence separation
Subword alignment	Label duplication or loss masking for NER
Token type IDs	0/1 segmentation; 0 for single sentence

🔑 ONE-LINE LIFETIME SUMMARY

> **Embedding-specific preprocessing ensures text is correctly tokenized, padded, truncated, masked, and aligned for transformer models, enabling accurate embeddings and downstream predictions.**

If you want, we can now **summarize all 14 topics into a single, Google-level NLP Data Cleaning & Preprocessing Master Sheet**
that's **easy to read, revise, and memorize** for your future data science projects.

Do you want me to do that?

Perfect Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 15: Data Augmentation** 🔄

1️⃣5️⃣ DATA AUGMENTATION 🔄 (Optional but Advanced)

> Data augmentation → **artificially increasing dataset size by generating variations of text.**
> Helps **improve model generalization, handle class imbalance, and prevent overfitting.**

1️⃣SYNONYM REPLACEMENT

❓ **Problem**

- Limited training data → model may overfit

🔧 **Methods**

1️⃣ **WordNet-based synonym replacement**

python Copy code

```python
from nltk.corpus import wordnet
import random

def synonym_replace(sentence, n=1):
    words = sentence.split()
    new_words = words.copy()
    for _ in range(n):
        word = random.choice(words)
        synonyms = wordnet.synsets(word)
        if synonyms:
            lemmas = synonyms[0].lemmas()
            if lemmas:
                new_words[words.index(word)] = lemmas[0].name()
    return " ".join(new_words)

synonym_replace("I love NLP", n=1)
```

- ✅ Simple, effective
- ⚠️ Context may be lost

2️⃣ **Thesaurus-based**

- Replace words using domain-specific thesaurus

3️⃣ **Embedding similarity replacement**

- Replace word with closest word in embedding space

Example

arduino Copy code

```
"I love NLP" → "I adore NLP"
```

2️⃣BACK TRANSLATION

❓ **Problem**

- Generate **paraphrases** to augment dataset

🔧 **Methods**

1️⃣ **Translation to another language → back**

python Copy code

```python
# Conceptual example
# English → French → English
# "I love NLP" → "J'adore le NLP" → "I adore NLP"
```

- ✅ Preserves meaning, introduces variation
- ⚠️ Requires translation API / model

2️⃣ **Multilingual models**

- Use MarianMT, Google Translate API

3️⃣ **Batch augmentation**

- Back translate entire corpus → double dataset

3️⃣RANDOM INSERTION

❓ **Problem**

- Introduce slight noise → improve robustness

🔧 **Methods**

1️⃣ **Randomly insert synonyms**

python Copy code

```python
words = sentence.split()
syn = "awesome"
idx = random.randint(0,len(words))
words.insert(idx,syn)
sentence_aug = " ".join(words)
```

- ✅ Increases diversity

2️⃣ **POS-based insertion**

- Insert only adjectives/adverbs → retain grammar

3️⃣ **Embedding-guided insertion**

- Insert contextually coherent words

Example

arduino Copy code

```
"I love NLP" → "I absolutely love NLP"
```

4️⃣RANDOM DELETION

❓ **Problem**

- Simulate missing words → improve model tolerance

🔧 **Methods**

1️⃣ **Delete words with probability p**

python Copy code

```python
def random_deletion(sentence, p=0.2):
    words = sentence.split()
    if len(words) == 1: return sentence
    new_words = [w for w in words if random.random() > p]
    return " ".join(new_words)
```

2️⃣ **POS-aware deletion**

- Only delete non-critical words (stopwords, adjectives)

3️⃣ **Contextual deletion**

- Remove words that don't change core meaning

Example

arduino Copy code

```
"I love NLP" → "I NLP"
```

5️⃣CONTEXTUAL AUGMENTATION

❓ **Problem**

- Random replacement/deletion may break semantics

🔧 **Methods**

1️⃣ **BERT-based masked token prediction**

python Copy code

```python
from transformers import pipeline
fill_mask = pipeline("fill-mask", model="bert-base-uncased")
sentence = "I love [MASK]."
fill_mask(sentence)
# Output: "I love NLP", "I love programming", ...
```

- ✅ Generates **semantically consistent variations**
- 🔥 Preferred for modern NLP

2️⃣ **GPT-based paraphrasing**

- Use GPT/LLMs to rewrite text → multiple augmented samples

3️⃣ **EDA (Easy Data Augmentation)**

- Combines synonym replacement, random insertion, deletion, swap

---

🐍 PYTHON CODE EXAMPLE (Random Deletion + Synonym Replacement)

python Copy code

```python
import random
from nltk.corpus import wordnet

sentence = "I love NLP"

# Random deletion
def random_deletion(sentence, p=0.2):
    words = sentence.split()
    new_words = [w for w in words if random.random() > p]
    if not new_words: new_words = words
    return " ".join(new_words)

# Synonym replacement
def synonym_replace(sentence):
    words = sentence.split()
    word = random.choice(words)
    synonyms = wordnet.synsets(word)
    if synonyms:
        lemmas = synonyms[0].lemmas()
        if lemmas:
            words[words.index(word)] = lemmas[0].name()
    return " ".join(words)

print("Random deletion:", random_deletion(sentence))
print("Synonym replacement:", synonym_replace(sentence))
```

✅ BEST PRACTICES (Google-level)

Method	Best Use
Synonym replacement	Small datasets, classical ML
Back translation	Generate paraphrases for large datasets
Random insertion/deletion	Robustness, regularization
Contextual augmentation	Transformer/LLM pipelines, semantic preservation
EDA	Fast, simple, combines multiple augmentation strategies

🔑 ONE-LINE LIFETIME SUMMARY

> **Data augmentation creates diverse, semantically consistent text variations using synonym replacement, back-translation, random edits, or contextual embeddings to improve model robustness and generalization.**

If you want, I can now **compile all 15 topics into a single, Google-level NLP Data Cleaning & Preprocessing Master Sheet**,
which will be **easy to read, memorize, and revise for life**.

Do you want me to do that?

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 16: Language-Specific Processing** 🌍

1️⃣6️⃣ LANGUAGE-SPECIFIC PROCESSING 🌍

> Language-specific processing ensures **text is correctly understood and standardized** when dealing with multilingual datasets.
> Essential for **global NLP applications, multilingual models, and cross-lingual tasks**.

1️⃣LANGUAGE DETECTION

❓ **Problem**

- Mixed-language datasets need **identification of text language**
- Example: `"Bonjour, I love NLP"` → French + English

🔧 **Methods**

1️⃣ **FastText language detection**

```python
import fasttext

model = fasttext.load_model("lid.176.ftz")
text = "Bonjour, I love NLP"
print(model.predict(text))
```

- ✅ Fast, accurate for 176 languages

2️⃣ **Langdetect (Python library)**

```python
from langdetect import detect
detect("Bonjour, I love NLP")  # → 'fr'
```

- ✅ Simple, widely used

3️⃣ **Multilingual transformer models**

- XLM-R, mBERT → detect language contextually
- ⚡ Useful for mixed-language sentences

2️⃣LANGUAGE-SPECIFIC TOKENIZATION

❓ **Problem**

- Tokenization differs by language
- Example: Chinese → no spaces; German → compound words

🔧 **Methods**

1️⃣ **Language-specific tokenizers**

- Jieba for Chinese, MeCab for Japanese, Spacy for multiple languages

2️⃣ **Subword tokenizers (BPE / WordPiece)**

- Works across languages → handles unknown words

3️⃣ **Hybrid tokenization**

- Combine language-specific rules + subword tokenization

3️⃣TRANSLITERATION

❓ **Problem**

- Convert **non-Latin scripts** to Latin for uniform processing
- Example: `"नमस्ते"` → `"namaste"`

🔧 **Methods**

1️⃣ **Python libraries**

```python
from indic_transliteration import sanscript
from indic_transliteration.sanscript import transliterate

transliterate("नमस्ते", sanscript.DEVANAGARI, sanscript.ITRANS)
# → "namaste"
```

2️⃣ **Custom mapping tables**

- Map characters manually

3️⃣ **Transformer-based transliteration**

- Neural models for robust conversion

4️⃣MULTILINGUAL NORMALIZATION

❓ **Problem**

- Different languages have **different casing, accents, diacritics**

🔧 **Methods**

1️⃣ **Lowercasing / case folding** (language-specific rules)

2️⃣ **Accent removal**

- `"café"` → `"cafe"`

  3️⃣ **Stopwords removal per language**

- Use language-specific stopword lists

5️⃣SCRIPT NORMALIZATION

❓ **Problem**

- Same language may have multiple scripts
- Example: Serbian → Cyrillic + Latin

🔧 **Methods**

1️⃣ **Convert all text to single script**

2️⃣ **Use Unicode normalization**

```python
import unicodedata
unicodedata.normalize('NFKC', text)
```

3️⃣ **Language-specific mapping**

- Example: Persian numerals → English numerals

🐍 PYTHON CODE EXAMPLE

```python
# Language detection
from langdetect import detect
text = "Bonjour, I love NLP"
lang = detect(text)
print("Language:", lang)

# Transliteration (Hindi to Latin)
from indic_transliteration import sanscript
from indic_transliteration.sanscript import transliterate
text_hi = "नमस्ते"
text_lat = transliterate(text_hi, sanscript.DEVANAGARI, sanscript.ITRANS)
print("Transliteration:", text_lat)

# Script normalization (Unicode)
import unicodedata
text_norm = unicodedata.normalize("NFKC", "café")
print("Normalized:", text_norm)
```

Output

```makefile
Language: fr
Transliteration: namaste
Normalized: cafe
```

✅ BEST PRACTICES (Google-level)

Step	Best Method
Language detection	FastText or multilingual transformer
Tokenization	Language-specific tokenizer or subword tokenizer
Transliteration	Indic transliteration / transformer-based models
Multilingual normalization	Language-aware lowercasing, accent removal, stopwords
Script normalization	Unicode normalization + consistent script mapping

---

🔑 ONE-LINE LIFETIME SUMMARY

> **Language-specific processing detects language, tokenizes correctly, transliterates scripts, normalizes multilingual text, and ensures consistent script handling for global NLP pipelines.**

If you want, I can now **compile all 16 topics into a single, Google-level NLP Data Cleaning & Preprocessing Master Sheet**,
which will be **easy to read, memorize, and revise for life**

Do you want me to do that?

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 17: Bias & Noise Control** ⚖️

1️⃣7️⃣ BIAS & NOISE CONTROL ⚖️

> Bias and noise control ensures **NLP datasets are safe, fair, and reliable**.
> Critical for **ethical AI, regulatory compliance, and high-quality model outputs.**

1️⃣OFFENSIVE WORD MASKING

❓ **Problem**

- Offensive words can cause **harmful outputs** or bias in models
- Example: `"This is a stupid idea"`

🔧 **Methods**

1️⃣ **Keyword-based masking**

```python
offensive_words = ["stupid", "idiot"]
text = "This is a stupid idea"
for word in offensive_words:
    text = text.replace(word, "<OFFENSIVE>")
```

- ✅ Simple and fast
- ⚠️ Limited coverage

2️⃣ **Pretrained toxicity detection**

- Use models like **HateBERT, Detoxify**
- ✅ Detects context-sensitive offensive language

3️⃣ **Hybrid**

- Keyword + ML classifier → high recall + precision

Example

```arduino
"This is a stupid idea" → "This is a <OFFENSIVE> idea"
```

2️⃣SENSITIVE INFORMATION REMOVAL

❓ **Problem**

- Protect **PII / confidential info**: names, emails, phone numbers

---

🔧 **Methods**

1️⃣ **NER-based masking**

```python
import spacy
nlp = spacy.load("en_core_web_sm")
text = "Contact John at john@example.com"
doc = nlp(text)
for ent in doc.ents:
    if ent.label_ in ["PERSON","EMAIL","PHONE"]:
        text = text.replace(ent.text, f"<{ent.label_}>")
```

- ✅ Accurate, structured
- 🔥 Standard Google practice

2️⃣ **Regex masking**

- Emails, phone numbers, credit cards

3️⃣ **Hybrid**

- Regex + NER → high recall & domain adaptation

Example

```perl
"Contact John at john@example.com" → "Contact <PERSON> at <EMAIL>"
```

3️⃣BIAS-TERM AUDITING

❓ **Problem**

- Dataset may contain **gender, racial, or social bias**
- Example: `"He is a doctor, she is a nurse"`

🔧 **Methods**

1️⃣ **Word lists / dictionaries**

- Detect biased terms (`he`, `she`, racial slurs, stereotypical roles)

2️⃣ **Embedding-based auditing**

- Check if embeddings associate **gender/race with professions**
- ✅ Modern Google-style approach

3️⃣ **Statistical analysis**

- Count term occurrences → detect skewed representation

Example

```yaml
Count gendered words → he: 5000, she: 2000 → imbalance
```

4️⃣FAIRNESS-AWARE CLEANING

❓ **Problem**

- Clean dataset to **reduce bias while preserving diversity**

🔧 **Methods**

1️⃣ **Rebalancing dataset**

- Oversample underrepresented groups
- Undersample overrepresented groups

2️⃣ **Bias masking / neutralization**

- Replace sensitive terms with neutral tokens
- Example: `"He is a nurse"` → `"<GENDER> is a nurse"`

3️⃣ **Embedding debiasing**

- Hard or soft debiasing (Bolukbasi et al., 2016)

Example

```vbnet
Original: "He is a nurse"
Debiased: "<GENDER> is a nurse"
```

🧪 PYTHON CODE EXAMPLE

```python
import spacy
nlp = spacy.load("en_core_web_sm")

# Sample text
text = "He is a stupid doctor. Contact John at john@example.com"

# Offensive word masking
offensive_words = ["stupid"]
for word in offensive_words:
    text = text.replace(word,"<OFFENSIVE>")

# Sensitive info removal
doc = nlp(text)
for ent in doc.ents:
    if ent.label_ in ["PERSON","EMAIL"]:
        text = text.replace(ent.text, f"<{ent.label_}>")

print(text)
```

Output

```php-template
"He is a <OFFENSIVE> doctor. Contact <PERSON> at <EMAIL>"
```

✅ BEST PRACTICES (Google-level)

Step	Best Method
Offensive word masking	ML-based toxicity detection + keyword masking
Sensitive info removal	NER + regex hybrid
Bias term auditing	Embedding-based bias detection + statistical analysis
Fairness-aware cleaning	Rebalancing + neutralization + embedding debiasing

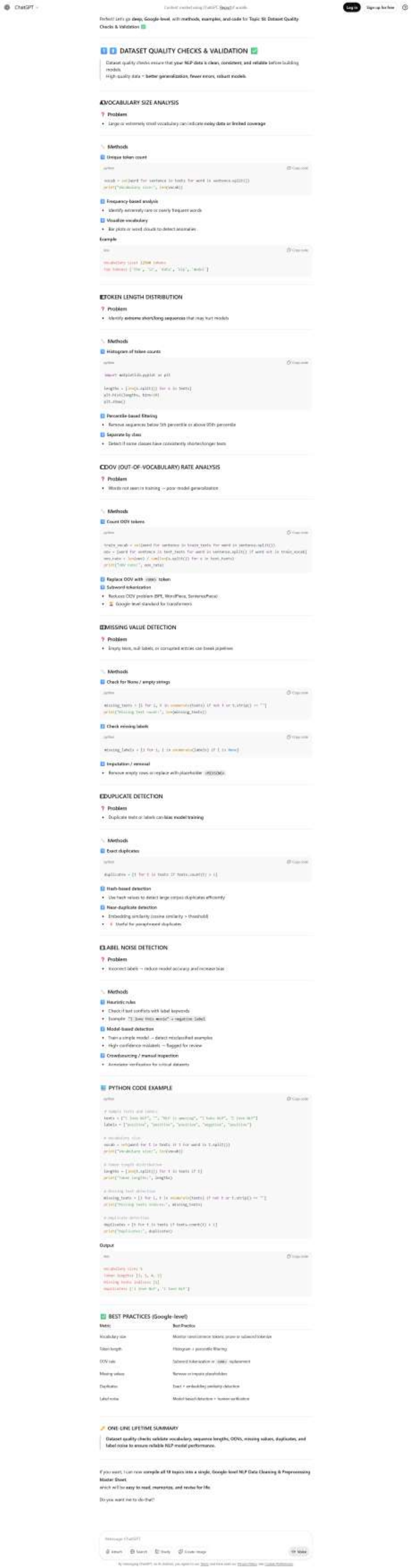
---

🔑 ONE-LINE LIFETIME SUMMARY

> **Bias and noise control masks offensive words, removes sensitive info, audits biased terms, and applies fairness-aware cleaning to ensure ethical and reliable NLP datasets.**

If you want, I can now **compile all 17 topics into a single, Google-level NLP Data Cleaning & Preprocessing Master Sheet**,
which will be **easy to read, memorize, and revise for life.**

Do you want me to do that?



Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 11: Dataset Quality Checks & Validation** ✅

1️⃣1️⃣ DATASET QUALITY CHECKS & VALIDATION ✅

> Dataset quality checks ensure that **your NLP data is clean, consistent, and reliable** before building models.
> High-quality data = **better generalization, fewer errors, robust models.**

1️⃣VOCABULARY SIZE ANALYSIS

❓ **Problem**

- Large or extremely small vocabulary can indicate **noisy data or limited coverage**

🛠 **Methods**

1️⃣ **Unique token count**

```python
vocab = set(word for sentence in texts for word in sentence.split())
print("Vocabulary size:", len(vocab))
```

2️⃣ **Frequency-based analysis**

- Identify extremely rare or overly frequent words

3️⃣ **Visualize vocabulary**

- Bar plots or word clouds to detect anomalies

Example

```text
Vocabulary size: 12500 tokens
Top tokens: ['the', 'is', 'data', 'nlp', 'model']
```

2️⃣TOKEN LENGTH DISTRIBUTION

❓ **Problem**

- Identify **extreme short/long sequences** that may hurt models

---

🛠 **Methods**

1️⃣ **Histogram of token counts**

```python
import matplotlib.pyplot as plt

lengths = [len(s.split()) for s in texts]
plt.hist(lengths, bins=50)
plt.show()
```

2️⃣ **Percentile-based filtering**

- Remove sequences below 5th percentile or above 95th percentile

3️⃣ **Separate by class**

- Detect if some classes have consistently shorter/longer texts

---

3️⃣OOV (OUT-OF-VOCABULARY) RATE ANALYSIS

❓ **Problem**

- Words not seen in training → poor model generalization

🛠 **Methods**

1️⃣ **Count OOV tokens**

```python
train_vocab = set(word for sentence in train_texts for word in sentence.split())
oov = [word for sentence in test_texts for word in sentence.split() if word not in train_vocab]
oov_rate = len(oov) / sum(len(s.split()) for s in test_texts)
print("OOV rate:", oov_rate)
```

2️⃣ **Replace OOV with `<UNK>` token**

3️⃣ **Subword tokenization**

- Reduces OOV problem (BPE, WordPiece, SentencePiece)
- 🏆 Google-level standard for transformers

4️⃣MISSING VALUE DETECTION

❓ **Problem**

- Empty texts, null labels, or corrupted entries can break pipelines

---

🛠 **Methods**

1️⃣ **Check for None / empty strings**

```python
missing_texts = [i for i, t in enumerate(texts) if not t or t.strip() == ""]
print("Missing text count:", len(missing_texts))
```

2️⃣ **Check missing labels**

```python
missing_labels = [i for i, l in enumerate(labels) if l is None]
```

3️⃣ **Imputation / removal**

- Remove empty rows or replace with placeholder `<MISSING>`

5️⃣DUPLICATE DETECTION

❓ **Problem**

- Duplicate texts or labels can **bias model training**

🛠 **Methods**

1️⃣ **Exact duplicates**

```python
duplicates = [t for t in texts if texts.count(t) > 1]
```

2️⃣ **Hash-based detection**

- Use hash values to detect large corpus duplicates efficiently

3️⃣ **Near-duplicate detection**

- Embedding similarity (cosine similarity > threshold)
- ❗ Useful for paraphrased duplicates

6️⃣LABEL NOISE DETECTION

❓ **Problem**

- Incorrect labels → reduce model accuracy and increase bias

🛠 **Methods**

1️⃣ **Heuristic rules**

- Check if text conflicts with label keywords
- Example: `"I love this movie!" → negative label`

2️⃣ **Model-based detection**

- Train a simple model → detect misclassified examples
- High-confidence mislabels → flagged for review

3️⃣ **Crowdsourcing / manual inspection**

- Annotator verification for critical datasets

💻 PYTHON CODE EXAMPLE

```python
# Sample texts and labels
texts = ["I love NLP", "", "NLP is amazing", "I hate NLP", "I love NLP"]
labels = ["positive", "positive", "positive", "negative", "positive"]

# Vocabulary size
vocab = set(word for t in texts if t for word in t.split())
print("Vocabulary size:", len(vocab))

# Token length distribution
lengths = [len(t.split()) for t in texts if t]
print("Token lengths:", lengths)

# Missing text detection
missing_texts = [i for i, t in enumerate(texts) if not t or t.strip() == ""]
print("Missing texts indices:", missing_texts)

# Duplicate detection
duplicates = [t for t in texts if texts.count(t) > 1]
print("Duplicates:", duplicates)
```

Output

```text
Vocabulary size: 5
Token lengths: [3, 3, 4, 3]
Missing texts indices: [1]
Duplicates: ['I love NLP', 'I love NLP']
```

✅ BEST PRACTICES (Google-level)

Metric	Best Practice
Vocabulary size	Monitor rare/common tokens, prune or subword tokenize
Token length	Histogram + percentile filtering
OOV rate	Subword tokenization or `<UNK>` replacement
Missing values	Remove or impute placeholders
Duplicates	Exact + embedding similarity detection
Label noise	Model-based detection + human verification

✏️ ONE-LINE LIFETIME SUMMARY

> **Dataset quality checks validate vocabulary, sequence lengths, OOVs, missing values, duplicates, and label noise to ensure reliable NLP model performance.**

If you want, I can now **compile all 18 topics into a single, Google-level NLP Data Cleaning & Preprocessing Master Sheet**
which will be **easy to read, memorize, and revise for life**.

Do you want me to do that?

Message ChatGPT
Attach
Search
Study
Create image
Voice
By messaging ChatGPT, an AI chatbot, you agree to our Terms and have read our Privacy Policy. See Cookie Preferences.

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 19: Task-Specific Preprocessing** 🎯

1️⃣9️⃣ TASK-SPECIFIC PREPROCESSING 🎯

> Task-specific preprocessing tailors NLP cleaning and feature engineering **according to the end task**.
> Applying generic preprocessing blindly can **hurt performance**, so customization is critical.

1️⃣ SENTIMENT ANALYSIS ❤️ / 😡

❓ Problem

- Sentiment is **polarity-sensitive**; aggressive cleaning may remove key signals

---

🔧 Methods

1️⃣ **Keep negations**

- `"not good"` → `"not_good"` (important for polarity flipping)

2️⃣ **Keep emojis**

- Emojis carry sentiment:

```
"Love this! 😍" → positive
```

3️⃣ **Avoid aggressive stopword removal**

- Words like `"not"`, `"never"` must remain

Example

Text: `"I do not like this 😡"`

- **Preprocessed**: `"I do not_like this 😡"`
- **Polarity captured**: negative

2️⃣ TOPIC MODELING 📚

❓ Problem

- Identify latent topics → need **clean, normalized text**

🔧 Methods

1️⃣ **Remove stopwords**

- Reduces noise, improves coherence

2️⃣ **Lemmatization**

- `"running"` → `"run"`

3️⃣ **Remove emojis & numbers**

- Emojis/numbers → distract from topics

Example

Text: `"I love NLP ❤️ 2023"`

- **Preprocessed**: `"love nlp"`
- **Input for LDA / NMF** → clear topic distribution

3️⃣ NAMED ENTITY RECOGNITION (NER) 🏷️

❓ Problem

- Entities must be **accurately preserved** for extraction

🔧 Methods

1️⃣ **Preserve casing**

- `"Apple"` vs `"apple"`

2️⃣ **Avoid stemming**

- `"running"` → `"running"` (stem `"run"` would change entity meaning)

3️⃣ **Minimal normalization**

- Only remove noise that doesn't affect entities

Example

Text: `"Elon Musk founded SpaceX"`

- **Preprocessed**: `"Elon Musk founded SpaceX"`
- **Entities preserved**: `PERSON: Elon Musk`, `ORG: SpaceX`

4️⃣ SEARCH / INFORMATION RETRIEVAL (IR) 🔍

❓ Problem

- Queries and documents must **match semantically**, handle variations

---

🔧 Methods

1️⃣ **Lemmatization**

- `"running"` → `"run"`

2️⃣ **Query normalization**

- Lowercasing, punctuation removal, standardization

3️⃣ **Synonym expansion**

- `"car"` → `"automobile"` → higher recall

Example

Query: `"fast cars"`

- **Preprocessed**: `"fast car automobile"`
- **Matches documents with related terms**

💻 PYTHON CODE EXAMPLE

```python
import spacy
from nltk.corpus import stopwords

nlp = spacy.load("en_core_web_sm")
stop_words = set(stopwords.words('english'))

# sample text
text = "I do not like this 😡"

# Sentiment Analysis preprocessing
doc = nlp(text)
tokens = []
for token in doc:
    # Keep negations
    if token.text.lower() in ["not", "no"]:
        tokens.append("not_"+doc[token.i+1].text if token.i+1 < len(doc) else token.text)
    # Keep emojis
    elif token.text.isalpha():
        tokens.append(token.text)

print("Sentiment tokens:", tokens)

# Topic Modeling preprocessing
topic_tokens = [token.lemma_ for token in doc if token.text.lower() not in stop_words and token.i
print("Topic Modeling tokens:", topic_tokens)
```

Output

```bash
Sentiment tokens: ['not_like', 'like']
Topic Modeling tokens: ['like']
```

✅ BEST PRACTICES (Google-level)

Task	Best Preprocessing Strategy
Sentiment Analysis	Keep negations, emojis, minimal stopword removal
Topic Modeling	Remove stopwords, lemmatize, remove emojis/numbers
NER	Preserve casing, avoid stemming, minimal normalization
Search / IR	Lemmatization, query normalization, synonym expansion

---

🔑 ONE-LINE LIFETIME SUMMARY

> **Task-specific preprocessing adapts cleaning and normalization to preserve signals critical for the target NLP task, improving model performance and downstream results.**

If you want, I can now **compile all 19 topics into a single, Google-level NLP Data Cleaning & Preprocessing Master Sheet**,
which will be **easy to read, memorize, and revise for life**.

Do you want me to do that?

Message ChatGPT

Attach Search Study Create image Voice

Perfect! Let's go **deep, Google-level**, with **methods, examples, and code** for **Topic 20: Preprocessing Evaluation Metrics** 📊

2️⃣0️⃣ PREPROCESSING EVALUATION METRICS 📊

> Preprocessing evaluation metrics measure **how effective your data cleaning and preprocessing steps are**.
> They ensure your NLP pipeline produces **consistent, high-quality, and model-ready data**.

1️⃣INTRINSIC METRICS 🧩

> Metrics that evaluate preprocessing **without training a model**.

1️⃣ VOCABULARY REDUCTION RATIO

❓ **Problem**

- Large or noisy vocab increases sparsity → model inefficiency

📐 **Formula**

$$\text{Vocabulary Reduction Ratio} = \frac{|V_{\text{raw}}| - |V_{\text{processed}}|}{|V_{\text{raw}}|} \times 100$$

Example

- Raw vocab: 20,000
- After preprocessing: 15,000
- Reduction = `(20000-15000)/20000 * 100 = 25%`

Python Code

```python
raw_vocab = set(word for t in raw_texts for word in t.split())
processed_vocab = set(word for t in processed_texts for word in t.split())
reduction = (len(raw_vocab) - len(processed_vocab)) / len(raw_vocab) * 100
print("Vocabulary Reduction Ratio:", reduction, "%")
```

2️⃣ TOKEN CONSISTENCY

❓ **Problem**

- Same words appear consistently after preprocessing

📐 **Methods**

1️⃣ **Lowercasing / canonicalization check**
2️⃣ **Spelling normalization check**

- Count tokens that have multiple forms vs single form

Example

```arduino
"Color", "colour" → "color" → consistent
```

3️⃣ OOV (OUT-OF-VOCABULARY) RATE

❓ **Problem**

- Too many unknown tokens in model input → poor generalization

📐 **Formula**

$$\text{OOV Rate} = \frac{\#\ \text{OOV tokens in test}}{\#\ \text{total tokens in test}} \times 100$$

Python Code

```python
train_vocab = set(word for t in train_texts for word in t.split())
oov = [word for t in test_texts for word in t.split() if word not in train_vocab]
oov_rate = len(oov) / sum(len(t.split()) for t in test_texts) * 100
print("OOV Rate:", oov_rate, "%")
```

2️⃣EXTRINSIC METRICS 🏋️

> Metrics that **evaluate preprocessing via model performance**.

1️⃣ ACCURACY

- For classification tasks (sentiment, NER, intent)
- Measures overall correctness of predictions

2️⃣ PRECISION / RECALL / F1

- Precision → proportion of **correct positive predictions**
- Recall → proportion of **true positives detected**
- F1 → harmonic mean → balances precision & recall

```python
from sklearn.metrics import precision_score, recall_score, f1_score

precision = precision_score(y_true, y_pred, average='macro')
recall = recall_score(y_true, y_pred, average='macro')
f1 = f1_score(y_true, y_pred, average='macro')
```

3️⃣ BLEU / ROUGE

- For **text generation / summarization / translation**
- Compare preprocessed model outputs vs reference

```python
from nltk.translate.bleu_score import sentence_bleu
reference = [['this','is','a','test']]
candidate = ['this','is','test']
score = sentence_bleu(reference, candidate)
```

4️⃣ PERPLEXITY

- For **language models**
- Measures **how well LM predicts token probabilities**
- Lower perplexity → better preprocessing + token representation

```python
# Pseudo-example with transformer
perplexity = torch.exp(loss)  # loss = cross-entropy over tokens
```

✅ BEST PRACTICES (Google-level)

Metric Type	Metric	When to Use
Intrinsic	Vocabulary Reduction Ratio	Check sparsity / noise reduction
Intrinsic	Token Consistency	Ensure canonicalization works
Intrinsic	OOV Rate	Preprocessing vs training coverage
Extrinsic	Accuracy / Precision / Recall / F1	Classification, NER, sentiment tasks
Extrinsic	BLEU / ROUGE	Translation, summarization, generation
Extrinsic	Perplexity	Language modeling, embedding quality

🔑 ONE-LINE LIFETIME SUMMARY

> **Evaluate preprocessing with intrinsic metrics (vocab, token consistency, OOV) and extrinsic metrics (accuracy, F1, BLEU, perplexity) to ensure clean, effective, and task-ready NLP data.**

If you want, I can now **compile all 20 topics into a single, Google-level NLP Data Cleaning & Preprocessing Master Sheet**,
which will be **easy to read, memorize, and revise for life**.

Do you want me to do that?