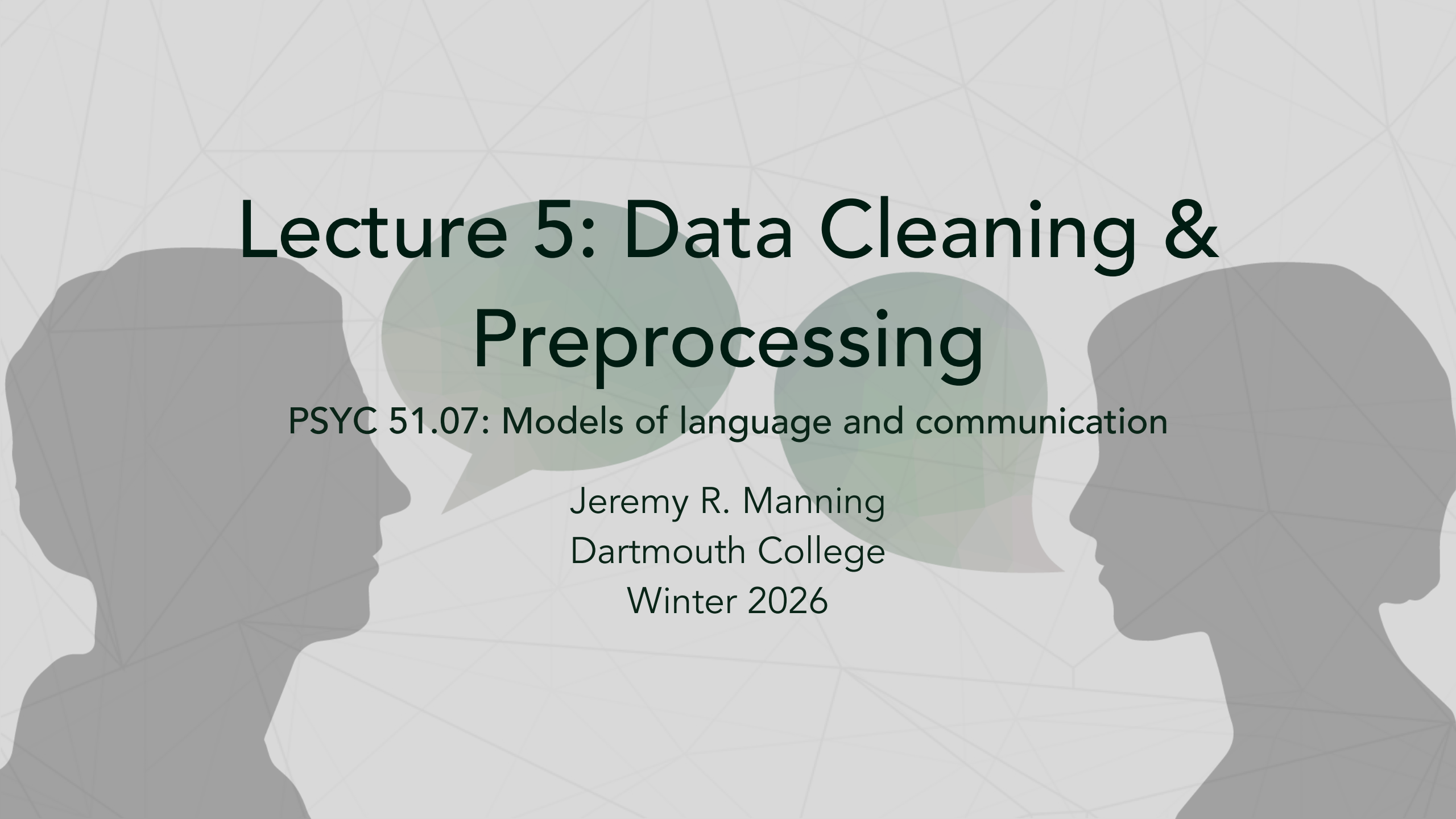


Lecture 5: Data Cleaning & Preprocessing

PSYC 51.07: Models of language and communication

Jeremy R. Manning
Dartmouth College
Winter 2026

Learning objectives

By the end of this lecture, you will be able to...

1. Understand why data cleaning is critical for NLP tasks
2. Apply common preprocessing techniques to raw text
3. Use web scraping to collect text data
4. Implement lemmatization and stemming
5. Make informed decisions about preprocessing strategies

Key theme

Garbage in, garbage out!

Real-world text is messy

Common problems:

- HTML/XML tags: `<p>text</p>`
- Special characters: &
- Inconsistent formatting
- Extra whitespace
- Mixed encodings (UTF-8, ASCII...)
- Emoji and Unicode characters

Consequences of dirty data:

- Models learn noise, not signal
- Reduced accuracy and consistency
- Poor generalization to new data
- Wasted computation on junk
- Unpredictable model behavior
- Difficulty debugging issues

Remember

"Quality of input = quality of output"

The data cleaning pipeline

A typical data cleaning pipeline

Important

Pipeline varies by task! Not all steps are always needed.

Cleaning messy text step by step

Raw input:

```
<p>I LOVE this!!!
https://ex.com
a@b.com</p>
```

Steps:

1. Remove HTML tags
2. Remove URLs/emails
3. Normalize whitespace

```python
import re
raw = "<p>I LOVE this!!!
https://ex.com a@b.com</p>"

text = re.sub(r'<.*?>', '', raw)
# Step 1
text = re.sub(r'http\S+', '',
text)    # Step 2
text = re.sub(r'\S+@\S+', '',
text)
text = ' '.join(text.split())
# Step 3

print(text)  # "I LOVE this!!!"
```

Worked example: Complete pipeline

Full transformation:

Stage	Text
Raw	`<p>I LOVE this product!!! https://ex.com</p>`
Remove HTML	`I LOVE this product!!! https://ex.com`
Remove URLs	`I LOVE this product!!!`
Normalize spaces	`I LOVE this product!!!`
Lowercase	`i love this product!!!`
Remove extra punctuation	`i love this product!`

Key decision points:

- Keep emoji ? **Yes** for sentiment analysis (conveys emotion)
- Keep punctuation !!!? **Maybe** (emphasis, but noisy)
- Lowercase? **Depends** on task (lose "LOVE" emphasis)

Different tasks need different preprocessing

Task	Keep	Remove
Sentiment analysis	Emoji, punctuation (style matters)	Extra whitespace, special chars
Named entity recognition	Case (proper nouns important)	Extra whitespace
Topic modeling	Content words only	Punctuation, case (lowercase all)

Key principle

Preserve information relevant to your task!

The web is a massive source of text data

What you can collect:

- News articles and blog posts
- Product reviews and ratings
- Social media and forum discussions
- Domain-specific technical content

Popular tools:

- **Beautiful Soup:** Parse HTML/XML
- **Requests:** Fetch web pages
- **Scrapy:** Full scraping framework
- **Selenium:** JavaScript-heavy sites

Be ethical

Check `robots.txt`, respect rate limits, and honor terms of service and copyright.

Web scraping with Beautiful Soup

Basic example:

```python
from bs4 import BeautifulSoup
import requests

# Fetch webpage
url = "https://example.com/article"
response = requests.get(url)
html_content = response.content

# Parse HTML
soup = BeautifulSoup(html_content, 'html.parser')

# Extract text from specific elements
title = soup.find('h1').get_text()
article = soup.find('article').get_text()
clean_text = ' '.join(article.split())  # Remove extra whitespace
print(f"Title: {title}\nArticle: {clean_text[:200]}...")
```

Advanced Beautiful Soup techniques

```
from bs4 import BeautifulSoup

html = """<div class="article">
 <h2>Breaking News</h2>
 <p class="content">First paragraph.</p>
 <p class="content">Second paragraph.</p>
 <div class="ads">Advertisement</div>
</div>"""

soup = BeautifulSoup(html, 'html.parser')

paragraphs = soup.find_all('p', class_='content')  # Find content paragraphs
text = ' '.join([p.get_text() for p in paragraphs])

for ad in soup.find_all('div', class_='ads'):      # Remove unwanted
```

Always use UTF-8 encoding

Common encoding issues:

- UTF-8 vs. ASCII vs. Latin-1
- Special characters: é, ñ, ü
- Emoji require Unicode support
- Mixed encodings in scraped data

Best practices:

- Default to UTF-8 for all files
- Detect unknown: `chardet` library
- Normalize Unicode: NFKC form
- Fix broken text: `ftfy` library

Web scraping often produces mixed encodings

```python
import chardet

raw_bytes = b'Caf\xe9 au lait \x96 delicious!'  # Unknown encoding

result = chardet.detect(raw_bytes)              # Detect encoding
print(f"Detected: {result}")  # {'encoding': 'Windows-1252', 'confidence': 0.73}

text = raw_bytes.decode(result['encoding'])     # Decode with detected encoding
print(text)                         # "Café au lait – delicious!"

import ftfy                                     # Alternative: fix broken Unicode
```

Pro tip: Save all files as UTF-8 to avoid these headaches!

Five common preprocessing steps

1. HTML/XML tag removal

- `<p>Hello</p>` → `Hello`

2. Whitespace normalization

- `"Hello world\n"` → `"Hello world"`

3. Punctuation handling

- Keep for sentiment ("Great!" vs "Great")
- Remove for topic modeling

4. Case normalization

- Lowercase: "Apple" = "apple"
- Preserve for NER: "Apple Inc."

5. Special character removal

- URLs, emails, numbers
- Task-dependent decisions

Preprocessing example

```python
import re

def preprocess_text(text):
    text = re.sub(r'http\S+|www.\S+', '', text)  # Remove URLs
    text = re.sub(r'\S+@\S+', '', text)          # Remove emails
    text = re.sub(r'<.*?>', '', text)            # Remove HTML tags
    text = ' '.join(text.split())                # Remove extra whitespace
    text = text.lower()                          # Lowercase
    return text

raw = "Check out https://example.com! <b>Amazing</b> deals!!"
print(preprocess_text(raw))  # "check out amazing deals!!"
```

Stemming is fast but crude; lemmatization is accurate

Stemming

- Rule-based crude chopping
- Fast and simple
- May produce non-words
- Porter Stemmer (1980)
- "running" → "run"

Lemmatization

- Uses vocabulary + morphology
- Slower but more accurate
- Always produces real words
- Requires POS context
- "better" → "good"

When to use which?

- **Stemming:** Speed matters, approximate matching OK
- **Lemmatization:** Need interpretable, real words

Lemmatization produces real words; stemming produces fragments

Word	Porter Stemmer	Lemmatizer	Notes
ran	ran	run	Lemma recognizes irregular verb
runs	run	run	Both work well
running	run	run	Both work well
runner	runner	runner	Both keep as-is (different word)
better	better	good	Lemma handles irregular adjective
best	best	good	Lemma handles superlative
geese	gees	goose	Stemmer produces non-word!
studies	studi	study	Stemmer produces non-word!

continued...

Lemmatization produces real words; stemming produces fragments

Word	Porter Stemmer	Lemmatizer	Notes
organizing	organ	organize	Stemmer too aggressive!

Key insight: Lemmatization produces real words; stemming can produce fragments.

...continued

Stemming can over-stem or under-stem

```python
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()

# Over-stemming: Different words become the same
words = ['universe', 'university', 'universal']
stems = [stemmer.stem(w) for w in words]
print(stems) # ['univers', 'univers', 'univers']
# All three map to the same stem - meaning is lost!

# Under-stemming: Same root stays different
words2 = ['absorb', 'absorption']
```

Takeaway: Use lemmatization when you need interpretable, meaningful tokens.

Stemming with NLTK

```python
from nltk.stem import PorterStemmer, SnowballStemmer
porter = PorterStemmer()
snowball = SnowballStemmer('english')

words = ['running', 'runs', 'runner', 'ran', 'easily', 'fairly']

print("Word          Porter       Snowball")
for word in words:
    print(f"{word:12} {porter.stem(word):12} {snowball.stem(word)}")

# Output:
# running      run          run
# runs         run          run
# runner       runner       runner
# ran          ran          ran
```

Lemmatization with spaCy

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

text = "The cats are running faster than dogs ran yesterday"
doc = nlp(text)

print(f"{'Word':<12} {'Lemma':<12} {'POS'}")
print("-" * 36)
for token in doc:
    print(f"{token.text:<12} {token.lemma_:<12} {token.pos_}")

# Output:
# Word         Lemma        POS
# cats         cat          NOUN
# running      run          VERB
```

spaCy vs. NLTK for lemmatization

Feature	spaCy	NLTK
Accuracy	High	Good
Setup	Easy	Requires data download
POS tagging	Built-in	Separate step
Dependencies	Included	Manual WordNet
Use case	Production	Research/Teachin

Stop words: common words with little semantic value

When to remove:

- Topic modeling
- Text classification
- Search engines
- Information retrieval

When to keep:

- Sentiment ("not good" ≠ "good")
- Machine translation
- Text generation
- Neural models (they learn!)

Examples

"the", "a", "is", "in", "and", "or" — language-specific lists available in NLTK and spaCy.

Stop words in practice

```
from nltk.corpus import stopwords
import spacy

stop_words_nltk = set(stopwords.words('english'))  # NLTK approach
text = "This is an example showing stop word removal"
filtered_nltk = [w for w in text.lower().split() if w not in
stop_words_nltk]
print("NLTK:", filtered_nltk)  # ['example', 'showing', 'stop', 'word',
'removal']

nlp = spacy.load("en_core_web_sm")  # spaCy approach
doc = nlp(text)
filtered_spacy = [token.text for token in doc if not token.is_stop]
print("spaCy:", filtered_spacy)  # ['example', 'showing', 'stop',
```

Sentiment analysis requires preserving emotional signals

> **Amazon review**
>
> *"I LOVE this product!!! Best purchase ever! See details at http://example.com"*

Keep:

- Punctuation (! conveys emphasis)
- Capitalization (LOVE = strong)
- Emoji if present

Remove:

- URLs (no sentiment value)
- HTML tags
- Extra punctuation (!!! → !)

Discussion: Preprocessing trade-offs

Questions to consider

1. **Information loss:** What do we lose when we lowercase everything?
   - Think: "US" (United States) vs. "us" (pronoun)
2. **Task dependency:** Why does preprocessing differ by task?
   - Hint: What signals matter for sentiment vs. topic modeling?
3. **Modern models:** Do transformer models still need heavy preprocessing?
   - Consider: BERT handles subwords, capitalization, punctuation...
4. **Bias introduction:** Can preprocessing introduce bias?
   - Example: Removing slang might remove cultural markers

Practical tips for data cleaning

Best practices

1. **Inspect your data first!** — Look at samples before deciding on preprocessing
2. **Keep raw data separate** — Never overwrite originals; you might need them!
3. **Document your pipeline** — Track what preprocessing you applied and why
4. **Experiment!** — Try different approaches, measure impact on task
5. **Validate incrementally** — Check results after each preprocessing step
6. **Consider automation** — Use libraries: spaCy, NLTK, HuggingFace tokenizers

Complete preprocessing pipeline example

```python
import spacy, re

class TextPreprocessor:
    def __init__(self, remove_stopwords=False):
        self.nlp = spacy.load("en_core_web_sm")
        self.remove_stopwords = remove_stopwords

    def clean(self, text):
        text = re.sub(r'http\S+', '', text)   # Remove URLs
        text = re.sub(r'<.*?>', '', text)     # Remove HTML
        text = ' '.join(text.split())         # Normalize whitespace
        doc = self.nlp(text)
        tokens = [t.lemma_ for t in doc if not (self.remove_stopwords and t.is_stop)]
        return ' '.join(tokens)
```

Preprocessing checklist

Before starting your project, ask...

- ☐ What is my task? (classification, generation, extraction...)
- ☐ What signals are important? (sentiment, topics, entities...)
- ☐ Should I lowercase? (preserve case for names?)
- ☐ How to handle punctuation? (keep for emotion?)
- ☐ Do I need lemmatization? (or will model handle it?)
- ☐ Should I remove stop words? (or keep for grammar?)
- ☐ How to handle special characters? (emoji, numbers, symbols)
- ☐ What about rare/unknown words? (keep, remove, replace?)
- ☐ Have I validated on sample data? (inspect before/after!)

Remember

There's no one-size-fits-all solution!

Tools and libraries

Web Scraping:

- Beautiful Soup
- Scrapy

Text Processing:

- spaCy
- NLTK
- TextBlob

Encoding:

- chardet: Character encoding detection
- ftfy: Fixes broken Unicode

HuggingFace Resources:

- Chapter 3.2: Processing Data
- Chapter 2.4: Tokenizers

Primary references

Classic papers

- Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, 14(3), 130-137. — The original Porter Stemmer algorithm
- Manning, C. D., Raghavan, P., & Schutze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. — Chapter 2: Text preprocessing fundamentals

Modern resources:

- spaCy documentation: Industrial-strength NLP
- HuggingFace NLP Course: Modern preprocessing with transformers

Ethical considerations:

- Liang et al. (2020). "Towards Debiasing Sentence Representations"
- Consider how preprocessing choices affect fairness

Hands-on exercise

Try this yourself

1. Choose a website (news, blog, Reddit...)
2. Scrape 10-20 articles/posts
3. Apply different preprocessing pipelines:
   - **Minimal:** Just remove HTML
   - **Moderate:** + normalize whitespace, lowercase
   - **Heavy:** + lemmatize, remove stop words
4. Compare the results:
   - How does vocabulary size change?
   - What information is lost/preserved?
   - Which would work best for sentiment analysis? Topic modeling?

Bonus

Share interesting findings with classmates!

Key takeaways

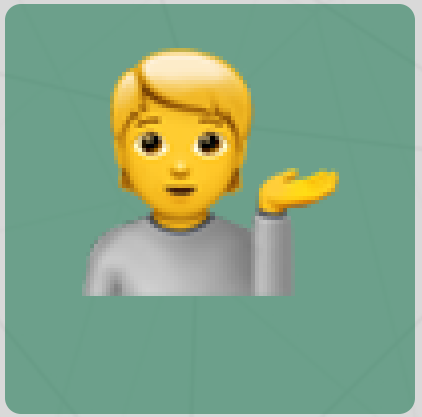
1. **Preprocessing is crucial** but task-dependent — No universal pipeline; adapt to your needs!
2. **Balance cleaning vs. information loss** — More preprocessing does not always mean better
3. **Stemming vs. Lemmatization** — Stemming: fast, approximate; Lemmatization: slow, accurate
4. **Modern models are robust** — Transformers can handle messy text better than older models
5. **Always validate** — Inspect data before and after preprocessing

> **Coming up**
>
> Tokenization deep dive!

Questions? Want to chat more?

Email me

Join our Discord



Come to office hours

Next up

Lecture 6 — Tokenization deep dive!